Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS'07)

Isabelle Puaut, Nicolas Navet, Françoise Simonot-Lion

▶ To cite this version:

Isabelle Puaut, Nicolas Navet, Françoise Simonot-Lion (Dir.). Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS'07). Isabelle Puaut and Nicolas Navet and Françoise Simonot-Lion. Institut National Polytechnique de Lorraine - Atelier de Reprographie, pp.238, 2007, 2-905267-53-4. inria-00168530

HAL Id: inria-00168530

https://inria.hal.science/inria-00168530v1

Submitted on 28 Aug 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Proceedings of the

15th International Conference on Real-Time and Network Systems

RTNS'07

LORIA, Nancy, France

29-30 March 2007

http://rtns07.irisa.fr

Table of contents

15th conference on Real-Time and Network Systems RTNS'07

Message from the conference chairs vii
Organizing committee viii
Program committee ix
Reviewers x

Keynote presentation

Implementation and Challenging Issues of Flash-Memory Storage Systems........ xi
Tei-Wei Kuo, National Taiwan University, Taiwan

Formal methods

The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems........ 15
Martin Ouimet, Massachusetts Institute of Technology, USA
Kristina Lundqvist, Massachusetts Institute of Technology, USA
Mikael Nolin, Maelardalen University, Sweden

Extended Real-Time LOTOS for Preemptive Systems Verification........ 25
Tarek Sadani, LAAS-CNRS / Ensica, France
Pierre de Saqui-Sannes, LAAS-CNRS / Ensica, France
Jean-Pierre Courtiat, LAAS-CNRS, France

Generation of Tests for Real-time Systems with Test Purposes 35
Sebastien Salva, LIMOS, France
Patrice Laurençot, LIMOS, France

Architectures and worst-case execution time estimation

Predictable Performance on Multithreaded Architectures for Streaming Protocol Processing........ 47
Matthias Ivers, Institut für Datentechnik und Kommunikationsnetze, Germany
Bhavani Janarthanan, Institut für Datentechnik und Kommunikationsnetze, Germany
Rolf Ernst, Institut für Datentechnik und Kommunikationsnetze, Germany

A Context Cache Replacement Algorithm for Pfair Scheduling........ 57
Kenji Funaoka, Keio University, Japan
Shinpei Kato, Keio University, Japan
Nobuyuki Yamasaki, Keio University, Japan

Exact Cache Characterization by Experimental Parameter Extraction 65

Tobias John, Chemnitz University of Technology, Germany
Robert Baumgartl, Chemnitz University of Technology, Germany

Towards Predictable, High-Performance Memory Hierarchies in Fixed-Priority Preemptive Multitasking Real-Time Systems *75*
Eugenio Tamura, Pontificia Universidad Javeriana, Cali, Colombia
José Vicente Busquets-Mataix, Universidad Politécnica de Valencia, Spain
Antonio Martí Campoy, Universidad Politécnica de Valencia, Spain

On the Sensitivity of WCET Estimates to the Variability of Basic Blocks Execution Times *85*
Hugues Cassé, IRIT, Toulouse, France
Christine Rochange, IRIT, Toulouse, France
Pascal Sainrat, IRIT, Toulouse, France

Scheduling 1

Efficient Computation of Response Time bounds under Fixed-Priority Scheduling *95*
Enrico Bini, Scuola Superiore Sant'Anna, Italy
Sanjoy Baruah, University of North Carolina, USA

Approximate Feasibility Analysis and Response-Time Bounds of Static-Priority Tasks with Release Jitters.. .... *105*
Pascal Richard, LISI/ENSMA, France
Joel Goossens, Universite Libre de Bruxelles, Belgium
Nathan Fisher, University of North Carolina, Chapel Hill, USA

Schedulability Analysis using Exact Number of Preemptions and no Idle Time for Real-Time Systems with Precedence and Strict Periodicity Constraints *113*
Patrick Meumeu Yomsi, INRIA Rocquencourt, France
Yves Sorel, INRIA Rocquencourt, France

Algorithm and Complexity for the Global Scheduling of Sporadic Tasks on Multiprocessors with Work-Limited Parallelism *123*
Sebastien Collette, Université Libre de Bruxelles, Belgium
Liliana Cucu, Université Libre de Bruxelles, Belgium
Joel Goossens, Université Libre de Bruxelles, Belgium

Scheduling 2

Schedulability Analysis of OSEK/VDX Applications *131*
Pierre-Emmanuel Hladik, LINA, France
Anne-Marie Deplanche, IRCCyN, France
Sebastien Faucou, IRCCyN, France
Yvon Trinquet, IRCCyN, France

Improvement of the Configuration and the Analysis of Posix 1003.1b Scheduling *141*
Mathieu Grenier, LORIA, France
Nicolas Navet, LORIA, France

An Extended Scheduling Protocol for the Enhancement of RTDBSs Performances *151*

Samy Semghouni , Laboratoire Informatique L.I.T.I.S antenne du Havre, France
Bruno Sadeg, Laboratoire Informatique L.I.T.I.S antenne du Havre, France
Laurent Amanton, Laboratoire Informatique L.I.T.I.S antenne du Havre, France
Alexandre Berred, Laboratoire de Mathématiques Appliquées du Havre, France

Scheduling and control

Comparative Assessment and Evaluation of Jitter Control Methods *163*
Giorgio Buttazzo, Scuola Superiore S. Anna, Pisa, Italy
Anton Cervin, University of Lund, Sweden

Reducing Delay and Jitter in Software Control Systems *173*
Hoai Hoang, Halmstad University, Halmstad, Sweden
Giorgio Buttazzo, Scuola Superiore S. Anna, Pisa, Italy

Task Handler Based on (m,k)-firm Constraint Model for Managing a Set of Real-Time Controllers183
Ning Jia, LORIA - University of Nancy, France
YeQiong Song, LORIA - University of Nancy, France
Francoise Simonot-Lion, LORIA - University of Nancy, France

Networks and distributed systems

Interface Design for Real-Time Smart Transducer Networks - Examining COSMIC, LIN, and TTP/A as Case Study *195*
Wilfried Elmenreich, Vienna University of Technology, Germany
Hubert Piontek, University of Ulm, Germany
Jörg Kaiser, Otto-von-Guericke-University Magdeburg, Germany

Delay-Bounded Medium Access for Unidirectional Wireless Links *205*
Björn Andersson, Institute Polytechnic Porto, Portugal
Nuno Pereira, Institute Polytechnic Porto, Portugal
Eduardo Tovar, Institute Polytechnic Porto, Portugal

Tolerating Arbitrary Failures in a Master-Slave Clock-Rate Correction Mechanism for Time-Triggered Fault-Tolerant Distributed Systems with Atomic Broadcast *215*
Astrit Ademaj, Vienna University of Technology, Real-Time Systems Group, Austria
Alexander Hanzlik, Vienna University of Technology, Real-Time Systems Group, Austria
Hermann Kopetz, Vienna University of Technology, Real-Time Systems Group, Austria

Exploiting Slack for Scheduling Dependent, Distributable Real-Time Threads in Unreliable Networks *225*
Kai Han, Virginia Tech, USA
Binoy Ravindran, Virginia Tech, USA
Douglas Jensen, Mitre Inc. , USA

Message from the conference chairs

RTNS'07

It is our great pleasure to welcome you to the fifteenth Conference on Real-Time and Network Systems (RTNS'07) in Nancy, France. The primary purpose of RTNS is to provide the participants, academic researchers or practitioners, with a forum to disseminate their work and discuss emerging lines of research in the area of real-time and network systems: real-time system design and analysis, infrastructure and hardware for real-time systems, software technologies and applications.

The first thirteenth issues of the conference were held within the "Real Time Systems" trade show in Paris (at first, Palais des Congrès Porte Maillot, then Paris Expo, porte de Versailles). In 2005, it was decided to make the conference independent of the exhibition, emphasize the role of Systems on Networks (hence the transformation of the name from RTS to RTNS), and switch to English as the official language of the conference.

In response to the call for papers, 42 papers were submitted, among which 22 were selected by the international Program Committee. The presentation are organized in 6 sessions covering all major aspects of real-time systems: task scheduling (2 sessions), scheduling and control, formal methods, architecture and worst-case execution time estimation, real-time networks and distributed systems. In addition to the contributed papers, the RTNS technical program has the privilege to include a keynote talk by Professor Tei-Wei Kuo, from the National Taiwan University, who will share his views on the challenging issues raised by the use of flash-memory storage systems in embedded real-time systems.. Furthermore, the second edition of the "Junior Researcher Workshop on Real-Time Computing" is held in conjunction with RTNS, and is a good opportunity for young researchers to present and get feedback on their ongoing work in a relaxed and stimulating atmosphere. All these presentations will provide an excellent snapshot of the current research results and directions in the area of real-time systems, and will certainly make RTNS a successful event.

Credit for the quality of the program is of course to be given to the authors who submitted high-quality papers and the program committee members and external referees who gave their time and offer their expertise to provide excellent reviews (at least three per paper). We are sincerely grateful to all of them.

RTNS'07 would not be possible without the generous contribution of many volunteers and institutions. First, we would like to express our sincere gratitude to our sponsors for their financial support : Conseil Général de Meuthe et Moselle, Conseil Régional de Lorraine, Communauté Urbaine du Grand Nancy, Université Henri Poincaré, Institut National Polytechnique de Lorraine and LORIA and INRIA Lorraine. We are thankful to Pascal Mary for authorizing us to use his nice picture of "place Stanislas" for the proceedings and web site (many others are available at www.laplusbelleplacedumonde.com). Finally, we are most grateful to the local organizing committee that helped to organize the conference. Let us hope for a bright future in the RTNS conference series !

Nicolas Navet, *INRIA-Loria, Nancy, France*
Françoise Simonot-Lion, *LORIA-INPL, Nancy, France*
General co-chairs

Isabelle Puaut, *University of Rennes / IRISA, France*
Program chair

Organizing Committee

General co-chairs

Nicolas Navet, *INRIA-Loria, Nancy, France*
Françoise Simonot-Lion, *LORIA-INPL, Nancy, France*

Program chair

Isabelle Puaut, *University of Rennes – IRISA, Rennes, France*

Junior workshop on real-time systems

Liliana Cucu, *LORIA-INPL, Nancy, France*

Local arrangements

Laurence Benini, *INRIA, France*
Najet Boughanmi, *INRIA-Loria, France*
Anne-Lise Charbonnier, *INRIA, France*
Liliana Cucu, LORIA-INPL, *Nancy, France*
Jean-François Deverge, *IRISA, Rennes, France*
Mathieu Grenier, *INRIA-Loria, France*
Christophe Païs, *IRISA, Rennes, France*
Xavier Rebeuf, *LORIA-INPL, France*
Olivier Zendra, *INRIA-Loria, France*

Program Committee

RTNS'07

Sanjoy Baruah, *University of North Carolina, USA*
Guillem Bernat, *University of York, UK*
Enrico Bini, *Scuola Superiore Sant'Anna, Pisa, Italy*
Alfons Crespo, Polytechnic *University of Valencia, Spain*
Jean-Dominique Decotignie, *EPFL, Lausanne, Switzerland*
Anne-Marie Déplanche, *IRCCyN, Nantes, France*
Jose. A. Fonseca, *University of Aveiro, Portugal*
Josep M. Fuertes, *Technical University of Catalonia, Spain*
Joel Goossens, *ULB, Brussels, Belgium*
Guy Juanole, *LAAS, Toulouse, France*
Joerg Kaiser, *University of Magdeburg, Germany*
Raimund Kirner, *TU Vienna, Austria*
Tei-Wei Kuo, *National Taiwan University, Taiwan*
Lucia Lo Bello, *University of Catania, Italy*
Zoubir Mammeri, *IRIT/UPS Toulouse, France*
Philippe Marquet, *INRIA/LIFL, Lille, France*
Pascale Minet, *INRIA-Rocquencourt, France*
Nicolas Navet, *INRIA-Loria, Nancy, France*
Nimal Nissanke, *London South Bank University, UK*
Mikael Nolin, *Mälardalen University, Sweden*
Marc Pouzet, *Université Paris Sud-LRI, France*
Pascal Richard, *LISI / Poitiers, France*
Guillermo Rodríguez-Navas, *University of Balearic Islands, Palma de Mallorca, Spain*
Bruno Sadeg, LITIS - *University of Le Havre, France*
Maryline Silly-Chetto, *IRCCyN, Nantes, France*
Daniel Simon, *INRIA-Rhônes Alpes, France*
Françoise Simonot-Lion, *LORIA-INPL, Nancy, France*
Eduardo Tovar, Polytechnic *Institute of Porto, Portugal*
Yvon Trinquet, *IRCCyN, Nantes, France*
Francisco Vasques, *University of Porto, Portugal*
François Vernadat, *LAAS, Toulouse, France*
Laurence T. Yang, *St. Francis Xavier University, Canada*

Reviewers

RTNS'07

Slim Abdellatif
Bjorn Andersson
Patricia Balbastre
Sanjoy Baruah
Mongi BenGaïd
Guillem Bernat
Marko Bertogna
Enrico Bini
Hadrien Cambazard
Chien-Wie Chen
Annie Choquet-Geniet
Yuan-Sheng Chu
Michele Cirinei
Alfons Crespo
Jean-Dominique Decotignie
Anne-Marie Déplanche
Andreas Ermedahl
Sébastien Faucou
Mamoun Filali
Jose A. Fonseca
Josep M. Fuertes
Joel Goossens
Emmanuel Grolleau
Pierre-Emmanuel Hladik
Ping-Yi Hsu
Guy Juanole
Joerg Kaiser
Thomas Kiebel
Raimund Kirner
Anis Koubaa
Tei-Wei Kuo
Didier Lime
Jian-Hong Lin
Lucia Lo Bello
Zoubir Mammeri
Philippe Marquet
Pascale Minet
Ricardo Moraes
Alexandre Mota
Nicolas Navet
Nimal Nissanke
Mikael Nolin
Harald Paulitsch
Marc Pouzet
Peter Puschner
Pascal Richard
Bernhard Rieder
Guillermo Rodríguez-Navas
Bruno Sadeg
Michael Schulze
Maryline Silly-Chetto
Daniel Simon
Françoise Simonot-Lion
Klaus Steinhammer
Eduardo Tovar
Yvon Trinquet
Francisco Vasques
François Vernadat
Ingomar Wenzel
Laurence T. Yang

Keynote presentation

Implementation and Challenging Issues of Flash-Memory Storage Systems

Tei-Wei Kuo

National Taiwan University

Flash memory is widely adopted in the implementations of storage systems, due to its nature in excellent performance, good power efficiency, and superb vibration tolerance. However, engineers face tremendous challenges in system implementations, especially when the capacity of flash memory is expected to increase significantly in the coming years. This talk will address the implementation issues of flash-memory storage systems, such as performance and management overheads. Summary on existing solutions will be presented, and future challenges will also be addressed.

Prof. Tei-Wei Kuo received the B.S.E. degree in Computer Science and Information Engineering from National Taiwan University in Taipei, Taiwan, ROC, in 1986. He received the M.S. and Ph.D. degrees in Computer Sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently a Professor and the Chairman of the Department of Computer Science and Information Engineering, National Taiwan University. Since February 2006, he also serves as a Deputy Dean of the National Taiwan University. His research interests include embedded systems, real-time operating systems, and real-time database systems. He has over 140 technical papers published or been accepted in journals and conferences and a number of patents. Prof. Kuo works closely with the industry and serves as a review committee member of several government agencies and research/development institutes in Taiwan.

Dr. Kuo serves as the Program Chair of IEEE Real-Time Systems Symposium (RTSS) in 2007 and a Program Co-Chair of the IEEE Real-Time Technology and Applications Symposium (RTAS) in 2001. He is also an Associate Editor of several journals, such as the Journal of Real-Time Systems (since 1998) and IEEE Transactions on Industrial Informatics (since 2007). Dr. Kuo also serves as the Steering Committee Chair of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) since 2005. He is an Executive Committee member of the IEEE Technical Committee on Real-Time Systems (TC-RTS) since 2005. Dr. Kuo received several prestigious research awards in Taiwan, including the Distinguished Research Award from the ROC National Science Council in 2003, and the ROC Ten Outstanding Young Persons Award in 2004 in the category of scientific research and development.

Formal methods

The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems

Martin Ouimet, Kristina Lundqvist, and Mikael Nolin
Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
{mouimet, kristina}@mit.edu, mikael.nolin@mdh.se

Abstract

We present a novel language for specifying real-time systems. The language addresses a key challenge in the design and analysis of real-time systems, namely the integration of functional and non-functional properties into a single specification language. The non-functional properties that can be expressed in the language include timing behavior and resource consumption. The language enables the creation of executable specifications with well-defined execution semantics, abstraction mechanisms, and composition semantics. The language is based on the theory of abstract state machines. Extensions to the theory of abstract state machines are presented to enable the explicit specification of non-functional properties alongside functional properties. The theory is also extended to define the execution semantics for the hierarchical and parallel composition of specifications. The features of the specification language are demonstrated using a light switch example and the Production Cell case study.

1 Introduction

The benefits and drawbacks of using formal methods have been documented heavily [10, 13]. Cited benefits include the detection of defects early in the engineering cycle, precise and concise specifications , and automated analysis [29]. Cited drawbacks include the heavy use of arcane mathematical notations, the lack of scalability of most methods, and the large investment typically required to use formal methods [13]. Besides the negative connotation that the term "formal methods" has taken in some circles, the benefits of unambiguous specifications and automated analysis during the early phases of the lifecycle have been generally accepted [5].

In the design and development of reactive real-time systems, the design and specification problem is more challenging than for traditional interactive systems because both functional behavior and non-functional behavior are part of the system's utility and must be specified precisely and concisely [9]. Furthermore, the specification and analysis of system designs is often performed at various levels of abstraction [17]. For example, the non-functional properties of system architectures can be specified and analyzed using an Architecture Description Language (ADL) such as the Society of Automotive Engineers' Architecture Analysis and Design Language (AADL) [28]. At the software application level, functional behavior can be specified and analyzed using state-transition systems such as finite state automata [18] or Petri nets [11]. Timing behavior can be specified and analyzed at either level using special purpose languages such as real-time logic or through specialized methods such as rate-monotonic analysis [12]. The need for multiple languages to specify and analyze system behavior can be expensive and error-prone because there is no formal connection between the different specifications resulting from the use of multiple languages [17]. This situation leads to redundant specifications that may not be consistent with one another.

This paper introduces the Timed Abstract State Machine (TASM) specification language, a novel specification language that removes the need to use many other specification languages. More specifically, TASM incorporates the specification of functional and non-functional behavior into a unified formalism. Furthermore, TASM is based on the theory of abstract state machines, a method for system design that can be applied at various levels of abstraction [8]. The TASM language has formal semantics, which makes its meaning precise and enables executable specifications.

The motivations and benefits of using Abstract State Machines (ASM), formerly known as evolving algebras, for hardware and software design have been documented in [6]. On the practical-side, ASMs have been used successfully on a wide range of applications, ranging from hardware-software systems to high-level system design [8]. Furthermore, there is enough evidence to believe that ASMs could provide a *literate* specification language, that is, a language that is understandable and usable without extensive mathematical training [13].

The anecdotal evidence supporting the success of the ASM method [6] suggests that tailoring the formalism to the area of reactive real-time systems could achieve similar benefits. The work presented in this paper extends the ASM formalism to make it amenable to real-time system specification. More specifically, the ASM formalism is extended to enable the explicit specification of timing behavior and resource consumption behavior. The resulting specification language, The Timed Abstract State Machine (TASM) language enables the specification of functional and non-functional properties into a unified formalism. The TASM language provides executable specifications that can express both sequential and concurrent behavior.

This paper is divided into five sections in addition to this Introduction. The following section situates the present work in relation to other research on similar topics. The abstract state machine formalism is introduced in section 3. Section 4 explains the modifications that have been made to the presented formalism to make it amenable to specification of real-time systems. Each extension is illustrated through the use of a light switch example. Section 5 provides a more substantial example of the feature of the TASM language through the production cell case study [19]. Finally, the Conclusion and Future Work section, Section 6, summarizes the contributions of the research and explains the additions that are to come in future development.

2 Related Work

In the academic community, there are numerous mathematical formalisms that have been proposed for specifying and analyzing real-time systems. The formalism presented in this paper is similar to the timed transition systems formalism presented in [16]. The two formalisms differ in the concurrency semantics since timed transition systems adopt an interleaving model whereas ASM theory adopts a general model of concurrency [8]. The most popular formalisms developed in academia can be classified into three main families: automata, process algebra, and Petri nets [3].

In the automata family, timed automata are finite state automata extended with real-valued clocks and communication channels. The formalism has been used on a variety of applications and is the formalism used in the model checker UPPAAL [18]. The formalism is well-suited for analysis by model-checking, but the lack of structuring mechanisms makes abstraction and encapsulation difficult to achieve [4]. Statecharts and the associated tool STATEMATE [15] augment automata with structuring mechanisms (superstates). Statecharts also include time concepts through the use of delays and timers.

In the Petri net family, a large number of variations on the traditional Petri net model have been developed, including various models of time [11]. Non-determinism is an essential part of Petri nets, which makes Petri net unsuitable for the specification of safety-critical real-time systems where predictability is of highest importance [4].

In the process algebra family, various offsprings of Communicating Sequential Processes (CSP) [2] and the Calculus of Communicating Systems (CCS) [20] have been defined, including multiple versions of timed process algebra [2]. However, in this formalism, it is difficult to express non-functional properties other than time (e.g., resource consumption). Timed LOTOS (ET-LOTOS) [2] is an example of a language from the process algebra family. Other well known formalisms include the Synchronous languages ESTEREL and LUSTRE [4].

In the industrial community, especially in the aerospace and automotive industries, the Unified Modeling Language (UML) [21] and the Architecture Analysis and Design Language (AADL) [28] have come to dominate notational conventions. At its onset, UML did not have formal semantics and remained a graphical language with limited support for automated analysis. Since its inception, many tools have defined their own semantics for UML, but the international standard [21] still does not contain a standard definition of the formal semantics. AADL contains formal semantics but is still in the early development stages, so it could not be completely evaluated. It is also unclear whether AADL can be used to specify low level functional behavior or if it is only applicable to architectural reasoning.

In the abstract state machine community, ASMs have been used to model specific examples of real-time systems [7, 14]. Some extensions have been proposed to the ASM theory to include timing characteristics [27] but the extensions make no mention of how time is to be specified (only the theoretical semantics are proposed) and do not address concurrency. The composition extensions for ASMs presented in this paper are based on the XASM language [1]. The XASM language does not include time or resource specification and only deals with single agent ASMs. The specification of resource consumption has not been addressed in the ASM community.

3 The Abstract State Machine (ASM) Formalism

The abstract state machine formalism revolves around the concepts of an abstract machine and an abstract state. System behavior is specified as the computing steps of the abstract machine. A computing *step* is the atomic unit of computation, defined as a set of parallel updates made to global *state*. A *state* is defined as the values of all variables at a specific step. A machine *executes* a step by yielding a set of state updates. A *run*, potentially infinite, is a sequence of steps.

The following subsection presents the basic concepts of ASM theory. For a complete description of the theory of abstract state machines, the reader is referred to [8]. Our proposed extensions to the base theory are explained in section 4.

3.1 Basic ASM Specification

The term *specification* is used to denote the complete document that results from the process of writing down a system design. This section introduces specifications that contain only a single abstract state machine, also known as *basic* or *single-agent* ASMs in the ASM community [8].

A basic abstract state machine specification is made up of two parts - an abstract state machine and an environment. The machine executes based on values in the environment and modifies values in the environment. The environment consists of two parts - the set of environment variables and the universe of types that variables can have. In the TASM language all variables are strongly typed. The machine consists of three parts - a set of monitored variables, a set of controlled variables, and a set of rules. The *monitored* variables are the variables in the environment that affect the machine execution. The *controlled* variables are the variables in the environment that the machine affects. The set of *rules* are named predicates, written in precondition-effect style, that express the state evolution logic.

Formally, a specification $ASMSPEC$ is a pair:

$$ASMSPEC = \langle E, ASM \rangle$$

Where:

- E is the environment, which is a pair:

  $$E = \langle EV, TU \rangle$$

  Where:

  - EV denotes the *Environment Variables*, a set of typed variables
  - TU is the *Type Universe*, a set of types that includes:
    - Reals: $RVU = \mathbb{R}$
    - Integers: $NVU = \{\ldots, -1, 0, 1, \ldots\}$
    - Boolean constants: $BVU = \{True, False\}$
    - User-defined types: $UDVU$

- ASM is the machine, which is a triple:

  $$ASM = \langle MV, CV, R \rangle$$

  Where:

  - MV is the set of *Monitored Variables* = $\{mv \mid mv \in EV$ and mv is read-only in $R\}$
  - CV is the set of *Controlled Variables* = $\{cv \mid cv \in EV$ and cv is read-write in $R\}$
  - R is the set of *Rules* = $\{(n, r) \mid n$ is a name and r is a rule of the form *if C then A* where C is an expression that evaluates to an element in BVU and A is an action$\}$

An action A is a sequence of one or more updates to environment variables, also called an *effect expression*, of the form $v := vu$ where $v \in CV$ and vu is an expression that evaluates to an element in the type of v.

Updates to environment variables are organized in *steps*, where each step corresponds to a *rule execution*. In the rest of this paper, the terms *step execution* and *rule execution* are used interchangeably. A rule is *enabled* if its guarding condition, C, evaluates to the boolean value $True$. The *update set* for the ith step, denoted U_i, is defined as the collection of all updates to controlled variables for the step. An update set U_i will contain 0 or more pairs *(cv, v)* of assignments of values to controlled variables.

A *run* of a basic ASM is defined by a sequence of update sets.

3.1.1 Light Switch Example Version 1

A small example is presented to illustrate some of the features of the TASM language. Here the example shows a basic ASM specification describing the logic for switching a light on or off based on whether a switch is up or down. The specification is divided into sections, identified by capital letters followed by a colon. Comments in the specification are preceded by the escape sequence "`//`".

```
ENVIRONMENT:

USER-DEFINED TYPES:
  light_status  := {ON, OFF}
  switch_status := {UP, DOWN}

VARIABLES:
  light_status  light   := OFF
  switch_status switch  := DOWN

------------------------------------------

MAIN MACHINE:

MONITORED VARIABLES:
  switch

CONTROLLED VARIABLES:
  light

RULES:

  R1: Turn On
    if light = OFF and switch = UP then
      light := ON

  R2: Turn Off
    if light = ON and switch = DOWN then
      light := OFF
```

A sample run with the initial environment (($light$, OFF), ($switch$, UP)) yields one update set:

$$U_1 = ((light, ON))$$

After the step has finished executing, the environment becomes: (($light$, ON), ($switch$, UP)).

4 The Timed Abstract State Machine Language

Here we describe the TASM language. We do this by introducing a series of modifications and extensions to the ASM formalism from Section 3.

4.1 Time

The TASM approach to time specification is to specify the duration of a rule execution. In the TASM world, this means that each step will last a finite amount of time before an update set is applied to the environment. Syntactically, time gets specified for each rule in the form of an annotation. The specification of time can take the form of a single value t, or can be specified as an interval [t_{min}, t_{max}]. The lack of a time annotation for a rule is assumed to mean t = 0. Semantically, a time annotation is interpreted as a value $\in \mathbb{R}$. If a time annotation is specified as an interval, the rule execution will last an amount t_i where t_i is taken randomly from the interval, which is interpreted as a closed interval on $\mathbb{R}$. The approach uses relative time between steps since each step will have a finite duration. The total time for a run of a given machine is simply the summation of the individual step times over the run. The time extensions are formally detailed in the following section, and the example from the previous section is extended to include time annotations.

4.2 Resources

The specification of non-functional properties includes timing characteristics as well as resource consumption properties. A *resource* is defined as a global quantity that has a finite size. Power, memory, and communication bandwidth are examples of resources. Resources are used by the machine when the machine executes rules.

Because resources are global quantities, they are defined at the environment level. The environment E is extended to reflect the definition of resources:

$$E = \langle EV, TU, ER \rangle$$

Where:

- EV and TU remained unchanged from Section 3.1
- ER is the set of named resources:
  - ER = {(rn, rs) | rn is the resource name, and rs is the resource size, a value $\in \mathbb{R}$}

Similarly to time specification, syntactically, each rule specifies how much of a given resource it consumes. The specification of resource consumption takes the form of an annotation, where the resource usage is specified either as an interval or as a single value. The omission of a resource consumption annotation is assumed to mean zero resource consumption. The semantics of resource usage are assumed to be *volatile*, that is, usage lasts only through the step duration. For example, if a rule consumes 128 kiloBytes of memory, the total memory usage will be increased by 128 kiloBytes during the step duration and will be decreased by 128 kiloBytes after the update set has been applied to the environment.

Formally, a rule R of a machine ASM is extended to reflect time and resource annotations:

$$R = \langle n, t, RR, r \rangle$$

Where:

- n and r are defined in Section 3.1
- t denotes the duration of the rule execution and can be a single value $\in \mathbb{R}$ or a closed interval on $\mathbb{R}$
- RR is the set of resources used by the rule where each element is of the form (rr, ra) where $rr \in ER$ is the resource name and ra is the resource amount consumed, specified either as a single value $\in \mathbb{R}$ or as a closed interval on $\mathbb{R}$

When a machine executes a step, the update set that is produced will contain the duration of the step, as well as the amounts of resources that were consumed during the step execution. We use the special symbol $\perp$ to denote the absence of an annotation, for either a time annotation or a resource annotation. The role of the $\perp$ symbol will become important in Section 4.3 and Section 4.4. Update sets are extended to include the duration of the step, $t \in \mathbb{R} \cup \{\perp\}$ and a set of resource usage pairs rc = (rr, rac) $\in RC$ where rr is the resource name and $rac \in \mathbb{R} \cup \{\perp\}$ is a single value denoting the amount of resource usage for the step. If a resource is specified as an interval, rac is a value randomly selected from the interval.

The symbol TRU_i is used to denote the timed update set, with resource usages, of the ith step of a machine, where t_i is the step duration, RC_i is the set of consumed resources, and U_i is the set of updates to variables from section 3.1:

$$TRU_i = (t_i, RC_i, U_i)$$

For the remainder of this paper, the term *update set* refers to an update set of the TRU_i form.

4.2.1 Light Switch Example Version 2

The light switch example from the previous section is extended with time annotations and resource annotations. The sample resource is memory. For brevity, only the modified rules of the main machine are shown. The remainders of the specification are the same as in Version 1.

```
R1: Turn On
  t      := [4, 10]
  memory := 512
  if light = OFF and switch = UP then
    light := ON

R2: Turn Off
```

```
t      := 6
memory := [128, 256]
if light = ON and switch = DOWN then
  light := OFF
```

A sample run with the initial environment ((light, OFF), (switch, UP)) yields the following update set:

TRU_1 = (5, ((memory, 512)), ((light, ON)))

The duration of 5 time units was randomly selected from the interval [4, 10].

4.3 Hierarchical Composition

The examples given so far have dealt only with a single sequential ASM. However, for more complex systems, structuring mechanisms are required to partition large specifications into smaller ones. The partitioning enables bottom-up or top-down construction of specifications and creates opportunities for reuse. The composition mechanisms included in the language are based on the XASM language [1]. In the XASM language, an ASM can use other ASMs in rule effects in two different ways - as a sub ASM or as a $function$ ASM. A sub ASM is a machine that is used to structure specifications. A $function$ ASM is a machine that takes a set of inputs and returns a single value as output, similarly to a function in programming languages. These two concepts enable abstraction of specifications by hiding details inside of auxiliary machines.

The definition of a sub ASM is similar to the previous definition of machine ASM:

$$SASM = \langle n, MV, CV, R \rangle$$

Where n is the machine name, unique in the specification, and other tuple members have the same definition as mentioned in previous sections. The execution and termination semantics of a sub ASM are different than those of a main ASM. When a sub ASM is invoked, one of its enabled rules is selected, it yields an update set, and it terminates.

The definition of a function ASM is slightly different. Instead of specifying monitored and controlled variables, a function ASM specifies the number and types of the inputs and the type of the output:

$$FASM = \langle n, IV, OV, R \rangle$$

Where:

- n is the machine name, unique in the specification
- IV is a set of named inputs (ivn, it) where ivn is the input name, unique in IV, and $it \in TU$ is its type.
- OV is a pair (ovn, ot) specifying the output where ovn is the name of the output and $ot \in TU$ is its type
- R is the set of rules with the same definition as previously stated, but with the restriction that it only operates on variables in IV and OV.

A function ASM cannot modify the environment and must derive its output solely from its inputs. The only side-effect of a function ASM is time and resource consumption.

A specification, $ASMSPEC$, is extended to include the auxiliary ASMs:

$$ASMSPEC = \langle E, AASM, ASM \rangle$$

Where:

- *E* is the environment
- *AASM* is a set of auxiliary ASMs (both sub ASMs and function ASMs)
- ASM is the main machine

Semantically, hierarchical composition is achieved through the composition of update sets. A rule execution can utilize sub machines and function machines in its effect expression. Each effect expression produces an update set, and those update sets are composed together to yield a cumulative update set to be applied to the environment. To define the semantics of hierarchical composition, we utilize the semantic domain $\mathbb{R} \cup \{\bot\}$ introduced in Section 4.2. The special value $\bot$ is used to denote the absence of an annotation, for either a time annotation or a resource annotation.

We define two composition operators, $\otimes$ and $\oplus$, to achieve hierarchical composition. The $\otimes$ operator is used to perform the composition of update sets produced by effect expressions within the same rule:

$$\begin{aligned} TRU_1 \otimes TRU_2 &= (t_1, RC_1, U_1) \otimes (t_2, RC_2, U_2) \\ &= (t_1 \otimes t_2, RC_1 \otimes RC_2, U_1 \cup U_2) \end{aligned}$$

The $\otimes$ operator is commutative and associative. The semantics of effect expressions within the same rule are that they happen in parallel. This means that the time annotations will be composed to reflect the duration of the longest update set:

$$t_1 \otimes t_2 = \begin{cases} t_1 & \text{if } t_2 = \bot \\ t_2 & \text{if } t_1 = \bot \\ max(t_1, t_2) & \text{otherwise} \end{cases}$$

The composition of resources also follows the semantics of parallel execution of effect expressions within the same rule. The $\otimes$ operator is distributed over the set of resources:

$$\begin{aligned} RC_1 \otimes RC_2 &= (rc_{11}, \ldots, rc_{1n}) \otimes (rc_{21}, \ldots, rc_{2n}) \\ &= (rc_{11} \otimes rc_{21}, \ldots, rc_{1n} \otimes rc_{2n}) \\ &= ((rr_{11}, rac_{11}) \otimes (rr_{21}, rac_{21}), \ldots, \\ &\quad (rr_{1n}, rac_{1n}) \otimes (rr_{2n}, rac_{2n})) \\ &= ((rr_{11}, rac_{11} \otimes rac_{21}), ... \\ &\quad ((rr_{1n}, rac_{1n} \otimes rac_{2n})) \end{aligned}$$

In the TASM language, resources are assumed to be *additive*, that is, parallel consumption of amounts r_1 and r_2 of the same resource yields a total consumption $r_1 + r_2$:

$$rac_1 \otimes rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \perp \\ rac_2 & \text{if } rac_1 = \perp \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

Intuitively, the cumulative duration of a rule effect will be the longest time of an individual effect, the resource consumption will be the summation of the consumptions from individual effects, and the cumulative updates to variables will be the union of the updates from individual effects.

The $\oplus$ operator is used to perform composition of update sets between a *parent* machine and a *child* machine. A parent machine is defined as a machine that uses an auxiliary machine in at least one of its rules' effect expression. A child machine is defined as an auxiliary machine that is being used by another machine. For composition that involves a hierarchy of multiple levels, a machine can play both the role of parent and the role of child. An example of multi-level composition is given at the end of this Section. To define the operator, we use the subscript p to denote the update set generated by the parent machine, and the subscript c to denote the update set generated by the child machine:

$$\begin{aligned} TRU_p \oplus TRU_c &= (t_p, RC_p, U_p) \oplus (t_c, RC_c, U_c) \\ &= (t_p \oplus t_c, RC_p \oplus RC_c, U_p \cup U_c) \end{aligned}$$

The $\oplus$ operator is *not* commutative, but it is associative. The duration of the rule execution will be determined by the parent, if a time annotation exists in the parent. Otherwise, it will be determined by the child:

$$t_p \oplus t_c = \begin{cases} t_c & \text{if } t_p = \perp \\ t_p & \text{otherwise} \end{cases}$$

The distribution of the $\oplus$ operator over the set of consumed resources is the same as for the $\otimes$ operator:

$$\begin{aligned} RC_p \oplus RC_c &= (rc_{p1}, \ldots, rc_{pn}) \oplus (rc_{c1}, \ldots, rc_{cn}) \\ &= (rc_{p1} \oplus rc_{c1}, \ldots, rc_{pn} \oplus rc_{cn}) \\ &= ((rr_{p1}, rac_{p1}) \oplus (rr_{c1}, rac_{c1}), \ldots, \\ &\quad (rr_{pn}, rac_{pn}) \oplus (rr_{cn}, rac_{cn})) \\ &= ((rr_{p1}, rac_{p1} \oplus rac_{c1}), \ldots \\ &\quad ((rr_{pn}, rac_{pn} \oplus rac_{cn})) \end{aligned}$$

The resources consumed by the rule execution will be determined by the parent, if a resource annotation exists in the parent. Otherwise, it will be determined by the child:

$$rac_p \oplus rac_c = \begin{cases} rac_c & \text{if } rac_p = \perp \\ rac_p & \text{otherwise} \end{cases}$$

Intuitively, the composition between parent update sets and child update sets is such that the parent machine overrides the child machine. If the parent machine has annotations, those annotations override the annotations from child machines. If a parent machine doesn't have an annotation, then its behavior is defined by the annotations of the auxiliary machines it uses.

Figure 1 shows a hierarchy of machines for an sample rule execution. Each numbered square represents a machine. Machine "1" represents the rule of the main machine being executed; all other squares represent either sub machines or function machines used to derive the update set returned by the main machine. Machine "3" is an example of a machine that plays the role of parent (of machine "7") and child (of machine "1").

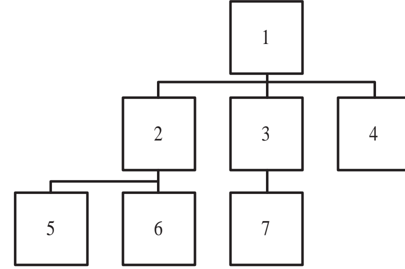


Figure 1. Hierarchical composition

Each machine generates an update set TRU_i, where i is the machine number. The derivation of the returned update set is done in a bottom-up fashion, where TRU_{ret} is the update set returned by the main machine:

$$\begin{aligned} TRU_{ret} = TRU_1 \oplus (\quad & (TRU_2 \oplus (TRU_5 \otimes TRU_6)) \otimes \\ & (TRU_3 \oplus TRU_7) \otimes \\ & TRU_4) \end{aligned}$$

4.3.1 Light Switch Example Version 3

The example from the previous sections is extended to illustrate the use of auxiliary ASMs. The example has been extended with a function ASM and a sub ASM.

```
FUNCTION MACHINE:          |   SUB MACHINE:
  TURN_ON                  |    TURN_OFF
                           |
INPUT VARIABLES:           |   MONITORED VARIABLES:
  switch_status ss         |     switch
                           |
OUTPUT VARIABLE:           |   CONTROLLED VARIABLES:
  light_status ls          |     light
                           |
RULES:                     |   RULES:
                           |
  R1: Turn On              |     R1: Turn Off
    t      := [4, 10]      |       t := 6
    memory := 128          |
    if ss = UP then        |       if switch = DOWN then
      ls := ON             |         light := OFF
                           |
  R2: Else                 |     R2: Else
    else then              |       else then
      ls := OFF            |         skip
```

The two modified rules of the main machine are shown below:

```
  R1: Turn On
    t := 1
    if light = OFF and switch = UP then
```

```
      light := TURN_ON(switch) //uses fASM

  R2: Turn Off
    memory := 1024
    if light = ON and switch = DOWN then
      TURN_OFF() //uses sASM
```

The first step of two sample runs are shown below:

- Initial environment: ((light, OFF), (switch, UP))
  Update set: (1, ((memory, 128)), ((light, ON)))
- Initial environment: ((light, ON), (switch, DOWN))
  Update set: (6, ((memory, 1024)), ((light, OFF)))

The first sample run invokes the function ASM and obtains the step duration from the main ASM definition and the resource consumption from the function ASM. The second sample run obtains the variable updates and rule duration from the sub ASM and the resource consumption from the main ASM.

4.4 Parallel Composition

To enable specification of multiple parallel activities in a system, the TASM language allows parallel composition of multiple abstract state machines. Parallel composition is enabled through the definition of multiple top-level machines, called *main* machines. Formally, the specification $ASMSPEC$ is extended to include a set of main machines $MASM$ as opposed to the single main machine ASM for basic ASM specifications:

$$ASMSPEC = \langle E, AASM, MASM \rangle$$

Where:

- *E* is the environment
- *AASM* is a set of auxiliary ASMs (both sub ASMs and function ASMs)
- $MASM$ is a set of main machines ASM that execute in parallel

The definition of a main machine ASM is the same as from previous sections. Other definitions also remain unchanged.

The semantics of parallel composition regards the synchronization of the main machines with respect to the global progression of time. We define *tb*, the global time of a run, as a monotonically increasing function over $\mathbb{R}$. Machines execute steps that last a finite amount of time, expressed through the duration t_i of the produced update set. The *time of generation*, tg_i, of an update set is the value of tb when the update set is generated. The *time of application*, ta_i, of an update set for a given machine is defined as $tg_i + t_i$, that is, the value of tb when the update set will be applied. A machine whose update set, generated at global time tg_p, lasts t_p will be *busy* until $tb = tg_p + t_p$. While it is busy, the machine cannot perform other steps. In the meantime, other machines who are not busy are free to perform steps. This informal definition gives rise to update sets no longer constrained by step number, but constrained by time. Parallel composition, combined with time annotations, enables the specification of both synchronous and asynchronous systems.

We define the operator $\odot$ for parallel composition of update sets. For a set of update sets TRU_i generated during the same step by i different main machines:

$$\begin{aligned} TRU_1 \odot TRU_2 &= (t_1, RC_1, U_1) \odot (t_2, RC_2, U_2) \\ &= \begin{cases} (t_1, RC_1 \odot RC_2, U_1) & \text{if } t_1 < t_2 \\ (t_2, RC_1 \odot RC_2, U_2) & \text{if } t_1 > t_2 \\ (t_1, RC_1 \odot RC_2, U_1 \cup U_2) & \text{if } t_1 = t_2 \end{cases} \end{aligned}$$

The operator $\odot$ is both commutative and associative. The distribution of the $\odot$ operator over the set of resource consumptions is the same as for the $\otimes$ and $\oplus$ operators:

$$\begin{aligned} RC_1 \odot RC_2 &= (rc_{11}, \ldots, rc_{1n}) \odot (rc_{21}, \ldots, rc_{2n}) \\ &= (rc_{11} \odot rc_{21}, \ldots, rc_{1n} \odot rc_{2n}) \\ &= ((rr_{11}, rac_{11}) \odot (rr_{21}, rac_{21}), \ldots, \\ &\quad (rr_{1n}, rac_{1n}) \odot (rr_{2n}, rac_{2n})) \\ &= ((rr_{11}, rac_{11} \odot rac_{21}), \ldots \\ &\quad ((rr_{1n}, rac_{1n} \odot rac_{2n})) \end{aligned}$$

The parallel composition of resources is assumed to be additive, as in the case of hierarchical composition using the $\otimes$ operator:

$$rac_1 \odot rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \bot \\ rac_2 & \text{if } rac_1 = \bot \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

At each global step of the simulation, a list of pending update sets are kept in an ordered list, sorted by time of application. At each global step of the simulation, the update set at the front of the list is composed in parallel with other update sets, using the $\odot$ operator and the resulting update set is applied to the environment. Once an update set is applied to the environment, the step is completed and the global time of the simulation progresses according to the duration of the applied update set.

To enable communication between different machines, the TASM language provides synchronization channels, in the style of the Calculus of Communication Systems (CCS) [20]. A *synchronization channel* is defined as a global object, uniquely identified by its name, that is used by two machines to synchronize. When using a communication channel, one machine plays the role of *sender* and the other machine plays the role of *receiver*. The syntax for using a communication channel is based on CCS. For an example communication channel named x, a sender would use $x!$ to send a notification and a receiver would use $x?$ to receive a notification. For more details on the features of the TASM language, the reader is referred to [22].

Component	Action	Duration	Power
Feed	Move block	5	500
Deposit	Move block	5	500
Robot	Rotate 30°	1	1000
Robot	Drop a block	1	1000
Robot	Pickup a block	1	1000
Press	Stamp a block	11	1500

Table 1. Durative actions

5 Production Cell Example

This section illustrates the features of the TASM language through the modeling of a more substantial example, the production cell case study [19]. The production cell consists of a series of components that need to be coordinated to achieve a common goal. The purpose of the production cell system is to stamp blocks. Blocks come into the system from the *loader*, which puts the block on the *feed belt*. Once the block reaches the end of the feed belt, the *robot* can pick up the block and insert it into the *press*, where the block is stamped. Once the block has been stamped, the robot can pick up the block from the press and unload it on the *deposit belt*, at which point the stamped block is carried out of the system. The schematic view of the production cell system is shown in Figure 2.

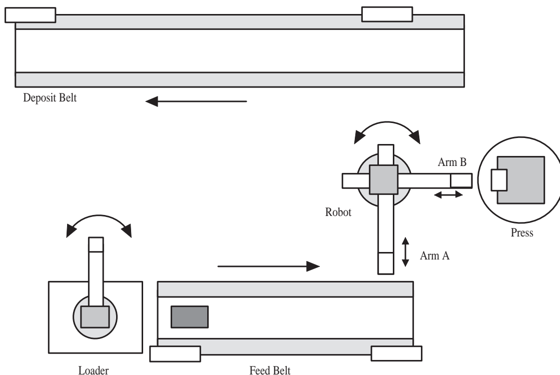


Figure 2. Top view of the production cell

All components operate concurrently and must be synchronized to achieve the system's goal. The robot has two arms, arm A and arm B, which move in tandem and can pick up and drop blocks in parallel. For example, the robot can drop a block in the press while picking up a block from the feed. A *controller* coordinates the actions of the system by sending commands to the robot.

Some simplifications and extensions have been made to the original problem definition [19]. For example, the elevating rotatory table has been omitted. The original example has been extended to reflect the reality that certain actions are *durative*, that is, they last a finite amount of time. The example has also been extended to include a resource, *power consumption*. The list of durative actions, with their power consumptions, are shown in Table 1.

All other actions are assumed to be instantaneous and are assumed to consume no power. In the TASM model, each component of the production cell is modeled as a main machine. The model also contains a main machine for the controller. Sub machines and function machines are used, mostly to structure the actions of the robot. The robot waits for a command from the controller and then executes that command before waiting for another command. Listing 1 shows the first rule of the submachine that updates the robot's position. Listing 2 shows the second rule of the main machine Feed, which carries a block from the loader to the robot. The time annotation specifies the amount of time that it takes for the block to travel from the loader to the robot.

Listing 1 Rule 1 of submachine UPDATE_POSITION

```
R1: Rotate CW
{
    t       := 1;
    power   := 1000;

    if command = rotatecw then
      robot_angle := rotateClockwise();
      armapos := armPosition(ARM_A_FEED_ANGLE,
                             ARM_A_DEPOSIT_ANGLE,
                             ARM_A_PRESS_ANGLE,
                             rotateClockwise());
      armbpos := armPosition(ARM_B_FEED_ANGLE,
                             ARM_B_DEPOSIT_ANGLE,
                             ARM_B_PRESS_ANGLE,
                             rotateClockwise());
}
```

Due to lack of space, other rules and other component specifications are omitted; the complete list of machines is shown in Table 2.

Listing 2 Rule 2 of main machine Feed

```
R2: Block goes to end of belt
{
  t     := 5;
  power := 500;

  if feed_belt = loaded then
    feed_block := available;
    feed_sensor!;
}
```

We use these two listings to illustrate the semantics of parallel composition. For a partial initial environment (($feed_belt$, $loaded$), ($command$, $rotatecw$), ($robot_angle$, 30), ($armapos$, $intransit$), ($armbpos$, $intransit$)) with the two main machines Feed and Robot with associated auxiliary machines, two update sets are generated by the main machines:

$$TRU_{Feed,1} = (5, ((\text{power}, 500)), ((feed_block, available))$$
$$TRU_{Robot,1} = (1, ((\text{power}, 1000)), ((robot_angle, 0.0), (armapos, atfeed), (armbpos, atpress)))$$

The parallel composition of these two update sets yields the environment described below. For brevity, only the environment variables whose values are changed are shown. The time t denotes the global simulation time:

- $t < 1$: (((power, 1500)), $\emptyset$)
- $t = 1$: (((power, 1500)), (($feed_block$, $available$)))

Name	Type	Purpose
Controller	Main	Sends commands to the robot
Loader	Main	Loads blocks onto the feed belt
Feed	Main	Carries blocks from the loader
Robot	Main	Processes commands
Press	Main	Stamps blocks
Deposit	Main	Carries blocks out of the system
allEmpty	Function	Determines whether the robot is loaded
armPosition	Function	Returns the position of an arm
rotateClockwise	Function	Changes the robot angle by +30°
rotateCounterClockwise	Function	Changes the robot angle by -30°
DROP_ARM_A	Sub	Drop a block from arm A
DROP_ARM_B	Sub	Drop a block from arm B
DROP_BLOCKS	Sub	Invokes the drop submachines
EXECUTE_COMMAND	Sub	Executes a command
PICK_UP_ARM_A	Sub	Picks up a block with arm A
PICK_UP_ARM_B	Sub	Picks up a block with arm B
PICK_UP_BLOCKS	Sub	Invokes the pick up submachines
UPDATE_POSITION	Sub	Updates the angle of the robot

Table 2. List of all machines used in the production cell model

- 1 < t < 5: (((power, 1000)), $\emptyset$)
- t = 5: (((power, 1000)), (($robot_angle$, 0.0), ($armapos$, $atfeed$), ($armbpos$, $atpress$))
- t > 5: (((power, 0)), $\emptyset$)

The complete model, including all the components, was run for a scenario where the loader fed a total of 5 blocks into the system, with the initial state shown in Figure 2. It took a total of 219 simulation steps and 103 time units for all 5 blocks to go through the system, given the controller strategy.

6 Conclusion and Future Work

The contributions of this paper span two different areas. On the theoretical side, the paper presents extensions to the ASM formalism to facilitate the specification of real-time systems. The specification includes both functional and non-functional properties, integrated into a unified formalism. The incorporation of timing, resource, and functional behavior into a single language fills an important need of the real-time system community [9, 17]. This is achieved both for basic TASM specifications and for the composition of specifications. The resulting formalism keeps the same theoretical foundations of ASM theory but is better suited for modeling real-time systems because of the support to explicitly state resource consumption and timing behavior. On the practical side, this paper defines a formalism that has the potential of being both formal and usable. By basing the formalism on the theory of abstract state machines, the purpose is to bring the stated benefits of using abstract state machines to the designers of reactive real-time systems.

On the theoretical side, verification techniques and test-case generation techniques are currently being surveyed and studied to understand how these capabilities could be applied to the proposed language. Preliminary results suggest leveraging existing verification tools (e.g., UPPAAL [18]) by defining semantic preserving formal mappings between the TASM language and the formalisms of existing verification tools [25]. On the practical side, the TASM language is the first step towards a framework for validation and verification of high-integrity embedded systems [24]. The language will serve as the basis for the framework and an associated toolset to write and analyze real-time system specifications is being developed [23]. The TASM language will be incorporated into a suite of tools that will be used to verify timing and resource consumption behavior of embedded real-time systems [26].

References

[1] M. Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In *Abstract State Machines - ASM 2000, International Workshop on Abstract State Machines*. TIK-Report 87, 2000.

[2] J. Bergstra, A. Ponse, and S. Smolka. *A Handbook of Process Algebra*. North-Holland, 2001.

[3] M. Bernardo and F. Corradini. *Formal Methods for the Design of Real-Time Systems*. Springer-Verlag, 2004.

[4] G. Berry. The Essence of ESTEREL. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[6] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM '95*, volume 1012 of *LNCS*. Springer-Verlag, 1995.

[7] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In *Specification and Validation Methods*. Oxford University Press, 1995.

[8] E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[9] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Springer, 2005.

[10] J. P. Bowen and M. G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computers*, 28(4), April 1994.

[11] A. Cerone and A. Maggiolo-Schettini. Time-based Expressivity of Time Petri Nets for System Specification. In *Theoretical Computer Science*, volume 216. Springer-Verlag, 1999.

[12] A. K. Cheng. *Real-Time Systems: Schedulability, Analysis, and Verification*. John Wiley and Sons, 2003.

[13] E. M. Clarke and J. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(3), 1996.

[14] J. Cohen and A. Slissenko. On Verification of Refinements of Asynchronous Timed Distributed Algorithms. In *International Workshop on Abstract State Machines*. Springer-Verlag, 2000.

[15] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering*

and Methodology, 5(4), 1996.

[16] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In *Real-Time: Theory in Practice, REX Workshop*, pages 226–251, 1991.

[17] A. Jantsch and I. Sander. Models of Computation and Languages for Embedded System Design. *IEE Proceedings - Computers and Digital Techniques*, 152(2), March 2005.

[18] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[19] C. Lewerentz and T. Lindner. Production Cell: A Comparative Study in Formal Specification and Verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, 1995.

[20] R. Milner. *Communication and Concurrency*. Prentice Hall, 1980.

[21] Object Management Group, Inc. Unified Modeling Language: Superstrucure. Version 2.0. OMG Specification, August 2005.

[22] M. Ouimet. The TASM Language Reference Manual, Version 1.1. Available from http://esl.mit.edu/tasm, November 2006.

[23] M. Ouimet, G. Berteau, and K. Lundqvist. Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. In *Proceedings of the Satellite Events of the 2006 MoDELS Conference*, LNCS, October 2006.

[24] M. Ouimet and K. Lundqvist. The Hi-Five Framework and the Timed Abstract State Machine Language. In *Proceedings of the 27th IEEE Real-Time Systems Symposium - Work in Progress Session*, December 2006.

[25] M. Ouimet and K. Lundqvist. Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. In *Proceedings of the 3rd International Workshop on Model-Based Testing (MBT '07), Satellite Workshop of ETAPS '07*, April 2007.

[26] M. Ouimet and K. Lundqvist. Verifying Execution Time using the TASM Toolset and UPPAAL, January 2007. Technical Report ESL-TIK-000212, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[27] H. Rust. Using Abstract State Machines: Using the Hypperreals for Describing Continuous Changes in a Discrete Notation. In *Abstract State Machines – ASM 2000*. Springer-Verlag, March 2000.

[28] SAE Aerospace. *Architecture Analysis & Design Language Standard*. SAE Publication AS506, 2004.

[29] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computers*, 23(9), 1990.

Extended Real-Time LOTOS for Preemptive Systems Verification

Tarek Sadani[(1)(2)], P. de Saqui-Sannes[(1)(2)], J-P. Courtiat[(1)]
[1]LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse Cedex 04, France
[2]ENSICA, 1 place Emile Blouin, 31056 Toulouse Cedex 05, France
tsadani@ensica.fr; desaqui@ensica.fr; courtiat@laas.fr
Fax: +33 5 61 61 86 88

Abstract

Real-time systems not only interact with their environment and hopefully deliver their expected outputs on time. Unlike transformational systems, they may be interrupted at any time while keeping the capacity to restart later on without loosing their state information. Therefore, a real-time system specification language should include a suspend/resume capability. In this paper, we propose to extend the timed process algebra RT-LOTOS with a suspend/resume operator. Extended RT-LOTOS specifications are translated to Stopwatch Time Petri nets that may be analyzed using the TINA tool. We define an RT-LOTOS to SwTPN translation pattern. A formal proof is included. Case studies show the interest of our proposal for preemptive systems specification and verification.

1 Introduction

A wealth of formal models have been proposed in the literature to describe and analyze real-time systems. Few of them enable explicit description of suspend/resume operations. Examples include Stopwatch Time Petri nets[6] and Stopwatch automata[11]. As a timed process algebra, RT-LOTOS[13] also makes it possible to describe important features of real-time systems (e.g., parallelism, reaction to stimuli from the environment, delay, temporal indeterminism). RT-LOTOS supports a *disrupt* operator which allows a process Q to suspend another process P for ever. Suspension in RT-LOTOS is hence reduced to unrecoverable abortion. Clearly, RT-LOTOS misses resume capabilities. This weakness is inherited from (untimed) LOTOS. In [15] it is shown that LOTOS misses some mechanisms to deal with suspend/resume behaviors. This paper's contribution is to extend RT-LOTOS with a *suspend/resume* operator. The proposed extension is given a formal semantics without disturbing the semantic model of RT-LOTOS. It is worth to be noticed that discussion goes beyond language aspects. The challenge is to have extended RT-LOTOS specifications effectively model checked. We propose to translate RT-LOTOS specifications into Stopwatch Time Petri nets (SwTPN)[6]. The latter are rigorously analyzed using TINA[8], the Time Petri net analyzer developed by LAAS-CNRS.
The paper is organized as follows. Section 2 presents RT-LOTOS. Section 3 explains the expected benefits of adding a suspend/resume operator to RT-LOTOS. Section 4 presents SwTPN. Section 5 discusses RT-LOTOS to SwTPN translation, it includes a formal proof. Section 6 presents three examples. Section 7 surveys related work. Section 8 concludes the paper and outlines future work.

2 RT-LOTOS

Real-Time LOTOS, or RT-LOTOS for short, is a timed extension of the ISO-based formal description technique LOTOS (Language of Temporal Ordering of Specification)[20]. LOTOS relies on the CCS process algebra and inherits a multiple rendez-vous mechanism from Hoare's CSP. RT-LOTOS enables explicit and semantically well-founded description of temporal mechanisms. Three generic temporal operators have been added to LOTOS: First, a deterministic delay expressed by the *delay* operator. For instance, *delay(d)* makes it possible to delay a process P for a certain amount of time d. Second, a non-deterministic delay expressed by the *latency* operator. For instance, *latency(l)* makes it possible to delay a process for a value that is non-deterministically selected in [0, l]. Its usefulness and efficiency have been demonstrated in [12]. The third temporal operator is a time-limited offer associated with an action. For instance, $g\{t\}$ allows one to limit the amount of time allocated to offer an action a.
The following processes P and PL illustrate the use of the three temporal operators of RT-LOTOS.

```
process P[a]: exit:=      process PL[a]: exit:=
 delay(2)a{5}; exit       delay(2)latency(6)a{5};exit
endproc                   endproc
```

Process P starts with a 2 time units delay. Once the delay expires, action a is offered to the environment during 5 time units. If the process's environment does not synchronize on a before this deadline, a time violation occurs and the process transforms into *stop*. Process PL differs from P, for it contains a *latency* operator. Action a is delayed by a minimum delay of 2 units of time and a maximum

delay of 8 units of time (in case the *latency* goes to its maximum value). From the environment's point of view, if the latency lasts l time units, the process behaves like *delay(2+l)a{5-l}* (cf. the left part of Figure 1). Of course, if the duration of the latency goes beyond 5 units of time, a temporal violation occurs and process *PL* transforms into *stop*(cf. the right part of Figure 1).

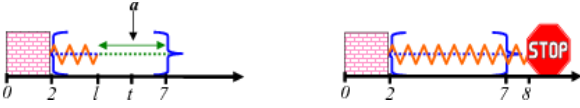


Figure 1. Delay, latency and Limited offering

The originality and interest of the latency operator is more obvious when one combines that operator with the *hiding* operator. In LOTOS, hiding allows one to transform an external *observable* action into an *internal* one. In RT-LOTOS, hiding has the form of a renaming operator which renames action *a* into *i(a)*. In most timed extensions of LOTOS, hiding implies urgency. It thus removes any time indeterminism inherent to the limited time offering. In RT-LOTOS, a hidden action is urgent *as soon as it is no longer delayed by some latency operator*. Let us, e.g., consider the RT-LOTOS behavior `hide a in PL` where action *a* is hidden in process *PL*. If l is the duration of the latency, *i(a)* will *necessarily* occur at date $2 + l$, if $l < 5$. (cf. Figure 2). But, if ($l > 5$), a temporal violation occurs (similarly to the situation where action *a* was an observable action).

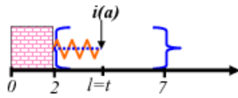


Figure 2. Hiding in RT-LOTOS

3 An RT-LOTOS Suspend/Resume operator

3.1 Rationale

Is RT-LOTOS well-suited to specify that a process execution may be stopped and resumed later on? To answer that question, let us consider the behavior of a simplified washing machine. It is made up of two processes named Machine and Cover, respectively.

```
process Machine[start,b_wash,e_wash]:exit:=
start; b_wash; e_wash; exit
endproc
process Cover[open, close]: exit:=
open; close; Cover[open,close]
endproc
```

Machine may be suspended by Cover at any time during its execution, and resumed after the completion of each Cover instance (each instance of Cover yields an *'open; close'* action sequence).
The RT-LOTOS disrupt [> operator is not appropriate to model that kind of behavior. Machine[...][> Cover[...] allows Machine to be suspended, but not to be resumed. An infinite sequence of 'open; close' actions will follow Machine's interruption.
The use of the parallel composition operator ||| would not be more appropriate, since the resulting interleaving of actions in Machine[...] ||| Cover[...], does not ensure that Cover will terminate before Machine is resumed[1].

Another solution is to adopt a state-oriented style. Using the choice [] and the enabling ≫ operators, we explicitly define all the possible suspending points in Machine (actions are indivisible). For this sake the behavior of Machine is changed as follows:

```
Process Machine2[start,b_wash,e_wash,open,close]
:exit
 ((start; exit)
 [](Cover[open, close] >> (start; exit)))
 >>((b_wash; exit)
 [](Cover[open, close] >> (b_wash; exit)))
 >>((e_wash; exit)
 [](Cover[open, close] >> (e_wash; exit)))
endproc
```

The definition of Cover has also to be changed to a non recursive one. This is to avoid an infinite sequence of Cover's actions and to ensure Machine actually resumes. However this 'contortion' lowers the readability and compositionality of the specification (although Machine is a simple sequence of actions) and allows only three times the 'open; close' sequence.
Let us now consider a timed extension of the washing machine.

```
Hide b_wash, e_wash in
process Machine[start,b_wash, e_wash]: exit:=
start;delay(1,2)b_wash;delay(40,70)e_wash;exit
endproc
```

After the occurrence of start, b_wash is delayed by a 1 up to 2 units delay. Washing takes between 40 and 70 units of time. Actions b_wash and e_wash are internalized and therefore urgent. In this revisited specification, it is impossible to define all the suspension points because we are considering a dense time model. Moreover using a specification technique based on the modification of the suspended behavior (Machine2) is no longer possible; it does not preserve the timing constraints in the original Machine process. Let us consider the following behavior expressed in state-oriented fashion:

```
(delay(1,2)b_wash; exit)
[](Cover[open,close]>>delay(1,2)b_wash; exit)
```

The choice offered between the two alternatives is resolved in the interaction with the environment. Cover is executed if the environment offers $open$, unless b_wash is executed (after a delay between 1 and 2 units of time). Until the choice is resolved the two alternatives age similarly. Let us now suppose the environment offers action $open$ after 1 unit of time, which leads to the following execution:

(delay(1,2)b_wash...) [] (Cover[open, close]≫ ...) $\xrightarrow{1}$
(delay(0,1)b_wash...) [] (Cover[open, close]≫ ...)$\xrightarrow{Cover[...]}$
(delay(1,2)b_wash; exit)

[1] ||| may be seen as a non deterministic solution to model coroutines.

After the elapsing of 1 unit of time, b_wash must be enabled at most after 1 time unit. In the resulting behavior, after the completion of process Cover, b_wash has to wait for a delay between 1 and 2 units of time, whereas 1 unit of time of this delay has already elapsed (the timing context is not restored).

3.2 Syntax and Semantics of the Suspend/Resume operator

Section 3.1 pointed out that RT-LOTOS lacks a mechanism for a modular description, of a process *P* whose temporal evolution can be suspended and then resumed at the same point.

To remedy to this situation we propose to add a $suspend/resume$ operator to RT-LOTOS. It is represented by [g$\gg$ (g being the gate used to resume), which is a mix between the disrupt [$>$ and the enabling $\gg$ operators (we extend [$>$ to handle resumption). *P*[g$\gg$*Q* models the possible suspension of main behavior *P* by *Q*, *Q* is executed till it terminates or executes action g, in the last case, the control returns to process *P*, in the same point it has been suspended. This special gate g is not fixed syntactically in the operator, and the user is free to use any gate name.

Special attention is provided to give the proposed extension a simple semantics, which further suits abstract reasoning. We follow on the recommendation made in [5] on providing preemption primitives at first-class level and with full orthogonality to other concurrency and communication primitives. The behavior (P[g$\gg$Q) is formally defined using Plotkin-style structural semantics (SOS rules).

We introduce the following semantic operator $\ll$g]. The latter expresses that *P* has been suspended by *Q*. This operator appears only at the semantic level. It may not be used in RT-LOTOS specifications.
In the following G_P denotes the set of observable and hidden gates of *P*.

1) $\dfrac{P \xrightarrow{a} P'}{P[g \gg Q \xrightarrow{a} P'[g \gg Q} \quad a \in G_P \backslash \{\text{exit}\}$

2) $\dfrac{P \xrightarrow{exit} P'}{P[g \gg Q \xrightarrow{exit} P'}$

3) $\dfrac{Q \xrightarrow{a} Q'}{P[g \gg Q \xrightarrow{a} P \ll g]Q'}$

4) $\dfrac{Q \xrightarrow{a} Q'}{P \ll g]Q \xrightarrow{a} P \ll g]Q'} \quad a \in G_Q \backslash \{g\}$

5) $\dfrac{Q \xrightarrow{t} Q'}{P \ll g]Q \xrightarrow{t} P \ll g]Q'}$

6) $\dfrac{Q \xrightarrow{g} Q'}{P \ll g]Q \xrightarrow{g} P[g \gg Q'}$

7) $\dfrac{P \xrightarrow{t} P', Q \xrightarrow{t} Q'}{P[g \gg Q \xrightarrow{t} P'[g \gg Q'}$

Rule 1 defines the normal execution of P. It says that after any action a of P, Q is still given the chance to suspend P (Q is still active).
Rule 2 says that if P has successfully completed its execution, then Q can no longer be executed.
Rule 3 defines control passing between P and Q at the occurrence of one of the first actions of Q.
Rule 4 and rule 5 say that Q is the only active behavior. The suspended behavior P can not perform any action (rule 4). Further, it cannot age (rule 5).
Rule 6 permits the resuming of P. As soon as Q executes action g, P is resumed and Q is restarted. Instead of writing $P \ll g]\ Q$, we could have written $P \ll g](Q, Q^0)$, where the operand Q^0 is used to keep the initial sate of Q. We then write $P \ll g](Q, Q^0) \xrightarrow{g} P[g \gg (Q^0, Q^0)$.
Rule 7 says that regarding time,[g$\gg$ behaves as [$>$ (the aging of P induces the aging Q).

One could think that we are missing some rule, to define the behavior of $P \ll g]Q$ when Q terminates successfully, in order to get rid of the suspended behavior P, but the termination of Q is supported by rule 4 where the suspended behavior is kept (this is not a problem, since Q cannot do any action after its termination, therefore P cannot be resumed). Moreover, if this termination is captured by an enabling, all the whole behavior ($P \ll g]Q$) is forgotten.

4 Time Petri Nets with Stopwatches

Interruptions and suspend/resume operations are common mechanisms in real-time systems. Several modeling formalisms allowing description of behaviors which can be suspended and resumed with a memory of their status have been proposed in the literature. [11] proposes a subclass of Linear Hybrid Automata (LHA) :Stopwatch Automata (SWA). In SWA the derivative of a variable in a location is either 0 or 1, to account for a computation progress and suspension. [11] proves that SWA with hidden delays are as expressive as LHA. The reachability problem for this class of automata is undecidable[2]. As no expressive enough decidable subclass of SWA has been identified, [11] proposes an 'over-approximating' algorithm for model checking safety property of SWA. The approach is based on a DBM encoding of state classes. The authors admit that such an over-approximation is often too coarse.
Several extensions of Time Petri nets (TPN)[23] have been proposed for modeling action suspension and resumption[10] Scheduling-TPNs[25] where resources and priorities primitives are added to the original TPN model. In Inhibitor Hyperarc TPNs (IHTPN's)[26], special inhibitor arcs are introduced to control the progress of transitions. [6] proves that the state space reachability problem of these extensions is undecidable even for bounded nets.[2] Efficient state space over-approximation methods are available for all these extensions. As for SWA the overapproximation obtained is often too coarse.

In this paper, we consider a TPN model[23] extended with suspend/resume capabilities[6]. SwTPN's extend

[2] The state space reachability problem of bounded Petri nets and bounded Time Petri nets is known to be decidable.

TPN's with stopwatch arcs that control transitions progress.

Definition 1. A Stopwatch Time Petri net is a tuple $\langle P, T, Pre, Post, Sw, m_0, IS \rangle$ where $\langle P, T, Pre, Post, m_0, IS \rangle$ is a Time Petri net and $Sw : T \to P \to N$ is a function called the stopwatch incidence function. Sw associates an integer with each $(p, t) \in P \times T$. Values greater than 0 are represented by special arcs, called stopwatch arcs (possibly weighted). They are represented with diamond shaped arrows. Note that these arcs do not convey tokens. As usual, a transition t is enabled at marking m iff $m \geq pre(t)$. In addition, a transition enabled at m is active iff $m \geq Sw(t)$, otherwise it is said suspended. Figure 3 shows a Stopwatch Time Petri net. The arc from place p_0 to transition t_2 is a stopwatch arc (grey shadowed diamond) of weight 1. The

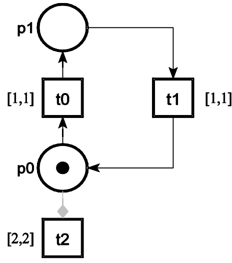


Figure 3. SwTPN example

firing of t_0 will freeze the timing evolution of t_2. t_2 will be fireable when its total enabling time reaches 2 time units. If we replace the stopwatch arc by a normal *pre* arc, t_2 will never be fired because of the continuous enabling condition.[3]

[6] proves that space reachability is undecidable for SwTPN's, even when bounded. In [6] the authors adapted the algorithm published in [7]. This construction yields exact state space abstraction, but as a consequence of the above undecidability result, boundedness of the SwTPN does not imply finiteness of the graph of state classes. To ensure termination on a class of bounded SwTPN's. [6] proposes a new overapproximation method based on quantization of the polyhedra representing temporal information in state classes. The exact behavior of the SwTPN can be approximated as closely as desired; both the exact and approximate computation methods have been implemented in an extension of the TINA tool [8]. However, the exact characterizations of the behavior of a SwTPN, obtained by the algorithm of [6] are finite in many practical cases, such as the experimentations reported in this paper.

[3]in the TPN model, a transition t with an associated static interval [a,b] can be fired if it has 'continuously' been enabled during at least 'a' time units, and it must fire if continuous enabling reaches 'b' time units, unless it conflicts with another transition.

5 Translating the Suspend/Resume RT-LOTOS operator to SwTPN

Two verification techniques have been developed for RT-LOTOS specifications. The first one is implemented by the *rtl* tool. The latter compiles an RT-LOTOS specification into a Timed automata and generates a minimal reachability graph. A more efficient technique was proposed in [28][27]. An RT-LOTOS specification is translated into a Time Petri net using the *rtl2tpn* tool, and verified using a Time Petri net analyzer.

Unfortunately neither Timed automata nor Time Petri net can represent clocks whose progression can be suspended and later resumed at the same point. Hence none of these models can be used as intermediate model for verifying the RT-LOTOS extension proposed in this paper. This is why we investigate a translation from RT-LOTOS to SwTPNs. This opens avenues to verification based on state space analysis methods proposed in [6]. The proposed translation method can be seen as providing RT-LOTOS terms with a SwTPN semantics. On this net semantics depends the quality of our translation. As defined in [24] a good net semantics should satisfy the 'retrievability' principle. Accordingly, we must not introduce any auxiliary transitions into the resulting SwTPN. By defining a one to one mapping of actions between RT-LOTOS and SwTPN we guarantee that a run of a system under design will necessarily lead to the same execution sequence in both the RT-LOTOS term and in its associated SwTPN. Moreover, the proposed RT-LOTOS *suspend/resume* operator is compositional. It is then inevitable, during the translation process, to consider SwTPNs as composable entities. Unfortunately, SwTPN miss such a structuring facilities. To fill this gap, we introduce the concept of *Sw-Component* as a basic building block.

5.1 Stopwatch Component

A *Sw-Component* encapsulates a labeled SwTPN which describes its behavior. It is handled through its interfaces and interactions points. A *Sw-Component* performs an action by firing a corresponding transition. A *Sw-Component* has two sets of labels. Act : the alphabet of the component. $Time$: a set of three labels, introduced to represent the intended temporal behavior of a component. A tv label represents a temporal violation in a time-limited offer. A $delay$ or $latency$ label represents a deterministic delay or a non deterministic delay, respectively.

A Sw-Component is graphically represented by a box containing a SwTPN (cf. Figure 4). The black-filled boxes at the component boundary represent interaction points. A token in the "out" place of a component means that the component has successfully completed its execution.

$C =< \Sigma, Act, Lab, I, O >$ is a *Sw-Component* where:

- $\Sigma =< P, T, Pre, Post, Sw, m_0, IS >$ is a SwTPN.
- $Act = A_o \cup A_h \cup \{exit\}$. A_o (observable actions) and

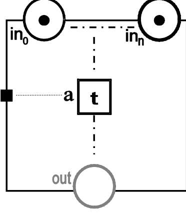


Figure 4. Sw-Component

A_h (hidden actions) are finite, disjoints sets of transitions labels. $A_o \cup \{exit\}$ represents the component's interaction points. During the translation process A_o and A_h will be used to model observable and hidden RT-LOTOS actions, respectively.

- $Lab : T \rightarrow (Act \cup Time)$ is a labelling function which labels each transition in Σ with an action name (Act) or with a time-event ($Time = \{tv, delay, latency\}$). Let T^{Act} (resp. T^{Time}) be the set of transitions whose label belongs to Act (resp. $Time$).
- $I \subset P$ is a non empty set of places defining the input interfaces of the component.
- $O \subset P$ is the output interface of the component. A component has an output interface if it has a transition(s) labelled by exit. If so, O is the outgoing place of those transitions. Otherwise, $O = \emptyset$.

Moreover, a set of invariants is associated with the *Sw-Components*:
[H1] The encapsulated SwTPN contains no source transition.
[H2] The encapsulated SwTPN is 1-bounded.
[H3] If all the 'input' places are marked, all other places are empty ($I \subset M \Rightarrow M = I$).
[H4] If the out place is marked, all other places are empty ($O \neq \emptyset \wedge O \subset M \Rightarrow M = O$).
H2 is called the "safe marking" property. This assumption is made by most analysis methods.

5.2 Translation Pattern for the Suspend/Resume operator

Let C_P be a *Sw-Component* associated with an RT-LOTOS behavior *P*. The set of first actions $\mathcal{FA}(C_P)$ is defined in Appendix A.
We denote as usual, by $p^\bullet = \{t \in T / post(t,p) \geq 1\}$ the set of outgoing transitions of p, and by $^\bullet p = \{t \in T / pre(t,p) \geq 1\}$ the set of incoming transitions of p.

In Figure 5, $C_{P[g \gg Q}$ is the component where C_P can be suspended by C_Q and then resumed at the same point. To model this behavior we introduce a shared place *SR*. It is connected with all transitions of C_P with stopwatches arcs of weight 1, except the $exit$ transition (if there is one).

$\forall t \in T_{C_P}\ lab(t) \neq exit \Rightarrow Sw(SR, t) = 1$. If the *SR* place is unmarked, the execution of C_P is suspended.

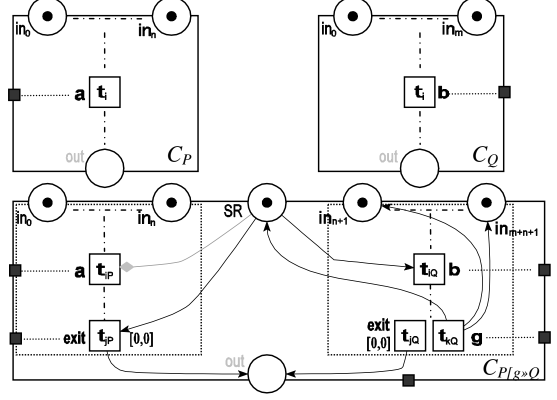


Figure 5. Suspend/Resume pattern

Moreover C_P is suspended at the occurrence of the first action of C_Q (SOS rule 3). *SR* is then an input place of the first action of C_Q.
After the successful termination of C_P, the latter can not be suspended anymore, and C_Q is deactivated (SOS rule 2). Hence *SR* has to be an input place for the 'exit' transition of C_P. $SR^\bullet = \mathcal{FA}(C_Q) \cup \bigcup_{t \in T_{C_P} \wedge lab(t) = exit}$.
After the execution of action *g* in C_Q, C_P is resumed (SOS rule 6). For this, action *g* restores the token in *SR*. The latter is an output place for the *g* transition of C_Q. $^\bullet SR = \{g\}$.
Finally, the output interfaces of the two components are merged. $O_{P[g \gg Q} = \{out\}\ if\ (O_P \neq \phi \vee O_Q \neq \phi)$. Let us notice that if the alphabet of the component C_Q does not include the special gate *g*, the operator behaves exactly as a disabling.

The definition of $C_{P[g \gg Q}$ follows:

$$C_{P[g \gg Q} = < \sum_{P[g \gg Q}, Act_P \cup Act_Q, \\ Lab_{P[g \gg Q}, I_P \cup I_Q \cup \{SR\}, O_{P[g \gg Q} >$$

$$where: \sum_{P[g \gg Q} = \langle P_{P[g \gg Q}, T_{P[g \gg Q}, Pre_{P[g \gg Q}, \\ Post_{P[g \gg Q}, Sw_{P[g \gg Q}, m_0, IS_{P[g \gg Q} >$$

$$P_{P[g \gg Q} = P_P \cup P_Q \cup \{SR\} \cup \{out\} \backslash O_Q \backslash O_P$$
$$T_{P[g \gg Q} = T_P \cup T_Q$$

$$Pre_{P[g \gg Q} = Pre_P \cup Pre_Q \cup \bigcup_{t \in \mathcal{FA}(C_Q)} (SR, t) \\ \cup \bigcup_{(t \in T_P) \wedge Lab(t) = exit} (SR, t)$$

$$Post_{P[g \gg Q} = Post_P \cup Post_Q \cup \\ \bigcup_{t \in T_Q \wedge Lab(t) = g} (t, SR)$$

$$Sw_{P[g\gg Q} = Sw_P \cup Sw_Q \cup \bigcup_{t \in T_P \wedge Lab(t) \neq' exit'} (SR, t)$$

$$IS_{P[g\gg Q} = IS_P \cup IS_Q$$

Moreover, a set of arcs is introduced to connect the places in I_Q with the *exit* transition of C_P ($\bigcup_{t \in T_{C_P} \wedge Lab(t)=' exit' \wedge p \in I_Q} (p,t)$). The aim of these arcs is to purge C_Q. A purge is the operation which consists in emptying the suspending component C_Q from a remaining tokens in its input interface after the successful termination of C_P. Reciprocally, if C_Q terminates successfully, C_P is purged. For readability reason we do not represent *purge* arcs in the above translation pattern.

5.3 Proof of translation consistency

We prove that the translation preserves the RT-LOTOS semantics of the new suspend/resume operator and that the associated *Sw-Component* (cf. Figure 5) satisfies the good properties (H1–H4).

The proof is made by induction: assuming that two Sw-Component C_P (respectively C_Q) are *equivalent* to RT-LOTOS behaviors P (respectively Q), we prove that $C_{P[g\gg Q}$ is equivalent to $P[g\gg Q$ (the behavior over time must be accounted for). Intuitively $P[g\gg Q$ and $C_{P[g\gg Q}$ are timed bisimilar iff they perform the same action at the same time and reach bisimilar states.

More precisely, what we have to prove is that, from each reachable state, if a time move (respectively an action move) is possible in $P[g\gg Q$, it is also possible in $C_{P[g\gg Q}$ and vice-versa. We then ensure that the proposed translation preserves the sequences of possible actions as well as the occurrence dates of these actions. The proof is given in Appendix B.

6 Three case studies

Washing Machine:

The extended RT-LOTOS specification of the Washing-Machine of section 3 follows:

```
Specification WM[start,open,close]:exit
 behavior hide b_wash, e_wash in
  Machine[start,b_wash,e_wash]
  [close>>
  Cover[open,close]
 where

 process Machine[start,b_wash,e_wash]:exit:=
 start;delay(1,2)b_wash;delay(40,70)e_wash;exit
 endproc

 process Cover[open, close]: exit:=
 open; close; Cover[open,close]
 endproc
endspec
```

Following the translation procedure proposed in section 5 we obtain the SwTPN of Figure 6. Using the construction that preserves markings and *LTL* (Linear Temporal Logic) properties, proposed in [6] we obtain the state class graph of Figure 7. If we check the detailed textual description of the class graph output by TINA, we can locate the following suspended transitions: $start$ in class 1, b_wash in 12, e_wash in class 9 and $delay$ in classes 10 and 13.

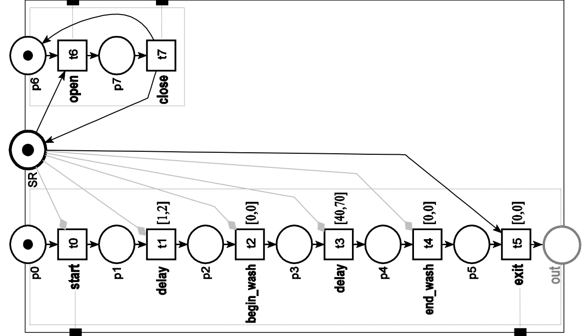


Figure 6. Washing-Machine SwTPN

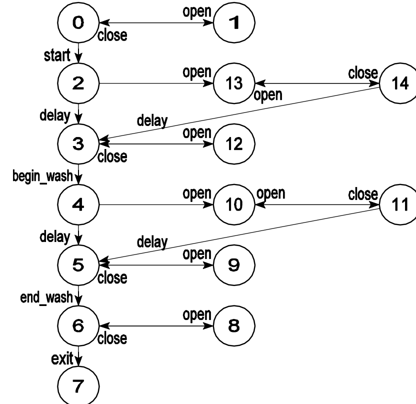


Figure 7. Washing-Machine class graph

A system of three tasks: The *suspend/resume* operator can be used for the modeling of scheduling problems in dense time semantics. The problem of real-time schedulability analysis involves establishing that a set of concurrent processes will always meet its deadlines when executed under a particular scheduling discipline on a given system. A schedulability discipline can be described by an RT-LOTOS specification which defines how a scheduler chooses among processes competing for processing time. The schedulability analysis is performed in two steps: First the RT-LOTOS specification is translated into SwTPN, whose states space is generated using a Petri net analyzer which further permits checking for missed deadlines. If no state with a missed deadline is reachable, then the system is schedulable.

Let us consider the case study presented in [10]. A controller controls three types of tasks by initiating a request int_i. The three tasks are executed on the same (and

unique) processor. $Task_3$ and $Task_1$ are periodic with a period of 150 (respectively 50) units of time. $Task_2$ is sporadic with a minimum interarrival time of 100. $Task_1$ has priority over both $Task_2$ and $Task_3$. $Task_2$ has priority over $Task_3$.

The RT-LOTOS specification is given follows.

```
Specification Three_Tasks_System : noexit
behviour hide int1,int2,int3, endT1, endT2, endT3 in
(Controller [int1,int2,int3]
|[int1,int2,int3]|
((Task1[int1,endT1]
 [endT2>> Task2[int2,endT2])
 [endT3>> Task3[int3,endT3]))
 where

 process Controller [int1,int2,int3] :noexit :=
     Launcher1[int1]
 ||| Launcher2[int2]
 ||| Launcher3[int3]
 where
  process Launcher1[int1]: noexit :=
  delay(50)(int1;stop
          ||| Launcher1[int1])
  endproc

  process Launcher2[int2]: noexit :=
  delay(150,INF)(int2;stop
               ||| Launcher2[int2])
  endproc

  process Launcher3[int3]: noexit :=
  delay(150)(int3;stop
           ||| Launcher3[int3])
  endproc
 endproc

 process Task1[int1, endT1]:noexit:=
 int1;delay(10,20)endT1; Task1[int1, endT1]
 endproc

 process Task2[int2, endT2]:noexit:=
 int2;delay(18,28)endT2; Task2[int2, endT2]
 endproc

 process Task3[int3, endT3]:noexit:=
 int3;delay(20,28)endT3; Task3[int3, endT3]
 endproc
endspec
```

Figure 8 depicts the resulting SwTPN. For this exam-

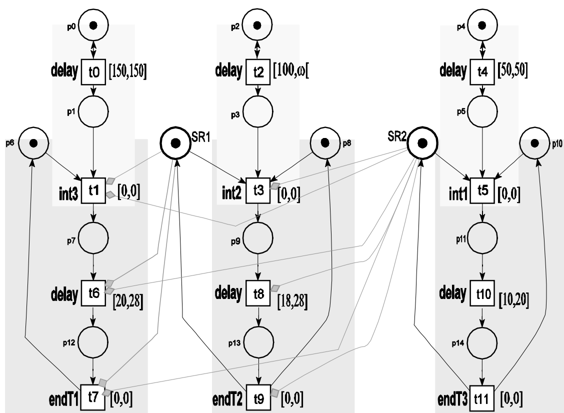


Figure 8. 3-Tasks system SwTPN

ple TINA builds a graph of 615 classes and 859 transitions. All the markings are safe (none of the places of the net contains more than one token), which implies schedulability. Transitions t_0, t_2 and t_3 don't satisfy the property $(m \setminus {}^{\bullet}t_i) \cap t_i^{\bullet} = \emptyset$. As a consequence the net of Figure 8 might be not safe, but the timing constraints prevent from the insertion of a new token in p_1, p_3 or p_5. None of places p_1, p_3 and p_5 contains more than one token in all possible computations, which means that the controller never initiates a task while its previous instance is pending. In other words, none of the $Task_i$ misses its deadline. Note that, quantitatives properties, like worst case response time (WCRT) may be checked by adding observers to the extended RT-LOTOS specification.

A Distributed Control system with Time-outs:
The description of the system is taken from [17]. The latter consists of two sensors and a controller that generates control commands to a robot according to the sensors readings (cf. Figure 9).

The two sensors share a single processor and the priority of sensor 2 for using the processor is higher than the priority of sensor1. If sensor1 loses the processor because of preemption by sensor2, it can continue the construction of its reading after the processor is released by sensor2. Each sensor constructs a reading and sends the latter to the controller. Each sensor takes 1 to 2 milliseconds of CPU time to construct a reading. Once constructed, the reading of sensor1 expires if it is not delivered within 4 milliseconds and the reading of sensor 2 expires if it is not delivered within 8 milliseconds. The Controller accepts a reading from each sensor in either order and then sends a command to the robot (signal action). The two sensors readings that are used to construct a robot command must be received within 10 milliseconds, if not, the first sensor reading is disregarded. The controller takes 3 to 5 milliseconds to synthesize a robot command.

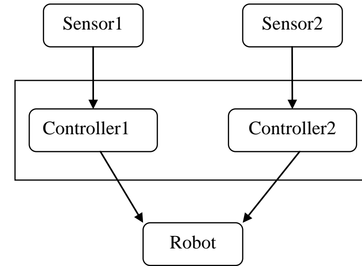


Figure 9. The robot controller architecture

From the extended RT-LOTOS specification of the system we obtain a SwTPN with 38 places and 34 transitions. Using the construction of [6], TINA builds in 80.59 secs[4] a state graph of 40 723 classes and 76 806 transitions.

7 Related Work

E-LOTOS and ET-LOTOS: A suspend/resume operator has been proposed for E-LOTOS[21]. An exception is specified inside the operator for resuming. Let us

[4] All the experiments described in this paper have been performed on a PC with 512 MB memory and a processor at 3.2 GHz.

comment on the use of an exception for resuming. In E-LOTOS an exception is a visible urgent event. Note that this definition violates the RT-LOTOS philosophy which states that one cannot enforce urgency on visible events. The only way to introduce urgency in RT-LOTOS is through the *hide* operator. Moreover, a systematic use of urgency for resuming may introduce unnecessary constraints and thus may result on deadlock situations. We think the specifier should have the freedom to specify when urgency is needed for resuming.

In [18] the authors propose a suspend/resume operator for ET-LOTOS. It allows self suspension. A visible action *g* is used for both self suspension and resuming. However an ambiguous interpretation of an occurrence of gate *g* may produce undesirable effects in case of recursive behaviours.

A key issue is not addressed in [18] [21]. The problem of suspension and resuming is addressed at the specification level. The problem of verifying the resulting behaviours is not tackled. At the present knowledge of the authors, there is no analysis tools implemented for such extensions. Our approach proposes a translation into SwTPN Thus extended RT-LOTOS specifications can effectively be model checked using the TINA tool.

Schedulability Analysis:A large number of techniques have been developed to model and solve scheduling problems. in [19] timed automata have been used to solve non-preemptive scheduling problems. in [1] stopwatch automata are used for solving preemptive scheduling problems. In this paper we use an algebraic way for specifying a tasks system and a scheduling discipline.
Techniques for real time schedulability analysis that rely on process algebra have already been published in [14][4]. These papers consider tasks as sequential processes. Schedulability analysis is performed either by reachability analysis [14] or by checking a bisimulation-based equivalence [4]. Unlike the process algebra presented in [14] [4], RT-LOTOS is not a formalism dedicated to schedulability analysis. RT-LOTOS is instead a wide-spectrum process algebra not endowed with built-in mechanisms for priority assignation or for resolving indeterminism in resource access. Nevertheless, the example in section 6 shows that extended RT-LOTOS may be used for schedulability analysis. In fact RT-LOTOS supports a wide-spectrum process model with various forms of inter-process interaction, communication and timing constraint expression. RT-LOTOS supports a dense time model. Conversely, [14] and [4] are discretely timed. When feasible, i.e. when a scheduling discipline may be described in extended RT-LOTOS, schedulability analysis based on RT-LOTOS is less restrictive than schedulability analysis based on the cited approaches. The difference comes from the expressiveness of RT-LOTOS, and consequently from the type of processes and time constraints that can be specified and analyzed in RT-LOTOS. In extended RT-LOTOS, both shedulability and functional correctness may be verified on the same specification. However, schedulability analysis with the approach proposed in this paper, is limited to fixed priority policies. This is because priorities are encoded in the static operators of extended RT-LOTOS.
A comparison of our work with existing approaches relating process algebras and Petri nets[9][22][16] can be found in [27].

8 Conclusions

The RT-LOTOS formal description technique supports three generic temporal operators that enable explicit and semantically well-founded description of real-time mechanisms and constraints. Nevertheless, real-time systems description in RT-LOTOS has been hampered by the lack of *suspend/resume* operator.

In this paper, we propose to extend RT-LOTOS with a suspend/resume operator '[g≫' which permits to suspend a behavior and to resume it later on. The proposed extension is given a formal semantics. Formal verification of extended RT-LOTOS specifications is also discussed. The paper presents an intermediate level model: *Sw-Component*. The latter are used as a gateway between extended RT-LOTOS and Stopwatch Time Petri nets, a new formalism supported by TINA [8].The use of an extended RT-LOTOS with a *suspend/resume* operator is illustrated on three examples. The RT-LOTOS to SwTPN translation procedure has now to be integrated into the *rtl2tpn* tool [28]. The translation algorithm of the *disrupt* operator is easily adapted to handle extended RT-LOTOS specifications.
This work is not limited to the verification of real-time systems directly specified in RT-LOTOS. The ultimate goal is to provide a more powerful verification environment for real-time systems modelled in TURTLE[3], a real-time UML profile built upon RT-LOTOS.

References

[1] Y. Abdeddaim and O. Maler. Preemtive job-shop scheduling using stopwatch automata. In *Proc. of TACAS'02*, pages 113–126.

[2] R. Alur, C. Courcoubetis, N. Hallbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. 138:3–34, 1995.

[3] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE : A real-time UML profile supported by a formal validation framework. *IEEE Transactions on Software Engineering*, 30(4), July 2004.

[4] H. Ben-Abdallah, J.-y. Choi, D. Clarke, Y.-S. Kim, I. Lee, and H.-L. Xie. A process algebraic approach to the schedulability analysis of real time systems. 15(3):189–219, 1998.

[5] G. Berry. Preemption in concurrent systems. In Springer-Verlag, editor, *Proc. of FSTTCS*, volume 761 of *LNCS*, pages 72–93, 1993.

[6] B. Berthomieu, D. Lime, O. Roux, and F. Vernadat. , reachability problems and abstract state spaces for time petri nets with stopwatches, to appear 2007. 2006.

[7] B. Berthomieu and M. Menasche. Une approche par énumération pour l'analyse des réseaux de Petri temporels. In *Actes de la conférence IFIP'83*, pages 71–77, 1983.

[8] B. Berthomieu, P. Ribet, and F. Vernadat. The TINA tool: Construction of abstract state space for petri nets and time petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[9] E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra*, volume ISBN: 3-540-67398-9 2001 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer-Verlag, New York, 2001.

[10] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Time state space analysis of real-time preemptive systems,-ieee trans. on software engineering-. 30(2):97–111, 2004.

[11] F. Cassez and K. Larsen. The impressive power of stopwatches. In *11th Int. Conf. on Concurrency Theory*, volume 1877 of *LNCS*, pages 138–152, University Park, P.A, USA, 2000. Springer-Verlag.

[12] J.-P. Courtiat. Formal design of interactive multimedia documents. In *FORTE'2003*, volume 2767 of *LNCS*, Berlin, 2003.

[13] J.-P. Courtiat, C. Santos, C. Lohr, and B. Outtaj. Experience with RT-LOTOS, a temporal extension of the LOTOS formal description technique. *Computer Communications*, 23(12):1104–1123, 2000.

[14] A. Fredette and R. Cleaveland. Rtsl: A language for real-time schedulability analysis. In *Proc. of the Real-Time Systems Symposium*, pages 274–283, Durham, North Carolina, 1993. Computer Society Press.

[15] H. Garavel and R.-P. Hautbois. An experiment with the lotos formal description technique on the flight warning computer of airbus 330/340 aircrafts. In *Proc. of the first AMAST International Workshop on Real-Time Systems*, IOWA, USA, 1993.

[16] H. Garavel and J. Sifakis. Compilation and verification of lotos specifications. *Proc the IFIP WG 6.1 International Symposium*, pages 379–394, 1990.

[17] T. A. Henzinger and P.-H. Ho. HYTECH: The cornell HYbrid TECHnology tool. In *Hybrid Systems*, pages 265–293, 1994.

[18] C. Hernalsteen and A. Fevrier. Introduction of a suspend/resume operator in et-lotos. volume 1231 of *LNCS*, pages 400–414, 1997.

[19] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using uppaal. *Nordic Journal of Computing*, 8:43–64, 2001.

[20] ISO. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. (8807:1989), September 1989.

[21] ISO. Information technology – enhancements to lotos (e-lotos). (15437:2001), 2001.

[22] M. Koutny. A compositional model of time petri nets. In *Proc. of the 21st Int. Conf. on Application and Theory of Petri Nets (ICATPN 2000)*, number 1825 in LNCS, pages 303–322, Aarhus, Denmark, 2000. Springer-Verlag.

[23] P. Merlin. *A study of the recoverability of computer system*. PhD thesis, Dep. Comput. Sci., Univ. California, Irvine, 1974.

[24] E.-R. Olderog. *Nets, Terms and Formulas*. Cambridge University Press, 1991.

[25] O. Roux and A.-M. Dplanche. A time petri net extension for real time task scheduling modeling. In *Eur. journal of Automation(JESA)*, 2002.

[26] O. Roux and D. Lime. Time petri nets with inhibitor hyperarcs, formal semantics and state space computation. In *Proc. Int. Conf. on Applications and Theory of Petri Nets*, Bologna, Italy, 2004.

[27] T. Sadani, M. Boyer, P. De Saqui-Sannes, and J.-P. Courtiat. Effective representation of RT-LOTOS terms by finite time Petri nets. In *FORTE 2006, Paris, France*, number 4229 in Lecture Notes in Computer Science, pages 404–419. Springer, 2006.

[28] T. Sadani, J.-P. Courtiat, and P. de Saqui-Sannes. From rt-lotos to time petri nets. new foundations for a verification platform. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'2005)*, pages 250–259. Computer Society Press, 2005.

A. Definition (First actions set)

Let C_P be a Sw-Component associated with an RT-LOTOS behavior P. The set of first actions $\mathcal{FA}(C_P)$ can be recursively built as follows:

$\mathcal{FA}(C_{stop}) = \emptyset$
$\mathcal{FA}(C_{exit}) = \{t_{exit}\}$
$\mathcal{FA}(C_{a\{d\}P}) = \{t_a\}$
$\mathcal{FA}(C_{delay(d)P}) = \mathcal{FA}(C_{latency(d)P}) = \mathcal{FA}(C_P)$
$\mathcal{FA}(C_{P \gg Q}) = \mathcal{FA}(C_P)$
$\mathcal{FA}(C_{P|||Q}) = \mathcal{FA}(C_{P[>Q} = \mathcal{FA}(C_{P[]Q}) = \mathcal{FA}(C_{P[g\gg Q}) = \mathcal{FA}(C_P) \cup \mathcal{FA}(C_Q)$
$\mathcal{FA}(C_{P;Q}) = \mathcal{FA}(C_P)$
$\mathcal{FA}(C_{\mu X.(P;X)}) = \mathcal{FA}(C_P)$ where $C_{\mu X.(P;X)}$ is the component which performs P's actions ad infinitum.
$\mathcal{FA}(C_{hide\ a\ in\ P}) = h_a(\mathcal{FA}(C_P))$ where $h_a(x) = x$ if $x \neq a$ and $h_a a = i_a$.

B. The proof

Notations: A paragraph starting with $\stackrel{\Re}{\rightarrow}$ proves that each time progression which applies in the RT-LOTOS behavior P[g≫Q is acceptable in its associated component $C_{P[g\gg Q}$. $\stackrel{\Re}{\leftarrow}$ denotes the opposite proof.
$\stackrel{a}{\rightarrow}$ (respectively $\stackrel{a}{\leftarrow}$) are used for the proof on action occurrence.
It is worth to mention that during the execution of a component the structure of the SwTPN remains the same (only the markings and temporal intervals are different), while the structure of an RT-LOTOS term may change through its execution. As a result, the SwTPN encapsulated in a component may not directly correspond to an RT-LOTOS behavior translation but is strongly bisimilar with a SwTPN which does correspond to an RT-LOTOS expression translation (The same SwTPN and current state without the unrechable structure).

$\stackrel{\Re}{\rightarrow}$ A time move in P[g≫Q falls into two categories:

1. Time move into both processes P and Q (SOS rule 7). If a time move is possible in both processes, then the first action of Q (respectively C_Q) is enabled but has not occurred yet. By induction hypothesis a time move is also possible in C_P and C_Q. 'SR'(the only

introduced place) is an input place of C_Q's first action. 'SR' is then marked. Consequently, C_P is not suspended. C_P can age, hence a time move in both C_P and C_Q is also possible in $C_{P[g \gg Q}$.

2. A time move only on the suspending behavior Q (SOS rule 5). P is suspended at the occurrence of the first action of Q. If a time move is possible in process Q, by induction hypothesis it is also possible in C_Q. 'SR' is the unique newly introduced place in $C_{P[g \gg Q}$. 'SR' does not interfere with the evolution of C_Q after the occurrence of its first action. Hence in $C_{P[g \gg Q}$ a time move is only possible in C_Q.

$\stackrel{\Re}{\leftarrow}$ similarly to $\stackrel{\Re}{\rightarrow}$

$\stackrel{a}{\rightarrow}$ the occurrence of action 'a' in $C_{P[g \gg Q}$ is either:

1. the occurrence of a first action of Q. (SOS rule 3) By induction hypothesis, 'a' is also possible in C_Q. 'a' is enabled in C_Q. 'SR' is an input place for C_Q's first action and 'SR' is marked. Hence 'a' is also possible in $C_{P[g \gg Q}$.

2. 'a' is not a first action of Q. Q is active and P is suspended (SOS rules 4). By induction hypothesis 'a' is also possible in C_Q. 'SR' is not marked after the firing of C_Q's first action, but SR does not interfere with the evolution of C_Q (it serves as input place only for the first action of Q). Hence, 'a' is also possible in $C_{P[g \gg Q}$.

3. 'a' is an action of P. P is not suspended. By induction hypothesis 'a' is also possible in C_P. 'SR' is marked, hence 'a' is also possible in $C_{P[\gg Q}$.

$\stackrel{a}{\leftarrow}$ similarly to $\stackrel{a}{\rightarrow}$
(H1–H4) are trivially satisfied in $C_{P[g \gg Q}$.

Generation of tests for real-time systems with test purposes

Sébastien Salva
LIMOS
Campus des Cézeaux
Université de Clermont Ferrand
salva@iut.u-clermont1.fr

Patrice Laurençot
LIMOS
Campus des Cézeaux
Université Blaise Pascal
laurencot@isima.fr

Abstract

Usually, the notion of time introduces space explosion problems during the generation of exhaustive tests, so test-purpose-based approaches have been developed to reduce the costs by testing (usually on the fly) the critical parts of the specification. In this paper, we introduce a test-purpose-based method which tests any behaviour and temporal properties of a real-time system. This method improves the fault detection in comparison with other similar approaches by using a state-characterization-based technique, which enables the detection of state faults on implementations. An example is given with the MAP-DSM protocol modelled with two clocks.
key words:Timed automata, conformance testing, test purpose

1. Introduction

Computer applications are being increasingly involved in critical, distributed and real-time systems. Their malfunctioning may have catastrophic consequences for the systems themselves, or for the ones who are using them. Testing techniques are used to check various aspects of such systems. Different categories of test can be found in literature: performance testing, robustness testing, interoperability testing and conformance testing which will be considered here.

Conformance testing consists in checking if the implementation is consistent with the specification by stimulating the implementation and observing its behaviour. *Test cases* which consist of interaction sequences are applied on the implementation via a test architecture [13, 23]. This one describes the configuration in which the implementation is experimented, which includes at least the implementation interfaces (called PCO, *point of control and observation*) and the tester which executes the test cases to establish the test verdict:

- *pass:* no error has been detected.
- *fail:* there is at least an error on the implementation.
- *inconclusive:* pass and fail cannot be given (the test cannot be performed).

Many testing methods have been proposed for generating automatically test cases from untimed systems [30, 9, 24, 3] or timed ones [5, 13, 10, 22, 29]. Most of the timed ones are exhaustive methods which generally transform specifications into larger automata (such as region graphs [13, 23], grid automata [29, 12], or SEA [19, 18]) to generate test cases on the complete specification. This kind of method is interesting and usable with small systems but can end in a space explosion problem (usually obtained from state explosion) with larger ones. So, others techniques called test-purpose-based approaches, have been proposed to test the most critical systems parts. These ones check local implementation parts from test requirements given by engineers, which are called *test purposes*. The conclusion of the test is given here by checking the satisfaction of the test purpose in the implementation.

Some test purpose based methods have been proposed [7, 8, 20, 28, 18] to test timed systems. These ones strongly reduce the test cost and can be generally used in practice to test specification properties on implementations. However, faults like extra(missing) states and transfer faults cannot be detected with the previous techniques. Such faults can modify the system internal state (this one becomes unknown and faulty), so detecting them is important.

In this paper, we introduce a test-purpose-based method which can test the conformance and the robustness of implementations, by testing any temporal or behaviour properties belonging to the specification (called Accept properties), but also any other ones given by designers (Refuse properties). The test case generation is performed by a *timed synchronous product* which combines the specification with the test purposes and prevents state explosion. With this product, we obtain a graph which includes the specification and the test purpose properties. Furthermore, to improve the fault detection, we use a state-characterization-based approach to identify each state visited in the implementation. So, missing and transfer faults can be detected.

This article is structured as follow: Section 2 describes the theoretical framework needed in this study. Section 3 provides an overview of testing methods, and a related works on timed testing with test purpose based approaches. Section 4 introduces the concept of Timed test purposes. The testing method is described in Section 5. We apply this one on a real system, which is a part of the MAP-DSM protocol. Then, we give the fault coverage of the method in Section 6. Finally, we give an overview of an academic test tool, which implements this testing method, in Section 7 and we conclude in Section 8.

2. Definitions

2.1. The Timed Input Output Automaton model

TIOA (Timed Input Output Automata) are graphs describing timed systems. This model, extended from the timed automaton one [1], expresses time with a set of clocks which can take real values (dense time representation) and by time constraints, called clock zones, composed of time intervals sampling the time domain. Actions of the system are modelled by symbols labelled on transitions: input symbols, beginning with "?" are given to the system, and output ones, beginning with "!" are observed from it. A TIOA transition, labelled by an input symbol ?a, can be fired if the system receives ?a while its time constraint is satisfied. In the same way, a TIOA transition, labelled by an output symbol !a, is fired if !a is observed from the system while the time constraint is satisfied.

Definition 1 (Clock zone) *A clock zone Z over a clock set C is a tuple $< Z(x_1), ...Z(x_n) >$ of intervals such that $card(Z) = card(C)$ and $Z(x_i) = [a_i\ b_i]$ is a time interval for the clock x_i, with $a_i \in \mathbb{R}^+$, $b_i \in \{\mathbb{R}^+, \infty\}$.*
If X_i is the clock value of the clock x_i, we say that a clock valuation $v = (X_1, ..., X_n)$ satisfies Z, denoted $v \models Z$ iff $X_i \in Z(x_i)$, with $1 \leq i \leq n$.
For two clock zones Z and Z', we denote some operators:

- $Z \cap Z' = \{v \mid v \models Z \text{ and } v \models Z'\}$
- $Z/Z' = \{v \mid v \models Z\}/\{v' \mid v' \models Z'\}$

Definition 2 (Timed Input Output Automata (TIOA)) *A TIOA $\mathcal{A}$ is a tuple $< \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}} >$ where:*

- *$\Sigma_{\mathcal{A}}$ is a finite alphabet composed of input symbols and of output symbols,*
- *$S_{\mathcal{A}}$ is a finite set of states, $s^0_{\mathcal{A}}$ is the initial one,*
- *$C_{\mathcal{A}}$ is a finite set of clocks,*
- *$E_{\mathcal{A}}$ is the finite transition set. A tuple (s, s', a, λ, Z) models a transition from the state s to s' labelled by the symbol a. The set $\lambda \in C_{\mathcal{A}}$ gathers the clocks which are reset while firing the transition, and $Z =< Z(1), ..., Z(n) >_{(n=card(C_{\mathcal{A}}))}$ is a clock zone.*

A TIOA example, modelling a MAP-DSM part, is given in Figure 1. Among the protocols used with GSM (Global system for Mobile communication), nine protocols are grouped into the MAP (Mobile application part). Each one corresponds to a specific service component. The Dialog State Machine (DSM) manages dialogs between MAP services and their instantiations (opening, closing...). A DSM description can be found in [6]. The specification of Figure 1, describes the request of the MAP service by an user(?I3). This one can invoke several MAP requests (?I4) which aim to start some services (!O3). A dialog can be accepted then established or it can be abandoned (!O5 or !O9).

If we consider the transition $(Tmp2, IDLE, !O9, \{X\ Y\}, < X_{[4\ +\infty[}Y_{[4\ +\infty[} >)$, the two clocks x and y must have an greater value than 4 so that the system produces the symbol !O9. After this execution , x and y are reset.

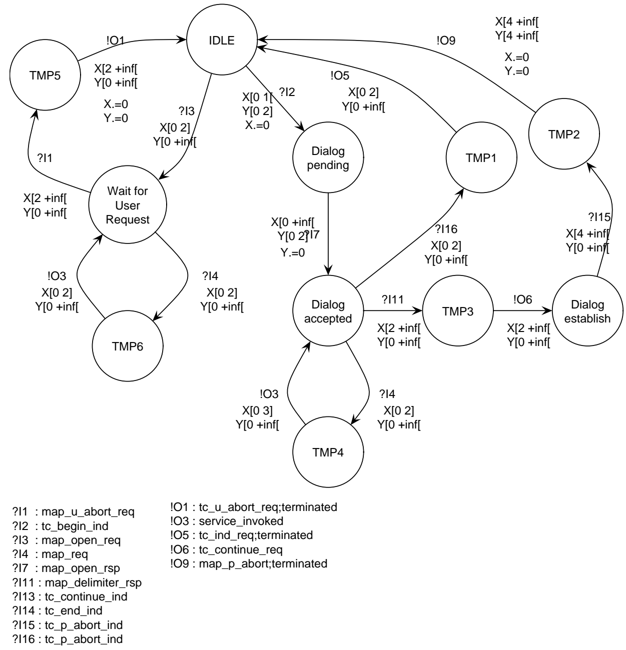


Figure 1. A TIOA

2.2. Fault model

The fault model is a set of potential faults (untimed and timed ones) that can be detected on implementations by the testing process. For the TIOA semantic, the fault model can be found in [13, 12]. This one is composed of:

- **Output faults**: An implementation produces an output fault, for a specification transition $(s, s', !a, \lambda, Z)$, if it does not respond with the expected output symbol $!a$.
- **Transfer faults**: An implementation produces a transfer fault if from a state, it goes into a state different from the expected one by accepting an input symbol or by giving an output one.

- **Extra state faults**: An implementation is said to have an extra (missing) state if its number of states must be reduced (increased) to be equal to the number of states of the specification.

- **Time constraint widening fault**: Such a fault occurs if the implementation does not respect the time delay granted by a specification clock constraint, that is if the upper (or lower) bound of a clock constraint is higher (smaller) in the implementation. This fault may occur on input or output symbols: for an output one, the implementation does not respond in the expected delay given by the specification, for an input symbol, the implementation accepts the input symbol in delays wider than the one given by the specification.

- **Time constraint restriction fault**: This fault occurs only with input symbols. An implementation produces this fault if it rejects an expected input symbol in delays satisfying the clock constraint given by the specification. In this case, the clock constraint of the implementation is more restrictive than the specification one. Since output symbol cannot be controlled by the system environment, an implementation that produces an output symbol in a more restrictive delay than the one specified is seen as a valid restriction of the specification.

3. Related Works

In the literature, testing methods can be grouped into two categories:

- **the exhaustive testing methods**, which involve generation of test cases on the complete specification, execution of the test cases on the implementation and analysis of the test results. To describe the set of correct implementations, a conformance relation is first defined, then test cases are given or generated from the specification to check if the relation is satisfied or not. Some works about timed systems testing can be found here [5, 13, 22, 29, 12].

- **the non exhaustive testing methods [7, 8, 21, 20, 28, 11, 18, 2, 14]**, which aims to test local parts of implementations. This concept aims to check if a set of properties, called a test purpose, can be executed on an implementation during the testing process. Test purpose can be either manually given by designers, or can be automatically generated [7, 17]. Then, test cases are generally generated from the test purposes and from some specification parts, reducing the specification exploration in comparison with exhaustive methods (reducing in the same time the test costs). Finally, test cases are executed on the implementation to observe its reactions and to give a verdict [28].

  In [8, 20], the authors use time automata to model the specification and the test purpose. Test cases are generated by synchronizing the specification with the test purpose and by extracting the paths which contain all the test purpose properties. During the synchronization, a reachability analysis is performed to keep only the reachable transitions. This method needs for each transition a resolution of linear inequalities and also a DFS algorithm to search some clocks constraints. The number of inequalities is proportional with the number of clocks and the transitions they constrained, consequently the resolution is generally costly.

  In [28], the specification and the test purpose, modelled with timed automata are translated into region graphs to sample the time domain into polyhedrons. The test cases are generated by synchronizing the specification region graph and the test purpose one. Each region clock of the region graph is accessible from the initial one, so a final test case can be completely executed on implementations. However, the region graph generation is costly and can suffer from state explosion.

  In [18], the test tool TGV [15] has been extended to test timed systems. This method can test non deterministic systems and takes into account the quiescence of states. Test purposes and specification are translated into non real time automata (SEA), then the TGV method is adapted and used to generate test cases.

  In [2], the authors use specifications and test purposes modelled by TIOA. Then, they search for a feasible path which match the specification and the test purpose with a DFS algorithm.

  In [14], test purposes are modelled by Message Sequence Charts (MSC). These ones are converted into TIOA. Then, the specification and test purposes are converted into grid automata. Finally, test cases are generated by using the synchronous product defined in [8].

In this paper, we propose a new test purpose definition to generate test cases which can test the conformance of timed system as well as their robustness by defining Refuse properties, that is test purpose properties which do not belong to the initial specification. So these ones can simulate the execution of different failures, like byzantine or scheduling ones, in order to check if the system can still respond correctly despite these errors. We do not translate timed automata into larger models to apply existing untimed test purpose methods on them [28, 18, 14]. We define a new timed synchronization product on timed automata which also takes into account Refuse properties. We also propose to improve the fault detection by enabling the detection of the missing state and transfer faults. We adapt a state characterization based approach, defined in

[26] for region graphs, to identify each system state with observable action sequences. With this state identification, missing and transfer faults can be detected.

Before describing the test case generation, we present our definition of timed test purposes.

4. Timed test purpose

Test purposes are graphs describing the requirements that engineers wish to test on the system implementation. These requirements can be specification properties which should be satisfied in the implementation during tests. We call them Accept properties. But, test purposes could also be constructed with properties which do not belong to the specification, that we call Refuse properties. These ones can be used to test the system robustness by checking if the system responds correctly despite the execution of unspecified actions.

So, we define that a *Timed Test Purpose* is a TIOA whose the states are either labelled by "accept" or "refuse" to model that transitions are composed of accept or refuse properties. An accept transition of the test purpose must exist in the specification. Its clock zone may be however more restrictive than the specification one.

Definition 3 (A Timed Test Purpose) *Let* $\mathcal{S} =< \Sigma_{\mathcal{S}}, S_{\mathcal{S}}, s^0_{\mathcal{S}}, C_{\mathcal{S}}, E_{\mathcal{S}} >$ *be a TIOA describing a specification. A timed test purpose* $\mathcal{TP}$ *is a TIOA* $< \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s^0_{\mathcal{TP}}, C_{\mathcal{TP}}, I_{\mathcal{TP}}, E_{\mathcal{TP}} >$ *where:*

- $C_{\mathcal{TP}} \subseteq C_{\mathcal{S}}$,
- $S_{\mathcal{TP}} \subseteq S_{\mathcal{S}} \times accept, refuse$ *is a set of states such that each state* $s' \in S_{\mathcal{TP}}$ *is labelled either by:*
  - *ACCEPT: if* s' *is the initial state of* $\mathcal{TP}$, *or if* $\forall (s, s', a, \lambda, Z) \in E_{\mathcal{TP}}, \exists (s_1, s_2, a, \lambda_2, Z_2) \in E_{\mathcal{S}}$ *such as* $Z \subseteq Z2$, $s \in \{(s_1, accept), (s_1, refuse)\}, s' = (s_2, accept)$
  - *REFUSE: otherwise*

Definition 4 (Accept and Refuse transition) *We call a transition* (s, s', a, λ, Z) *an accept transition iff* s' *is labelled by ACCEPT. We call it a REFUSE transition otherwise.*

A timed test purpose example is given in Figure 2. This one checks if after having a dialog accepted (?I7), a dialog can be established (!O6) during a more restrictive clock zone than the specification one.

Test purposes for testing the system robustness

Robustness aims to check the system behaviour under the influence of external errors (byzantine failure, bus error, scheduling problem, ...). Mutations are generally injected into test cases to simulate these errors. Some well-known mutations can be found in [16]:

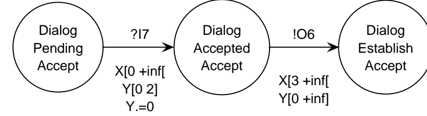


Figure 2. A test purpose example for the MAP-DSM protocol

1. Replacing an input action, to simulation that an unexpected action is received by the system from its external environment.

2. Changing the instant of an input action occurrence to simulate that the good input action is received later than expected

3. Exchanging two input actions to simulate an scheduling problem with external components to the system

4. Adding an unexpected action to simulate that an external component has send an additional action to the system.

5. Removing an action to simulate the lost of a information

Refuse properties are used here to model mutations, which are injected into test purposes and finally into test cases. Refuse properties can be added to test purposes by hands for specifying a precise error, or can be generated by some methods [16, 27]. The test purpose example of Figure 3 contains a refuse property which check that during the establishment of a connection between the MAP server and a service provider (?I11 !O6), the dialog cannot be aborted (?I16=tc_p_abort_ind). The action ?I16 is an unexpected action for the system. The test purpose also checks that the system continue to establish the dialog despite ordering the abort.

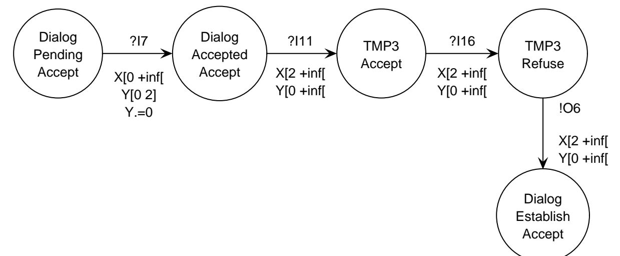


Figure 3. A test purpose with refuse properties

5. Test case generation

5.1. Testing hypotheses

Some assumptions are required on the implementation under test and on the specification. The "Implementation Reset" and "Determinism" hypotheses are required to execute the test cases. Indeed, without reset function, the tester cannot execute several test cases on the implementation, and if the implementation is nondeterministic, it may be uncontrollable and thus not testable. The two last hypotheses are required for using a state characterization based approach. These ones ensure and allow to identify each specification state.

Implementation Reset After each test, implementations can be reset to the initial state.

Determinism The specification must be timed deterministic on the set of alphabet. 1. from any state, we cannot have two outgoing transitions labelled with the same symbol. 2. we cannot have an outgoing transition, labelled with an input symbol and an outgoing transition labelled with an output one, whose the timing constraints are satisfied simultaneously. These properties ensure that a determined implementation path can be covered during the tests.

Minimality The specification must be minimal on the state set.

Completely specified system The specification must be completely specified on the set of input symbols (each input symbol is enabled from each state).

Remark 5 *To complete a specification on the set of input symbols, we propose to add a trap state $s_\sqcup$ and to complete each state s with outgoing transitions $(s, s_\sqcup, ?I, \lambda, G)$ from s to $s_\sqcup$. These transitions model the external actions refused by $\mathcal{A}$ and improve the observability and the controllability of the specification. So, the complete TIOA $\mathcal{UP}_\mathcal{A} =< \Sigma_{\mathcal{UP}_\mathcal{A}}, S_{\mathcal{UP}_\mathcal{A}}, s^0_{\mathcal{UP}_\mathcal{A}}, C_{\mathcal{UP}_\mathcal{A}}, I_{\mathcal{UP}_\mathcal{A}}, E_{\mathcal{UP}_\mathcal{A}} >$, derived from $\mathcal{A}$ can be obtained with these rules:*

- $\Sigma_{\mathcal{UP}_\mathcal{A}} = \Sigma_\mathcal{A}$, $S_{\mathcal{UP}_\mathcal{A}} = S_\mathcal{A} \cup \{s_\sqcup\}$,
- $s^0_{\mathcal{UP}_\mathcal{A}} = s^0_\mathcal{A}$, $C_{\mathcal{UP}_\mathcal{A}} = C_\mathcal{A}$,
- $I_{\mathcal{UP}_\mathcal{A}} = I_\mathcal{A} \cup \{Z' \mid \forall s' \in S_\mathcal{A}(s, s', ?I, \lambda, Z) \notin E_\mathcal{A}, Z' =< [0 +\infty[...[0 +\infty[>\}$ $\cup\{Z' \mid \exists s' in S_\mathcal{A}(s, s', ?I, \lambda, Z) \in E_\mathcal{A}, Z' =< [0 +\infty[...[0 +\infty[> /Z\}$,
- $E_{\mathcal{UP}_\mathcal{A}} = E_\mathcal{A} \cup \{(s, s_\sqcup, ?I, \lambda', Z') \mid \forall s' \in S_\mathcal{A}(s, s', ?I, \lambda, Z) \notin E_\mathcal{A}, \lambda' = \emptyset, Z' =< [0 +\infty[...[0 +\infty[>\}$ $\cup\{(s, s_\sqcup, ?I, \lambda', Z') \mid \exists s' in S_\mathcal{A}(s, s', ?I, \lambda, Z) \in E_\mathcal{A}, \lambda = \emptyset, Z' =< [0 +\infty[...[0 +\infty[> /Z\}$ $\cup\{(s_\sqcup, s_\sqcup, ?I, \lambda, Z) \mid ?I \in \Sigma_\mathcal{A}\}$

Test purposes are often composed of some specification actions, but not of complete specification action sequences [8, 28, 11, 18, 2, 14]. Test purposes may also be inconsistent with the specification, especially when we use refuse properties. So, test purposes based methods generally synchronise the test purpose with the specification to obtain paths which can be completely executed from the initial system path. Moreover, our testing method needs a state characterization based step to detect missing and transfer faults. So, these two steps are first presented below:

5.2. Timed Synchronous Product

The timed synchronous product aims to combine a test purpose and a specification to obtain paths which can be executed on the implementation. In comparison with the timed synchronous product that we have defined in [28] for region graph models, this one takes into account Refuse properties and injects them into the final test cases.

Consider two transitions, $s_1 \xrightarrow{A, Z_\mathcal{S}} s_2$ of a specification $\mathcal{S}$ and $s'_1 \xrightarrow{A, Z_{\mathcal{TP}}} s'_2$ of a timed test purpose $\mathcal{TP}$, labelled with the same symbol "A". By synchronizing them, we generate different clock zones, depending on $Z_\mathcal{S}$ and $Z_{\mathcal{TP}}$. The different kinds of synchronized clock zones are:

- **PASS clock zone:** The clock zone Z_{pass} gathers the values which satisfy the execution of the two transitions, that is the ones which belong to $Z_\mathcal{S} \cap Z_{\mathcal{TP}}$. If the transition is executed in this clock zone during the test, the test purpose transition is satisfied.
- **INCONCLUSIVE clock zone:** The clock zone $Z_{inconclusive}$ represents the values which satisfy the execution of the specification transition, but not the execution of the test purpose one. INCONCLUSIVE clock zones ensure that test cases can be executed on implementations, even though the test purpose cannot be satisfied. INCONCLUSIVE clock zones allow to give an inconclusive result, that means some specification properties have been tested instead of the test purpose ones. $Z_{inconclusive}$ contains values of $Z_\mathcal{S}/Z_{pass}$.
- **FAIL clock zone:** The FAIL clock zones represent the values which do not satisfy the execution of the specification transition. In this case, if the transition is executed in a FAIL clock zone during the test, the implementation is faulty.

Figure 4 shows an example of synchronized clock zones.

Now, we give the definition of the timed synchronous product between a specification and a test purpose which may contain refuse properties.

Definition 6 (Timed synchronous product) *Let $\mathcal{S} =< \Sigma_\mathcal{S}, S_\mathcal{S}, s^0_\mathcal{S}, C_\mathcal{S}, I_\mathcal{S}, E_\mathcal{S} >$ and $\mathcal{TP} =< \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s^0_{\mathcal{TP}}, C_{\mathcal{TP}}, I_{\mathcal{TP}} E_{\mathcal{TP}} >$ be two TIOA. The Timed Synchronous*

Product between $\mathcal{S}$ and $\mathcal{TP}$ is a graph $\mathcal{SP} =< \Sigma_{\mathcal{SP}}, S_{\mathcal{SP}}, s^0_{\mathcal{SP}}, C_{\mathcal{SP}}, E_{\mathcal{SP}} >$ defined by:

- $\Sigma_{\mathcal{SP}} \subseteq \Sigma_{\mathcal{S}} \cup \Sigma_{\mathcal{TP}}$, $S_{\mathcal{SP}} \subseteq S_{\mathcal{S}} \cup S_{\mathcal{TP}}$, $s^0_{\mathcal{SP}} \subseteq s^0_{\mathcal{S}}$, $C_{\mathcal{SP}} \subseteq C_{\mathcal{S}} \cup C_{\mathcal{TP}}$,
- *$E_{\mathcal{SP}}$ is the set of transitions $s_i \xrightarrow{a,PASS(Z),INCONCLUSIVE(Z')} s_{i+1}$, with $s_i \in S_{\mathcal{SP}}$, $s_{i+1} \in S_{\mathcal{SP}}$, Z a PASS clock zone and Z' an INCONCLUSIVE one. This set is constructed with the following algorithm.*

Algorithm

Input: T(Test Purpose), S(Specification)
Output: SP(Synchronous Product)
BEGIN:
For each specification path PS of S, and For each test purpose path TP containing in the same order the accept transition symbols of TP
We scan each transition $tp \xrightarrow{A,Z_{TP}} tp'$ of TP and each transition $s \xrightarrow{B,Z_S} s'$ of PS

if the symbol $A == B$ **then**
//the specification and the test purpose transitions are synchronized
- **if** $Label(tp') == REFUSE$ **then** we add $sp \xrightarrow[PASS(Z_{TP})]{A} sp'$ to E_{SP}
- **else** we add $sp \xrightarrow[PASS(Z_S \cap Z_{TP}),INCONCLUSIVE(Z_S/Z_{TP})]{A} sp'$ synchronizing the test purpose and the specification
- **endif**

if the symbol $A \neq B$
//the specification and the test purpose transitions cannot be synchronized
- **if** $Label(tp') == ACCEPT$ **then** we add $sp \xrightarrow[PASS(Z_S)]{B} sp'$ to $E_{SP}$to reach a next synchronization
- **else**
  - we scan PS to find if a synchronization on the symbol A with $tp \xrightarrow{A,Z_{TP}} tp'$ is possible later
  - **if** it is possible, **then** we add $sp \xrightarrow[PASS(Z_S)]{B} sp'$ to reach this synchronization.
  - **else** we add $sp \xrightarrow[PASS(Z_{TP})]{A} sp'$
  - **endif**
- **endif**

if some PTP transitions are not used **then** we add them to E_{SP}
endif
END

We illustrate the timed synchronous product with this simple example. Consider the path of Figure 5, derived from the specification of Figure 1. This one is synchronized with the test purpose of Figure 2. The timed synchronous product is expressed in Figure 6.

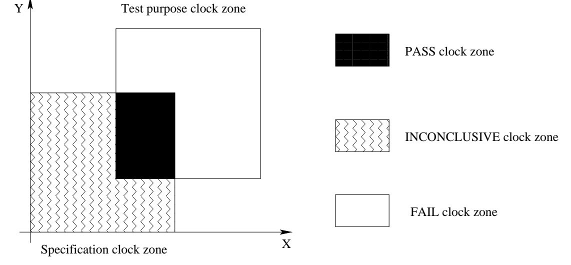


Figure 4. An example of synchronized clock zones with two clocks

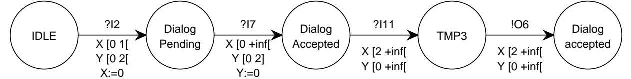


Figure 5. A specification path

5.3. State characterization set of TIOA

We have defined the state characterization based approach for region graphs in [26]. We have shown that the identification of two states depends on output symbols, which are observed during the system execution, and on the moments of these observations, that is the clock zones, for TIOA. So, to distinguish two TIOA states, we look for a transition sequence which provides either different output symbols, or the same ones with different clock zones or both. A state s is characterized by a identification set W_s if this one is composed of transition sequences which distinguish s from the other states. Finally, the state characterization set W is the union of the subsets W_{s_i} which characterize each state s_i. This is formally described in the following definition.

Definition 7 (Timed State Characterization Set W) *Let $\mathcal{IA} = (\Sigma_{\mathcal{IA}}, S_{\mathcal{IA}}, s^0_{\mathcal{IA}}, I_{\mathcal{IA}}, E_{\mathcal{IA}})$ be a TIOA satisfying the hypotheses of Section 5.1. Two states S and S' of $\mathcal{IA}$ are distinguished by a transition sequence $\sigma = (t_1, t_2, A_1, \lambda, Z_1)...(t_n, t_{n+1}, A_n, \lambda_n, Z_n)$, denoted $S\ D_\sigma\ S'$ iff*

1. *$\forall (t_k, t_{k+1}, A_k, \lambda_k, Z_k)(1 \leq k \leq n)$, with A_k an output symbol, we have a path $S \xrightarrow{A_1,\lambda_1 Z_1} S_2$ .... $S_{k-1} \xrightarrow{A_{k-1},\lambda_{k-1},Z_{k-1}} S_k \in (E_{\mathcal{IA}})^k$ and $(S_k, S_{k+1}, A_k, \lambda_k, Z_k) \in E_{\mathcal{IA}}$,*
2. *$\exists (t_k, t_{k+1}, A_k, \lambda_k, Z_k)(1 \leq k \leq n)$, with A_k an out-*

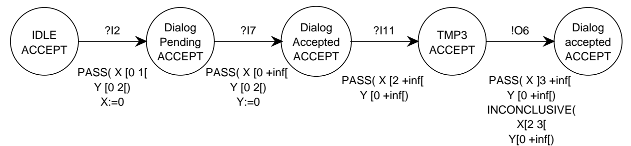


Figure 6. A synchronous product

put symbol, we have a path $S' \xrightarrow{A_1,\lambda_1,Z_1} S_2$ .... $S_{k-1} \xrightarrow{A_{k-1},\lambda_{k-1},Z_{k-1}} S_k \in (E_{\mathcal{IA}})^k$ *and* $(S_k, S_{k+1}, A_k, \lambda_k, Z_k) \notin E_{\mathcal{IA}}$.

We denote W_S*, the set of transition sequences allowing to distinguish* $S \in S_{\mathcal{IA}}$ *from the other states of* $S_{\mathcal{IA}}$. $W_S = \{\sigma_i \mid \forall S' \neq S \in S_{\mathcal{IA}}, S\ D_{\sigma_i}\ S'\}$.
Finally, a Timed Characterization Set of $\mathcal{IA}$*, denoted* $W_{\mathcal{IA}}$ *equals to* $\{W_{S_1}, ...W_{S_n}\}$*, with* $\{S_1, ..., S_n\} = S_{\mathcal{IA}}$.

A general algorithm of state characterization set generation can be found in [26].

If we take back our synchronous product example of Figure 6, the states can be distinguished with the following state-characterization sets. By applying this set to each pair of state, we always observe different output symbols at different time values, so we can distinguish them.

$W_{TMP3} = \{(TMP3, Dialog_establish, \{\}, !O6, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)\}$

$W_{Dialog_accepted} = \{(Dialog_accepted, TMP3, ?I11, \{\}, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)(TMP3, Dialog_establish, !O6, \{\}, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)\}$

$W_{Dialog_pending} = \{(Dialog_pending, Dialog_establish, ?I7, \{\}, < X_{[0\,+\infty[} Y_{[0\,2[} \rangle)(Dialog_accepted, TMP3, ?I11, \{\}, X_{[2\,+\infty]} Y_{[0\,+\infty]} >)(TMP3, Dialog_establish, !O6, \{\}, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)\}$

$W_{IDLE} = \{(IDLE, Dialog_pending, ?I2, \{\}, < X_{[0\,1[} Y_{[0\,2[} >)(Dialog_pending, Dialog_establish, ?I7, \{\}, < X_{[0\,+\infty[} Y_{[0\,2[} >)(Dialog_accepted, TMP3, ?I11, \{\}, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)(TMP3, Dialog_establish, !O6, \{\}, < X_{[2\,+\infty]} Y_{[0\,+\infty]} >)\}$

$W_{Dialog_establish} = \{(Dialog_establish, TMP2, ?I15, \{\}, < X_{[4\,+\infty[} Y_{[0\,+\infty[} >)(TMP2, IDLE, !O9, \{X\ Y\}, < X_{[4\,+\infty[} Y_{[4\,+\infty[} >)\}$

5.4. The testing method

The testing method is composed of four steps. Steps 1 and 2 synchronize the test purpose with the specification to generate paths, including the test purpose, which can be executed on the implementation. Step 3 applies a state-characterization-based approach on the synchronized paths. Finally, step 4 performs a reachability analysis on the paths obtained from the previous step and modify the clock zones to ensure that the test cases can be completely executed on the implementation.

These test case generation steps are detailed below:

Let $\mathcal{S}$ be a TIOA, satisfying the previous hypotheses, and $\mathcal{TP}$ be a timed test purpose. The test case generation steps are:

- **STEP1: Specification path search:** We extract the specification paths which can be synchronized with the test purpose. Instead of synchronizing all the specification with the test purpose, we extract only the needed. So, the transition sequences of $\mathcal{S}$, containing in the same order all the Accept transition symbols of the test purpose, are first extracted and named $TS_1(\mathcal{S}), ..., TS_n(\mathcal{S})$. If this set is empty, the process terminates and the following steps cannot be performed. We use a DFS (Depth First Path Search) algorithm to generate these paths. The path extraction is performed depth wise, so only one specification local path is memorized at a time.
- **STEP2: Timed synchronous product:** Each transition sequence $TS_i(\mathcal{S})$ is synchronized with $\mathcal{TP}$. This operation generates a graph SP, including $\mathcal{TP}$ and respecting the temporal and behaviour properties of $\mathcal{S}$.
- **STEP3: State characterization set generation:** Each state S_i of SP is identified with W_{S_i} (cf Section 5.3). Then, we combine, with Π, the synchronous product and the state characterization set: $\Pi = SP \otimes W = \{p.(s_i, s_j, a, \lambda, Z).\sigma_j \mid \forall(s_i, s_j, a, \lambda, Z) \in E_{SP}$, p is a path of SP from s_0 to s_i, $\sigma_j \in W_{s_j}$ if s_j is labelled by ACCEPT, $\sigma_j = \emptyset$ otherwise$\}$. It's the concatenation of a path p (finished by its state s_i) with the state characterization set of s_i. If we combine the synchronous product example of Figure 6 and the state-characterization set of Section 5.3, we obtain the paths of Figure 7.
- **STEP4: Search of feasible paths:** Test cases are finally all the feasible paths of Π [2]. The feasibility problem, for a given path $p = t_1..t_n$, aims to determine if it exists a possible execution to reach the transition t_n, and to generate the clock zones over p for firing t_n. The approach, described in [2], adds a global clock h and then performs a reachability analysis from the first and the last transitions of the initial path. The obtained feasible path clock zones can be modelled with the global clock h or with the clocks used in the initial path. For example, the feasible paths of the Π set, illustrated in Figure 7, are given in Figure 8. These ones are the final test cases.

Test cases are then executed on the implementation from the initial state. Each input symbol is given to the implementation at a clock valuation of its PASS clock zone. If the system is not faulty, output symbols should be observed at clock valuations of PASS clock zones as well. So, by applying a test case transition $t = (l, l', A, \lambda, PASS(Z), INCONCLUSIVE(Z'))$ on the implementation I, we can observe some reactions, denoted $React(t)$, and we can give a local verdict for the transition. $React(t) =$

- $PASS_{action}$ iff A is an output symbol and A is received from the tester in the PASS clock zone, that is at a clock value $v \models Z$,

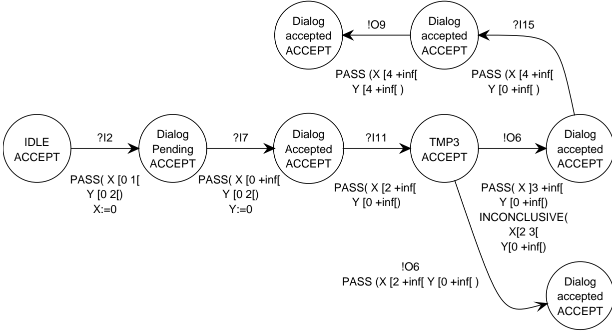


Figure 7. The synchronous product 6 combined with the state characterization sets

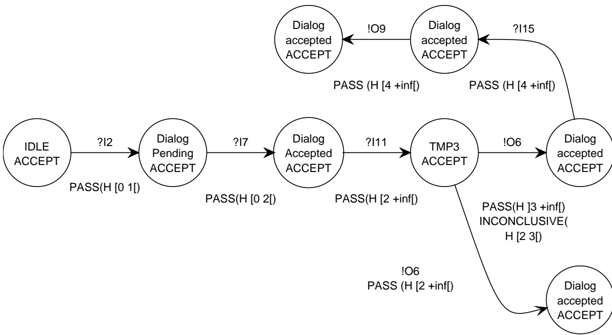


Figure 8. The test cases

- $INCONCLUSIVE_{action}$ if A is an input symbol or if A is an output symbol and A is received from the tester in the INCONCLUSIVE clock zone, that is at a clock value $v \models Z'$,
- $FAIL_{action}$ otherwise.

Finally, by executing the test cases and observing the implementation reactions, we can conclude on the success or on the failure of the test:

Definition 8 (Verdict assignment) *Let I be a system under test and* $T = (l_1, l'_1, A_1, \lambda_1, PASS(Z_1), INCONCLUSIVE(Z'_1))...(l_n, l'_n, A_n, \lambda_n, PASS(Z_n), INCONCLUSIVE(Z'_n))$ *be a test case. The verdict assignment* $V(I, T)$*, obtained by applying T on I, is given by:*

- *Pass iff* $\forall t = (l_i, l'_i, A_i, \lambda_i, PASS(Z_i), INCONCLUSIVE(Z'_i)) \in T$*, with* A_i *an output symbol, React*$(t) = PASS_{action}$*,*
- *Inconclusive iff* $\exists t = (l_i, l'_i, A_i, \lambda_i, PASS(Z_i), INCONCLUSIVE(Z'_i)) \in T$*, with* A_i *an output symbol, React*$(t) = INCONCLUSIVE_{action}$ *and iff* $\forall t' = (l_j, l'_j, A_j, \lambda_j, PASS(Z_j), INCONCLUSIVE(Z'_j)) \in T$*, with* A_j *an output symbol, React*$(t) \neq FAIL_{action}$*,*
- *Fail otherwise*

Method complexity: If N is the number of state and K the number of transitions of the specification, the test case generation complexity of our method is proportional to $C^2 * N + N * K + N + K$. For the first step, we use a DFS algorithm whose the complexity is proportional to $N + K$. The timed synchronous product complexity depends on the length of the paths to combine. In the worst case, this length equals to N and there is at most K specification paths. So, the complexity of the synchronous product is proportional to $N * K$. The step 3 complexity is proportional to $N^2 K$ [25]. In step 4, the search of feasible paths is proportional to $C * C * N$ [2], with C the number of clocks.

6. Fault coverage of the proposed method

In this Section, we introduce the fault coverage of our testing method. As a test purpose doesn't test the whole implementation of a system, the fault coverage is analyzed on a implementation part, called $I_{covered}$. Furthermore, to generate test cases, we use some specification paths, needed for the timed synchronous product. Let $S_{covered}$ be the set of these paths. $I_{covered}$ corresponds to implementation part tested by the test cases obtained from $S_{covered}$.

- *Output fault detection*: Output faults can be easily detected by firing all of the specification transitions. According to our definition of the timed synchronous product, each path of $S_{covered}$ exists completely in at least one test case. Moreover, we suppose that the system is deterministic. So, each transition of $I_{covered}$ is tested during the testing process.
- *Missing state fault detection*: Extra(missing) state faults are detected by checking if an extra or missing state exists in the implementation. As our method identifies each state, it can detect missing states on $I_{covered}$. Each state s_i of $S_{covered}$ is identified in the implementation and tested by test cases of the form $s_0 \xrightarrow{p} s_i.W_{s_i}$, with p a path from the initial state s_0 to s_i and W_{s_i} the subset allowing to identify s_i. Consequently, if a state is missing in $I_{covered}$ at least one test case cannot be completely executed.
- *Transfer fault detection*: Transfer faults can be easily detected by identifying the states of the implementation. So, any state-identification based technique, and particularly our method, detects transfer faults on $I_{covered}$. Each transition $t = (S_i, S_j, a, \lambda, G)$ of $S_{covered}$ is tested by a test case of the form $S_0 \xrightarrow{p} S_i \xrightarrow{a,\lambda,PASS(Z)} S_j.W_{S_j}$, with p a path from the initial state S_0 to S_i and W_{S_j} the subset allowing to characterize S_j. Consequently, the arrival state of the transition t in the implementation is tested and identified. So, transfer faults are detected.

- *Time constraint widening fault detection*: Time constraint widening faults are detected if at least an output symbol is not received by the tester in the time delay given by the specification. According to our definition of the timed synchronous product, each transition of $S_{covered}$ is visited during the testing process by at least one test case. Consequently, a test case transition $(s, s', a, \lambda, PASS(Z), INCONCLUSIVE(Z'))$ labelled by an output symbol, is tested by the tester which waits its receipt during the PASS clock zone Z. If no output symbol is received, a time constraint widening fault is detected on I.
For input ones, the method checks them only at clock valuations which belong to time delays given by the specification. So, time constraint widening faults can be detected with output symbols and not with input ones on $I_{covered}$.

- *Time constraint restriction fault detection*: In practice, it is unfeasible to detect all of the time constraint restriction faults. Consider a test case transition $(s, s', ?a, PASS(Z), INCONCLUSIVE(Z'))$, to detect the faults, the tester should send to the implementation the input symbol "?a" at all of the bounds of Z which are for each clock x_i the time values a_i and b_i such that $Z(x_i) = [a_i, b_i]$. Since the clocks are uncontrollable, these bounds are not necessary reached by the clocks.

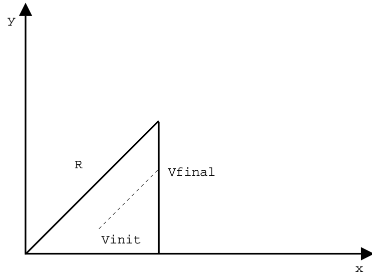


Figure 9. Reaching all the clock region bounds: a difficult issue

Consider the clock zone of the Figure 9. v_{init} represents the first clock valuation reached by the clocks in Z during an execution. v_{init} is not a bound of Z. So, if the implementation has a time constraint restriction fault between the bound of Z and v_{init}, the fault cannot be detected.

Consequently, we can detect such a fault if this one occurs during the execution. In this case, consider a test case $p.(s_i, s_j, ?a, \lambda, PASS(Z), INCONCLUSIVE(Z')).p'.W_{s_k}$. Let s_i be the implementation state reached by p and s_k the one reached by $p.(s_i, s_j, a, \lambda, PASS(Z), INCONCLUSIVE(Z')).p'$. If the implementation produces this fault, s_i rejects the input symbol "a" in delays given by Z. Therefore, the implementation stays in its current state s_i. Here, either the implementation rejects $p'.W_{S_k}$ too, or accepts it. If $p'.W_{S_k}$ is rejected, the implementation enters in a deadlock. Output symbols of $p'.W_{S_k}$ are not observed so the time constraint restriction fault is detected. If $p'.W_{S_k}$ is accepted by the implementation, according to the hypotheses (Section 5.1), S is minimal and deterministic therefore there exists an unique path from s_i to s_k, covered with $(s_i, s_j, a, \lambda, PASS(Z), INCONCLUSIVE(Z'))$ $.p'$ from s_i. Thus, a state s_l different from s_k is reached with p' from s_i. Since the state reached by p' is identified with W_{s_k}, if this one is different from s_k an error is produced. So, in both cases, time constraint restriction faults are detected.

7. Prototype tool functionality

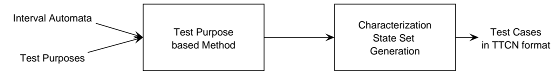


Figure 10. The test tool TTCG

We have implemented the previous methodology in an academic prototype tool, called TTCG (Timed Test Cases Generation). The description of its architecture is illustrated in Figure 10. The prototype tool takes specifications and test purpose modelled with TIOA. It is composed of two parts: the first one produces the timed synchronous product between the test purposes and some specifications paths. The second one produces the W set generation. The paths, obtained from the synchronous product and the state characterization set, are then concatenated to finally produce the test cases. These ones are given in TTCN or in Poscript format.

This tool has been written with the language C, excepted the second part which has been written in OpenMP to parallelize the W set generation. Clock zones modelling and operators on clock zone have been implemented with the Polylib library [31]. This one has a graphical interface which allows the user to load TIOA and test purposes. The amount of memory used depends on the specification. With the MAP-DSM specification (Figure 1), this one does not exceed 10 Mb.

8. Conclusion

In this paper, we have proposed a test purpose based approach which can test both the conformance and the robustness of implementations, by using test purposes composed of Accept and Refuse properties. This method uses a synchronous product between the specification and the test purpose to generate on the fly test cases and a state characterization based approach to improve the fault coverage by enabling the detection of transfer faults and miss-

ing state faults. The complexity is polynomial so we believe that this one can be used in practice.

Our approach could be extended for testing others aspects of timed systems like interoperability. The quiescence of critical states [4] could be tested with specific test purposes too, by checking if these states do not produce an output response without giving an input symbol. Moreover, this property could help to distinguish pair of states by considering the notion of quiescence as a special sort of output observation. As a consequence, the length of the state characterization set and so the test costs could be reduced.

References

[1] R. Alur and D. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[2] I. Berrada, R. Castanet, and P. Felix. A formal approach for real-time test generation. In *WRTES, satellite workshop of FME symposium*, pages 5–16, 2003.

[3] C. Besse, A. Cavalli, M. Kim, and F. Zaidi. Two methods for interoperability tests generation. an application to the tcp/ip protocol. 2004.

[4] L. B. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *FATES04 (Formal Approached to Testing of Software),Kepler University Linz, Austria*, pages 71–85, 2004.

[5] R. Cardel-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proc. of the 5th. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of LNCS, pages 251–261. SpringerVerlag, 1998.

[6] N. CARRERE. Dsm specification in lotos and test cases generation. *INT (French Telecommunication National Institute)*, 2001.

[7] R. Castanet, C. Chevrier, O. Kone, and B. L. Saec. An Adaptive Test Sequence Generation Method for the User Needs. In *IWPTS'95, Evry, France*, 1995.

[8] R. Castanet, P. Laurençot, and O. Kone. On the Fly Test Generation for Real Time Protocols. In *International Conference on Computer Communications and Networks, Louisiane U.S.A*, 1998.

[9] T. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, SE-4(3):178–187, 1978.

[10] D. Clarke and I. Lee. Automatic Generation of Tests for Timing Constraints from Requirement. In *International Workshop on Object-Oriented Real-Time Dependable Systems, California*. IEEE Computer Society Press, 1997.

[11] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France*, pages 211–225, May 2003.

[12] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed wp-method: Testing real-time systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Nov. 2002.

[13] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.

[14] A. En-Nouaary and G. Liu. Timed test cases generation based on msc-2000 test purposes. In *Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL'04), part of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE), Rennes, France*, Nov. 2004.

[15] J. C. Fernandez, C. Jard, T. Jron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96. LNCS 1102 Springer Verlag*, 1996.

[16] H. Fouchal, A. Rollet, and A. Tarhini. Robustness of composed timed systems. In *31st Annual Conference on Current Trends in Theory and Practice of Informatics, Liptovsky Jan, Slovak Republic, Europe, volume 3381 of LNCS*, pages 155–164, Jan. 2005.

[17] O. Henniger, M. Lu, and H. Ural. Automatic generation of test purposes for testing distributed systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, pages 185–198, Oct. 2003.

[18] A. Khoumsi, T. Jeron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, Oct. 2003.

[19] A. Khoumsi and L. Ouedraogo. A new method for transforming timed automata. In *Brasilian Symposium on Formal Methods (SBMF), Recife, Brazil*, Nov. 2004.

[20] O. Kone. A local approach to the testing of real time systems. *The computer journal*, 44:435–447, 2001.

[21] O. Kone and R. castanet. Test generation for interworking sytems. *Computer communications, Elsevier Science*, 23:642–652, 1999.

[22] B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In *TACAS01, vol. 2031 of LNCS, Genova, Italy*, pages 343–357, 2001.

[23] E. Petitjean and H. Fouchal. From Timed Automata to Testable Untimeed Automata. In *24th IFAC/IFIP International Workshop on Real-Time Programming, Schloss Dagstuhl, Germany*, 1999.

[24] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing Deterministic Implementations from Nondeterministic FSM Specifications. In *Proceedings of the 8th International Workshop on Test of Communicating Systems IWTCS'96 (Darmstadt, Germany)*, Amsterdam, september 1996. North-Holland.

[25] S. Salva and P. Laurenot. Gnration de tests temporiss oriente caractrisation d'tats. In *Colloque Francophone de l'ingénierie des Protocoles, CFIP*, Oct. 2003.

[26] S. Salva and P. Laurenot. A testing tool using the state characterization approach for timed systems. In *WRTES, satellite workshop of FME symposium*, 2003.

[27] S. Salva and P. Laurenot. Gnration automatique dobjectifs de test pour systmes temporiss. In *Colloque Francophone de l'ingénierie des Protocoles, CFIP, Bordeaux*, 2005.

[28] S. Salva, E. Petitjean, and H. Fouchal. A simple approach to testing timed systems. In *FATES01 (Formal Approaches for Testing Software), a satellite workshop of CONCUR, Aalborg, Denmark*, Aug. 2001.

[29] J. Springintveld, F. Vaandrager, and P. R. D'Argenio. Testing Timed Automata. *TCS*, 254(254):225–257, 2001.

[30] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[31] D. K. Wilde. A library for doing polyhedral operations. Technical report, IRISA. http://icps.u-strasbg.fr/PolyLib/.

Architectures and worst-case execution time estimation

Predictable Performance on Multithreaded Architectures for Streaming Protocol Processing

Matthias Ivers, Bhavani Janarthanan, Rolf Ernst[1]
{ivers,bhavani,ernst}@ida.ing.tu-bs.de

Abstract

Multithreaded architectures use processors with multiple hardware-supported threads which enable the efficient suspension of a running thread while it is waiting for a long-latency operation to finish. Multithreading is conceived as the panacea to fill the ever-growing gap between memory and processor speed.

In the domain of hard real-time systems, multithreaded architectures are hardly recognized as viable, as the possible gains of multithreading cannot be guaranteed easily, while the negative effect on worst-case execution time cannot be bounded easily.

We developed a hard real-time system based on high-speed (1.4GHz) microprocessors which use multithreading to allow high utilization despite long memory access times. In this paper, we describe a method how to benefit from multithreading while achieving good predictability.

1 Architecture

Architectures used in network processing share specific features to support traditional routing applications. Commonly found are algorithmic hardware acceleration units for checksum calculations, cryptography, longest prefix matching. Furthermore scratch-pad memories and intelligent memory controllers with direct memory access (DMA), atomic read-modify-write (RMW) and even linked list management support are used to improve system performance.

For our application domain the acceleration units designed to speed-up TCP/IP or Ethernet processing are of no interest. The features used by the design and critical for our methodology are the multithreaded high-speed RISC cores with their integrated control-stores and their virtual abundance of general purpose and memory transfer registers.

The architecture used for the presented work is based on Intel's network processors, the IXP-family. The Intel series of IXP network processors is based on so-called μEs (MicroEngines) which are programmable RISC cores with integrated instruction memory and 6kB memory for data storage arranged as 32bit-registers or 32bit-wide register-like RAMs. The processor supports up to 8 hardware threads scheduled in a cooperative (non-preemptive) round-robin fashion (cooperative multithreading). These hardware threads share the common instruction and data storage of the μE.

The number of μE available on an IXP processor varies from 2 μEs on an IXP-2325 to 16 μEs found on an IXP-2855. Our design (see Figure 1) under consideration here is based on 4 μEs, some coprocessors for hardware acceleration of macro functions and a specialized memory controller interfacing the system to 250 MHz SRAM.

1.1 Real-time Predictability of the Processor

An important factor for the selection of the μE as the processor of an architecture is its lean design and lack of heuristic features to speed-up the processing. The μEv2 does not feature branch predictors, out-of-order execution units or even caches. The design uses a clean five stage pipeline with virtually all relevant instructions taking a single cycle per pipeline stage. Only a small set of instructions can result in pipeline stalls.

All these facts account for the good analyzability of the "core" execution time on these processors. The only latency which cannot be precisely bounded statically, as it varies too much, is the execution time of instructions which involve the use of buses. The processor supports hardware-multithreading to achieve high utilizations even when the running software has to tolerate long latencies when transferring data to/from external units, , in particular external memory. The typical programming approach using the μE is to switch to a different thread while one thread is waiting for a memory transaction. Due to the large memory transaction latency, it is efficient to switch threads several times during one transaction thereby interleaving the transactions without overloading the buses.

2 Application

Traditionally network processors support applications running on Ethernet, TCP/IP or higher layer protocols. Our objective is to implement the low-level control plane of high-speed TDM telecommunication protocols which were never before implemented on a purely programmable platform. In our case we adress the ubiq-

[1] work is in part supported by a grant from Intel and the Lower Saxony Ministry of Economy

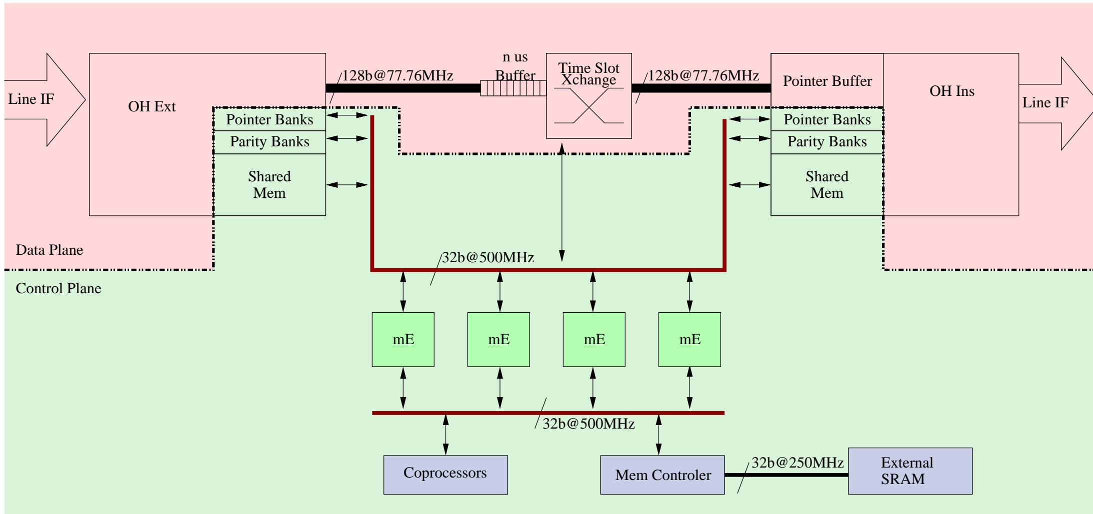


Figure 1. Architecture based on 4 μEv2

uitous standards SDH and SONET targeting line-speeds between 2.5Gbps and 10Gbps.

SDH and its counterpart SONET are the multiplexing standards formulated by ITU-T for optical telecommunication transport networks. They support the flexible and transparent transport of a mixture of protocols of different line rates in their virtual containers (VC) and provide sophisticated features of OAM through performance monitoring and protection mechanisms. The standards for SDH are specified in ITU-T G.707, network node interface for the synchronous digital hierarchy [4].

Figure 2 depicts the basic Synchronous Transport Module (STM-1) format for SDH. The transmission time of a frame is $125 \mu s$ corresponding to a rate of 8000 frames per second. The frame consists of overhead fields and a virtual container capacity accounting for a total of 2430 bytes resulting in the basic line rate of 155.52 Mbps. Higher level signals are integer multiples of the base rate formed by byte interleaving and multiplexing. A hierarchy up to 40 Gbps, STM-256/OC-768, has been defined. An STM-1 frame is arranged in 9 rows, each row consisting of 270 bytes (columns). The VC transported in an STM-1 has its own frame structure with nine rows and 261 columns. There are three layers of overhead bytes in an STM frame. Regenerator section, multiplex section and path as shown in Figure 2. The layers have a hierarchical relationship with each layer building on the services provided by all the lower layers. The overhead bytes provide information for synchronization, error monitoring, performance measures, tracing, status signaling, fault detections, automatic protection functions, network administration and management.

Apart from the overhead bytes, the pointer bytes are defined to indicate the phase alignment of the virtual containers within the STM frame. It is used to locate the start of a virtual container embedded in an STM-frame and to adjust for dynamic frequency and phase variations of the payload. The section, line and pointer overheads constitute the first 9 rows and 9 columns of the STM-1 frame and the higher order path overhead constitute the first column of the VC.

2.1 Characterization

SDH/SONET is a low-level networking protocol and designed to be efficiently implemented in hardware using a set of mostly independent finite state machines.

The protocol is designed in such a way that most control overhead can be processed completely in parallel with very little interdependency between different control functions. This fact is key to the success of the presented analysis method.

The finite state machines can be implemented in software and are quite small in code size and execution time. The average number of instructions per control function is very low: the largest control function executes at most 514 instruction on its longest path (average is below 100 instructions/control function). As a comparison note that the shortest deadline is in the order of 100,000 processor cycles.

The slack (difference between deadline and required execution time) of the control functions seem to be quite high at a first look. But we have to process many control functions at once (see below) and *all tasks* have to meet their deadline. The standards define these deadlines to be hard with an extremely low fault probability. This is necessary to reach the high persistency required for the optical switching devices. This is completely different from IP protocols where packet loss and retransmission are accepted.

So, the main factor in the processing of SDH/SONET is the sustainable throughput that must be reached. As higher line rates are achieved by byte-interleaving multiple frames, the overhead increases dramatically with increasing line speed. A 10Gbps signal has 41,472,000 overhead bytes per second. For that reason we want to maximize the reached throughput of the system while

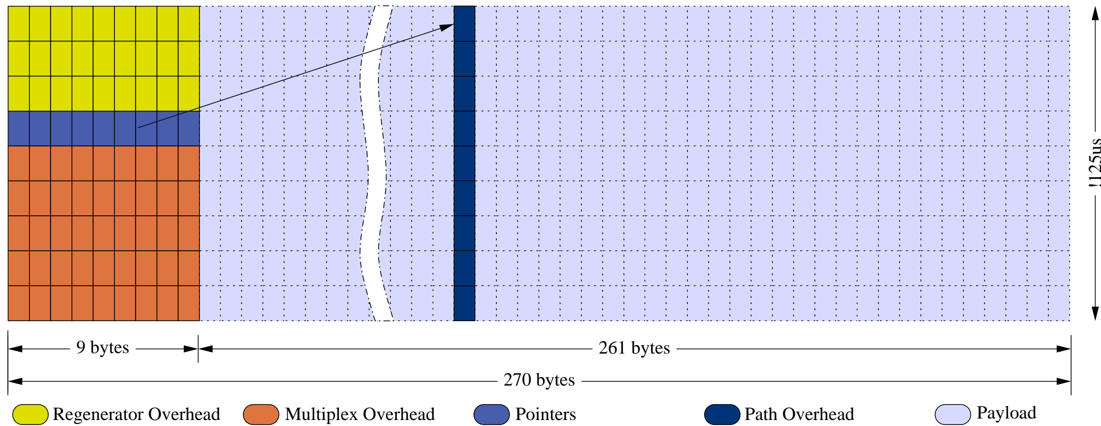


Figure 2. STM-1 Frame Structure

guaranteeing all deadlines.

3 Analysis

It is our objective to develop a performance analysis for high-speed multithreaded architectures. To do that, we start with an analysis that is aware of multithreading and does recognize multithreading in a beneficial - still conservative - way.

In a next step we'll introduce *a program transformation*. First we'll reschedule the memory operations with two effects: lowering the number of threads necessary to hide the latency and increasing the guaranteeable gain of multithreading.

We're going to extend that approach naturally by loop unrolling. Increasing the analytical gain of multithreading and, again, lowering the number of needed threads to hide the latency.

3.1 Known WCET Analysis for Multithreaded Processors

We implemented an execution time analysis for the μEv2-code based on previous work [1]. For the readers convenience, we're sketching the essential parts of the analysis here.

It starts with parsing the object files in order to generate a control flow graph (CFG) of the application (see Figure 3) [2] with basic blocks as nodes.

For a pair of basic blocks (a, b) each path from a to b represents a possible execution of the program. There are graph algorithms which efficiently enumerate all paths connecting two nodes. As the paths through the graph are possible traces through the real program, one can say that these graph algorithms enumerate all traces of the program.

3.1.1 Defining Execution Time

The execution time for normal basic blocks is easily derived for architectures under consideration. The used RISC processors does not have caches, branch predictors

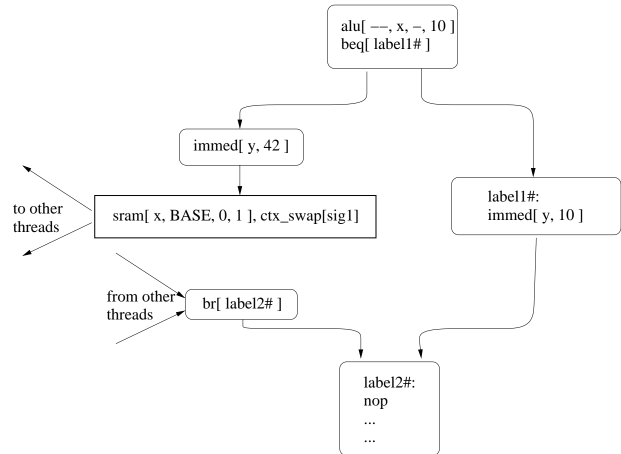


Figure 4. Fragmentation & Threading

or very deep pipelines. They are simple and lean in design, as the primary goal was to have a large number of equal processors on a single chip.

For that reason the execution time of a basic block without memory instructions can be estimated by summing up all individual instructions execution time and adding a constant architecture-specific overhead for each basic block. For the μEv2 the overhead can be completely hidden most of the time.

For nodes which access the memory the execution time has to be analyzed separately. There are methods to integrate the analysis of memory accesses and worst-case execution time [3]. This framework also supports the analysis of parallel memory accesses and considers the effect of pipelined memories, or chip interconnects. The presented method is orthogonal to the analysis of memory accesses. For that reason and to present results comparable to [1], we use the estimated upper bound of [1] and assume a memory access latency (including bus transfer) of 30 cycles (for 233 MHz μE) and 120 cycles (for 1400 MHz μE).

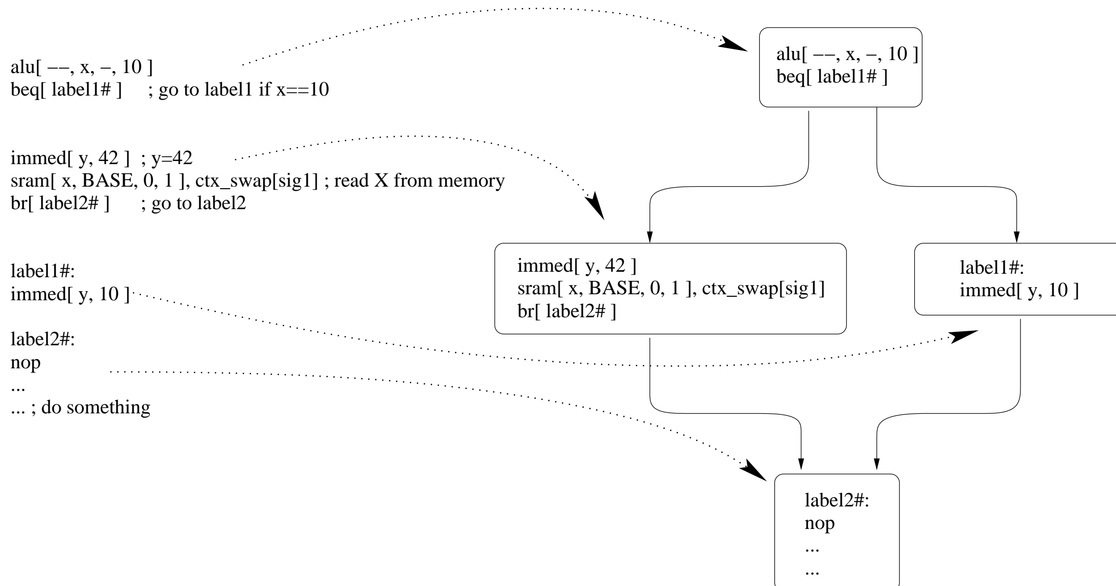


Figure 3. Assembly Code and Control Flow Graph

3.1.2 Obtaining the WCET

We transformed our program into a control flow graph and obtained execution times for the individual basic blocks. To calculate the WCET of our code, we have to designate a basic block where execution commences and a (possibly different) basic block where execution stops.

Once start- and end-node are designated standard graph algorithms are used to efficiently find a path connecting start and end node with maximal accumulated execution time [2].

The used algorithm allows to add further restrictions on the possible paths through the graph which makes it possible to model most semantic restrictions one can find in programs. We will make use of restrictions in the next section to eliminate impossible traces when analyzing multithreading.

3.1.3 Interleaved Extension

Non-preemptive multithreading has to be considered during the evaluation of the WCET. A multithreaded processor interleaves small traces of different threads in such a way that previously unavoidable utilization gaps in the processing of one thread are filled with execution of a different thread which is ready to run. This effect is beneficial as it uses previously stalling processing units. It can be disadvantageous to the individual WCET, as the concurrently running threads can block the processor while the thread under consideration is already ready to execute.

To enable the analysis of multithreaded architectures, basic blocks containing memory accesses with context switch opportunities are identified and and split, so that memory accesses with context switch opportunities constitute an individual basic block. A basic block with a context switch opportunity is also known as a yield node. The CFG is fragmented by disconnecting each yield node from its direct successor, as the scheduler of the processor will possibly execute other threads between a yield node and its direct successor. The result of this splitting and fragmentation can be seen in Figure 4.

Now each thread of the processor receives its own copy of the fragmented CFG and the yield node is connected with the successor nodes of the next thread's CFG. It is not only connected to the copies of its own successor node, but with all successor nodes of the following thread. The connection to all successor nodes is clearly depicted in Figure 5.

The CFGs of the different threads are connected so that paths through the graph resemble the possible traces through all threads that the hardware scheduler can generate. But the new interleaved CFG of all threads has lost some of the original semantics of the program. In a multithreaded processor each suspended thread will resume execution at instruction following the last executed instruction before the suspension. In the multithreaded CFG every yield node is connected to *all* successor nodes of the next thread. The CFG does not enforce that a suspended thread is resumed at the correct position. For that reason an additional constraint is added to our CFG. For every pair of nodes (a, b) where a is a yield node and b is the direct successor, we require that they are executed equally often (i.e. constraint: $x_a = x_b$ where x_n is the execution count of node n).

In [1] a strict round-robin scheduler is assumed. For that reason the CFG of Thread n points to nodes in Thread n+1 (modulo number of Threads). We present the complete CFG for a two-threaded setup in Figure 5. Yield nodes in Thread 0 transfer control to Thread 1 and yield nodes in Thread 1 transfer control to Thread 0, just like a non-preemptive round-robin schedule would do.

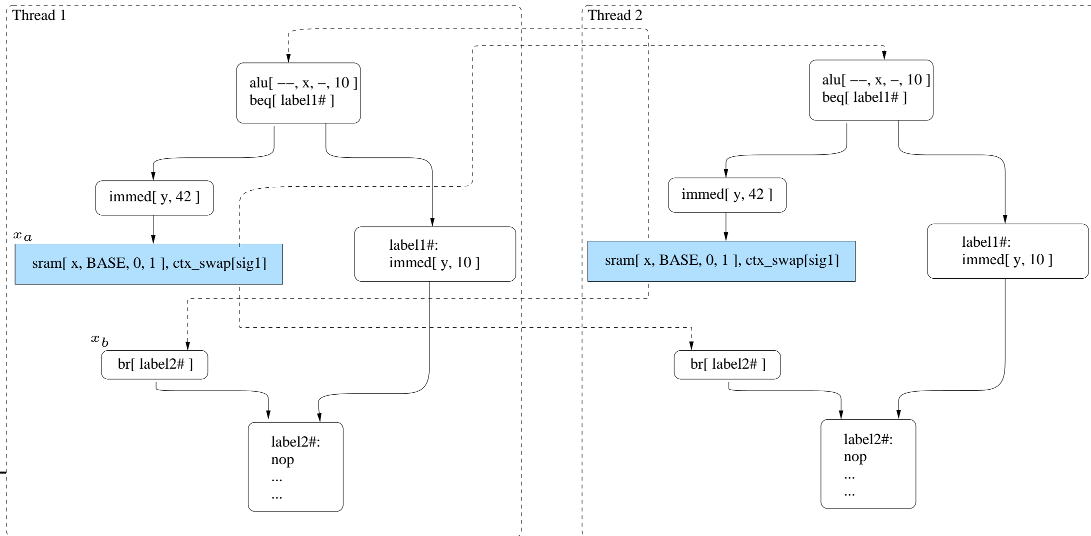


Figure 5. Example for two Threads

3.1.4 Multithreading Gain

The benefits of a multithreaded architecture are based on the fact that external transactions (e.g. memory accesses) no longer stall the processor, but are in parallel to another thread. Until now the CFG does not model this parallelism.

To model this parallelism we add negative weights (execution costs) to the newly added values. The negative weight equals the guaranteed parallelism of the threads. In Figure 6 the first (shaded) node models a memory access taking 30 cycles. The succeeding node of the next thread has an execution of 10 cycles. The 10 cycles are guaranteed to overlap with the 30 cycles of the memory access, for that reason we substract 10 cycles when taking the edge connecting the yield node with the next thread. The second memory access takes 30 cycles and is followed by an execution of 20 cycles. For this case, we can assume that 20 cycles are hidden.

The final version of the algorithm does not only consider the directly following execution node. As can be read in [1] the weight (execution cost) of the shortest (non-preemptible) path to the next preemption point is used.

3.1.5 Experiments

In Table 1 we present the analysis results using the method described by Crowley/Baer on our code. We present the results for μEv2 running at its natural speed of 1400 MHz and also an assumed μEv2 running at 233 MHz - which is the frequency that is assumed in [1].

Each column of the table shows the result for a different control function defined in the SDH standard. It is not necessary to discuss the details of these functions to evaluate the results. The first seven columns show results for the individual functions while the last column shows the results for the processing of all control functions at once.

For the different number of threads we assume to process multiple requests at once. So the results for two-threaded executions have twice the throughput, as these WCET for two parallel tasks are given. The results for eight threads assume that eight times the single-threaded payload is processed.

It is noteworthy that the analysis results for single-threaded execution are almost 100% precise. We found the estimates to be just slightly (5-22 cycles) higher than the measured worst-case execution time. This indicates that our model of the μEv2 is very good.

When looking at multithreaded execution, we can see that for the speed of 233 MHz the results do improve significantly, while the analysis for the 1400 MHz machine cannot be improved that much. Simulation runs show that the real performance scales much better than the predicted performance. For the 8-thread analysis our guaranteed WCET is 3 times higher than the simulated WCET. From this experiment we draw the conclusion that the methods must be adapted for higher μEspeeds.

Another interesting fact which can also be observed in [1] is that the method does not support more than two threads. The eight-threaded results are 4 times higher than the 2 threaded results.

3.2 Increasing the Multithreading Gain for Modern Architectures

As seen in the experiments, the method does improve the guaranteed performance when compared to a non-multithreading-aware solution. But the improvement is limited. The limitation stems from pessimism in the modeling of concurrency. Figure 6 shows an example (taken from [1]) of the pessimism. Two threads execute on a core and both access the memory. The analysis assumes that two memory accesses do not occur in parallel and that the memory access latency can only be hidden by the directly following block of code. This leads to a pessimism of 10 cycles as shown in Figure 6. A good measure for the effec-

	B3	C2	F2/F3	G1	H4	J1	N1	ALL
ME Speed: 233 MHz								
1 Thread	295 49%	356 67%	142 68%	266 56%	371 53%	502 47%	941 54%	2735 57%
2 Threads	478 62%	590 81%	269 72%	492 61%	601 67%	844 56%	1646 62%	4649 66%
ME Speed: 1400 MHz								
1 Thread	745 19%	716 33%	412 24%	626 24%	821 24%	1312 18%	2111 24%	6245 23%
2 Threads	1414 21%	1396 34%	777 25%	1210 25%	1535 26%	2571 18%	3905 26%	11965 25%
8 Threads	5650 21%	5653 34%	3106 25%	4856 24%	6033 26%	10281 18%	15626 26%	45295 25%

Table 1. Analysis of Original Code – Results show WCET in μE cycles and the associated μE-utilization

	B3	C2	F2/F3	G1	H4	J1	N1	ALL
ME Speed: 1400 MHz								
1 Thread	646 25%	650 37%	420 23%	586 25%	603 27%	912 26%	1234 41%	4986 32%
2 Threads	1120 26%	1105 43%	795 25%	1101 27%	1197 32%	1550 30%	2295 44%	9523 33%

Table 2. Analysis of Restructured Code – Results show WCET in μE cycles and the associated μE-utilization

tiveness of a multithreaded system is the system's utilization. In the given example the actual utilization is 60%: a total of 50 cycles response time split into 30 cycles execution time and 20 cycles idle time. The estimated utilization is 50%: a total of 60 cycles response time split into 30 cycles execution, 10 cycles idle time and 20 cycles stalling due to parallel memory accesses.

The fact that the analysis cannot handle parallel memory accesses from different threads is the reason why the analysis does not exhibit any extra multithreading-gain for more than two threads. The same reason accounts for the fact that the results of the 8-threaded setup are exactly four times higher than in the 2-threaded setup.

The pessimism of the analysis can be aggravated by control flow succeeding a context switch operation. If the thread executing in parallel to the memory request has variable control flow the shortest path is assumed to be taken when calculating the hidden latency (minimize hidden latency to be conservative) while the longest path can be used to calculate the worst-case-execution time. This does clearly lead to overestimations.

Given these observations we'll restructure the program's control flow in order to improve analyzability. To maximize the guaranteed hidden latency, we will restructure the code to show longer sequences of uninterrupted execution. We do this by concentrating prefetching and write-back actions at specific nodes in the control flow graph.

3.2.1 Rescheduling accesses

For arbitrary programs we want to reschedule memory operations in order to optimize the analytically guaranteed utilization.

To reschedule the memory accesses, for simplicity we introduce two new nodes at the beginning and at the end of the function to be optimized. The first node is used to read all possibly needed data from background memory into private high-speed memory (internal scratch-pad or registers), while the last node is used to write all possibly updated values back to memory.

Figure 9 shows an example of a control flow graph both before (top) and after (bottom) rescheduling accesses to background memory. Memory accesses are shown as shaded nodes.

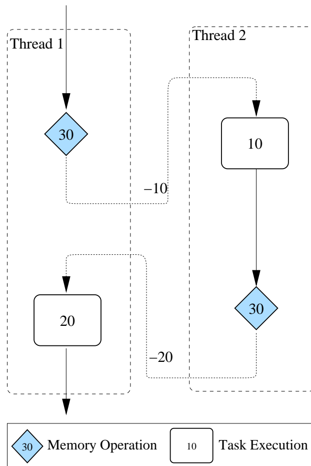
In the original code, accesses to variables stored in background memory are replaced by accesses to registers or private memory. The entry node contains a series of load operations to prefetch all needed variables into registers, while the exit node contains a series of store operations to write all used variables back to memory. We can safely assume that this transformation does not change the semantics of the code as all affected memory locations are used exclusively by the transformed function. Using that approach we implement a data cache in software.

The question of the code size to be optimized is still open. The optimal size depends on the available high-speed memory on the μEs and the exact nature of the code to be optimized.

As a granularity for the rescheduling of memory operations we chose not to exceed a single iteration of our main control loop. In Figure 8 one can see the structured CFG of our control application. The main loop is clearly visible and some distinct sub blocks that are part of the main loop. The blocks labeled "Process J1", "Process B3" etc. implement the control part of finite state machines. One can also see the basic blocks that contain a memory access, they are drawn shaded.

3.2.2 Rescheduling of conditional accesses

When concentrating the access to memory in a single or two central points in the program, it is usually not avoidable to transfer data which will not be needed during the



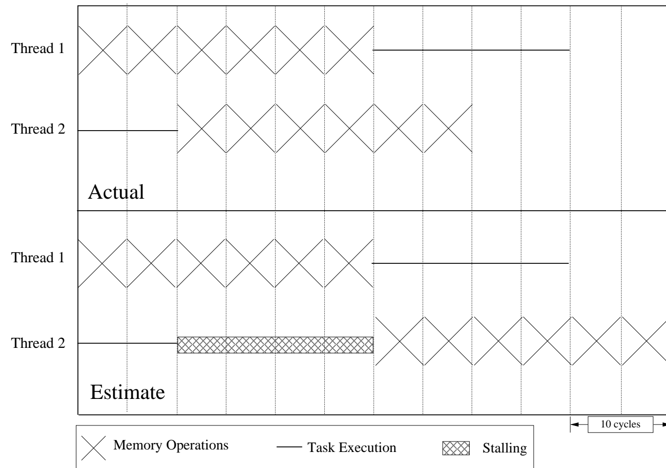


Figure 6. Modeling concurrency with yield edges

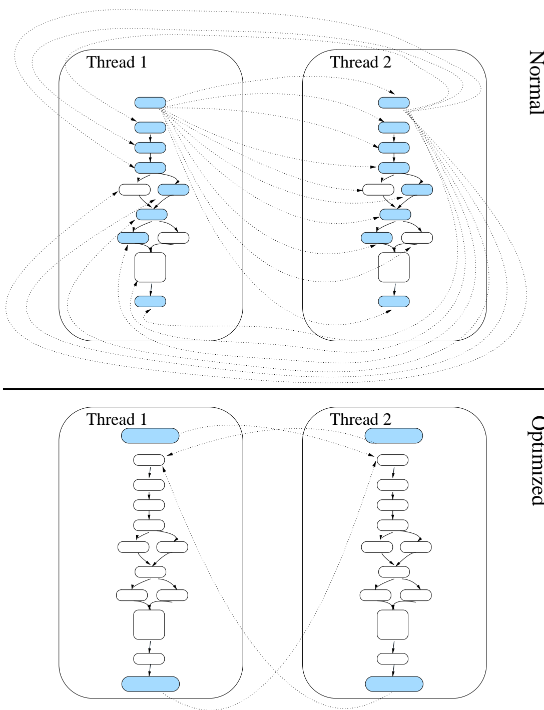


Figure 7. Interleaved CFG before and after Rescheduling of Memory Operations – (For illustrative reasons, the interleaving is only done for the first nodes in the unoptimized CFGs)

calculation. This fact seriously limits the scalability of this approach: e.g. a program which accesses large arrays with an unknown array index is likely not to be accessible using this approach as the range of possibly used memory locations is to big to prefetch.

If, however, the range is of small size prefetching memory can be a viable option even if unnecessary memory transfers are initiated. Developing a method to automatically decide when to apply this strategy is an open question - for our target application we found the introduced overhead to be reasonably small as can be seen in the analysis results.

3.2.3 Experiments

In Table 2 we show the results for the optimized code running on a 1400 MHz μEv2. Comparing these results with the results of the original code, we can see considerable improvement for the single-threaded case. This improvement stems from the fact that several memory accesses were concentrated into a single node and are – where applicable – transformed into a single burst-access (which results into back-to-back transfers of the requested data). This method leads to a better system performance and even more important a more precise estimation of the memory access times. The better estimation of the memory access times is already achieved by rescheduling the accesses - we do not need to transform the accesses in a burst access to get a much better estimation of the memory accesses. The main factor is that in a series of memory accesses that are executed back-to-back we can be sure that not all of them will experience the worst-case-state of the system [3].

The multithreading gain – which can be easily seen when comparing the utilizations – is still limited, but bigger than in the original approach. Refer to Table 1 to see that for the μEv2 running at 1400 MHz most results experienced virtually no improvement by the multithreading analysis.

Here, again, the guaranteed utilization is of key interest

and can be seen to be much better than before.

3.3 Bigger Workload Units

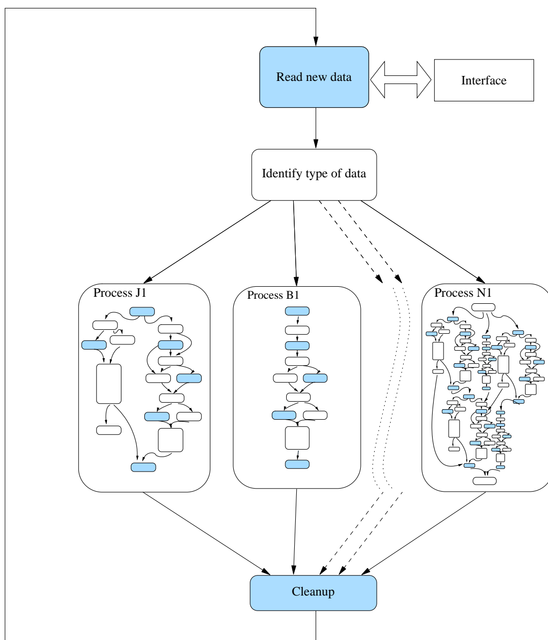


Figure 8. Block Diagram of Control Flow

To achieve more benefit with the presented method, we'll increase the granularity even more by extending it over multiple iterations of the main loop. Introducing a very small buffering stage which allows to read multiple input data from the line interface at once, we can guarantee to have one token of input data for each implemented control function. As we have eight different control functions within the main loop, we increase the number of read inputs to eight. Additionally we do no longer jump to a single function to process the incoming data, instead we serially process all control functions in a defined order and integrate the fetch/write-back nodes of all eight iterations into a single shared one. We can easily do this as the used μEs offer plenty of scratch-pad/register storage[1].

3.3.1 Experiments

Table 3 gives the results for the processing of eight input stimuli at once. As one can see, the guaranteed utilization bumps up considerably when using a multithreading-aware analysis for this system. The reason is simple: the latency of the memory access and the latency of the execution on the processor have the same magnitude. Thus a lot of the memory latency is guaranteed to be hidden by a second thread which already has its data ready in internal μE registers.

To show the improvement of this method over previously suggested analysis, the reader should compare the results for the 1400 MHz μE running the "ALL" benchmark in Table 1, Table 2 and Table 3. The same code was previously guaranteed to execute in 11,956 cycles (25% utilization) and now is guaranteed to execute in 9523 (33% utilization) or even 3487 cycles (84% utilization) if the main loop can be unrolled.

	ALL
ME Speed: 1400 MHz	
1 Thread	2231 66%
2 Threads	3487 84%

Table 3. Results of Batch Approach

4 Future Work

The results presented in Table 2 are too pessimistic for our system. We'd like to implement an improved analysis for memory access times into this work to show that the optimized code already provides a considerable multithreading gain compared to the original code.

The original code has a distribution of memory accesses that limit the analytical multithreading gain in a durable way, as the left-over blocks of code are too small to hide considerable memory latency.

In this paper we presented a work where we manually optimized our code to improve the analysis. The optimizing transformation itself is easily automated and integrated into an optimizing compiler for multithreaded architectures. Key questions are, however, how we can detect a situation where we can apply our optimization.

5 Conclusions

We presented a method to optimize the guaranteed processor utilization in a multithreaded architecture.

With state machines typically found in control applications we identified an important class of programs which is accessible to this approach.

The guaranteed processor utilization (and thus performance) for the implemented application was increased by a factor of three.

References

[1] Patrick Crowley, Jean-Loup Baer, "Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors", Proc. HPCA-9 WS on Network Processors, 2003

[2] Yau-Tsun Steven Li, Sharad Malik, Andrew Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches", Proc. IEEE Real Time Systems Symposium, 1996

[3] Simon Schliecker, Matthias Ivers, Jan Staschulat and Rolf Ernst, "A Framework for the Busy Time Calculation of Multiple Correlated Events", In 6th Intl.

[1] If the storage would not suffice to store all necessary data, we can easily decrease the granularity of optimized code-fragments

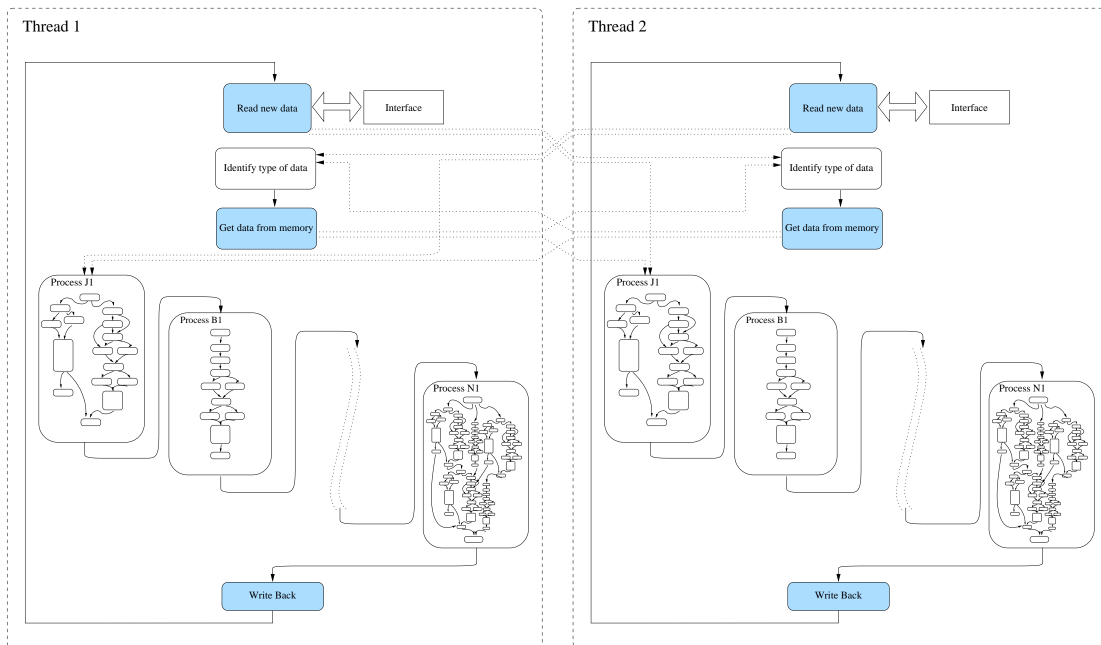


Figure 9. Batch Processing on two threads - memory access are in shaded nodes only

Workshop on WCET Analysis, Dresden, Germany, July 2006

[4] ITU-T Recommendation, G.707: Network Node Interface for the synchronous digital hierarchy (SDH), Dec. 2003.

A Context Cache Replacement Algorithm for Pfair Scheduling

Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki
Graduate School of Science and Technology
Keio University, Yokohama, Japan
{funaoka,shinpei,yamasaki}@ny.ics.keio.ac.jp

Abstract

Pfair scheduling is the only known optimal way for scheduling recurrent real-time tasks on multiprocessors. However, it causes significant overheads compared with the traditional approaches due to frequent task preemptions and migrations. Our approach makes the effective use of the context cache which is the exclusive on-chip memories of the hardware contexts to reduce overheads. In this paper, we propose a context cache replacement algorithm called Farthest Weight[1/2], which is more effective than traditional approaches. Simulation results show that the context cache is effective to reduce these overheads, and the proposed algorithm reduces overheads half as much as the case which is done by software.

1. Introduction

Processor performance has improved remarkably with the advancement of technologies. However, ascending heat and electricity consumption caused by these improvements become problematic. Accordingly, Simultaneous Multithreading [18] (SMT) and Chip Multiprocessing [15] (CMP) have been watched with keen interest because of their thread level parallelism. Furthermore, it is important for embedded systems to concern not only performance but also electricity consumption. These processors are attractive for embedded systems. Most embedded systems have the tasks which have their time constraints such as the robot control or image processing.

Real-Time scheduling. There are two approaches to schedule tasks when systems have multiple-contexts. In this paper, we call execution units *contexts* likewise processors, threads of SMT, or cores of CMP. In partitioning, all the jobs generated by a task are always scheduled on the same context. In global scheduling, on the other hand, tasks are inserted to the global queue, and the context to be scheduled is decided on time.

In partitioning, uniprocessor scheduling can be applied to per-processor scheduling. Liu and Layland [12] showed that Earliest Deadline First (EDF) is optimal on a uniprocessor. However, the assignment of tasks to processors is a bin-packing problem which is NP-hard in the strong sense. Lopez et al. [13] showed that no partitioned approach can guarantee schedulability with total utilization (or weight) over $(M+1)/2$ on M processors.

In global scheduling, Dhall and Liu [9] show that traditional uniprocessor algorithms do not work well because of the Dhall's effect. Srinvasan and Baruah [17] proposed EDF-US$[M/(2M-1)]$ to avoid the Dhall's effect. Baker [6] showed that EDF-US$[x]$ guarantees worst-case schedulable utilization of $(M+1)/2$. Andersson et al. [5] showed that no static priority multiprocessor scheduling algorithm can guarantee the utilization higher than $M/2$.

Proportionate-fair (Pfair) scheduling proposed by Baruah et al. [7] optimally solves the problem of scheduling periodic tasks on multiprocessor systems. PF [7], PD [8], and PD^2 [3] are the optimal pfair scheduling algorithms. EPDF [4] is optimal on one or two processors. However, Pfair scheduling causes overheads due to frequent task preemptions and migrations when this scheme is applied to the process scheduling. Srinvansan et al. [16] showed that PD^2 is competitive with EDF-FF (First Fit i.e. a partitioning heuristic) even if the overheads are considered. Moreover, they presented that Pfair scheduling provides many additional benefits.

The problem. Proofs of theoretical optimality of real-time scheduling algorithms are mostly constructed on some assumptions. One of these assumptions in some cases is that there are no overheads. However, the overheads need to be considered on practical systems. The overheads are absorbed the worst case execution time of tasks to guarantee schedulability. If the overheads of the system are significantly large, its performance is awfully degraded. One of the origin of performance degradation comes from the resource competitiveness. We propose an effective use of context cache [19] to reduce the overhead of context switching.

Contributions. The remainder of this paper is organized as follows. In Section 2, we give an overview of the related work. The background on Pfair scheduling is provided in Section 3. In Section 4, the context cache and its mechanism are described. In Section 5, we propose a context cache replacement algorithm to reduce overheads. In Section 6, we present the experimental results. Finally, we present conclusions and future work in Section 7.

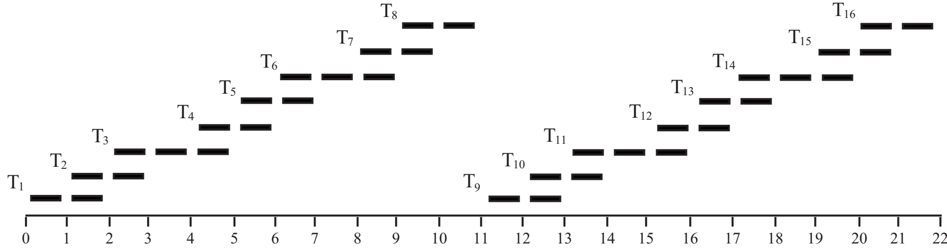


Figure 1. Windows of a task T which has $wt(T) = 8/11$.

2. Related Work

Anderson and Srinivasan [3] presented ERfair scheduling which is the work conserving version of Pfair scheduling. ERfair scheduling differs from Pfair scheduling in the sense that tasks can be executed early. In some cases, it is possible to execute tasks without task preemptions or migrations if a tasks can be executed early. However, if the system utilization is high, the tasks can be hardly executed early. Consequently, per task preemption or migration overheads have to be reduced.

Moir and Ramamurthy [14] showed the existence of Pfair schedule for any feasible task sets without migration. Although it intends to schedule the tasks which have resource restrictions, it can be possible to reduce the context overheads because to fix the tasks on one processor prevents the cache misses and TLB misses and so on.

Anderson and Calandrino [2] proposed a spread-cognizant scheduling method to decrease the spreads in Pfair scheduling and global EDF. The overheads caused by the cache misses can be reduced by this method.

Many algorithms of memory and disk cache replacement are performed [1, 10, 11]. These cache replacement algorithms are on the assumption that cache accesses have locality. Least Recently Used (LRU) and Least Frequently Used (LFU) are the typical cache replacement algorithms.

To our best knowledge, no context cache replacement algorithms are presented. The notion of context cache is different from typical cache. Consequently, new replacement algorithms for context cache are required. The details of context cache is shown in Section 4.

3. Pfair Scheduling

To show the overview of Pfair schedule, we give a real-time system model. A real-time system is modeled as the taskset τ which is a set of periodic tasks to be executed on M contexts. A task T ($\in \tau$) is characterized by two parameters, its worst case execution time $T.e$ and its period $T.p$. A task T requires $T.e$ times of contexts for execution at every $T.p$ interval. The ratio $T.e/T.p$, denoted $wt(T)$, is called the weight (or utilization) of task T, where $0 < wt(T) \leq 1$. A task T is called light if $wt(T) < 1/2$. The worst case execution time $T.e$ of light task T is smaller than a *not* execution time $T.p - T.e$. Otherwise, a task T is called heavy. $\sum_{\forall T \in \tau} wt(T)$ is the weight of the taskset τ.

In Pfair scheduling, context time is allocated in discrete quanta. The time interval $[t, t+1)$ is called a slot t. The time t means the start time of the slot t. Tasks can be executed on every contexts in one slot, however, simultaneous executions of a task on different contexts in one slot are not permitted. The sequence of allocation over slots defines a schedule S.

$$S : \tau \times N \rightarrow \{0, 1\} \tag{1}$$

where τ is a task set and N is the set of nonnegative integers. $S(T, t) = 1$ iff a task T is scheduled in slot t. $\sum_{\forall T \in \tau} S(T, t) \leq M$ holds for all t. A notion of lag which is the difference between the ideal allocation and actual allocation is defined as the following.

$$lag(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u) \tag{2}$$

A schedule is defined to be Pfair iff

$$(\forall T, t :: -1 < lag(T, t) < 1). \tag{3}$$

Informally, the allocation error must be less than one. To satisfy Equation 3, tasks are divided into subtasks with WCET $= 1$, denoted T_i, where $i \geq 1$. Each subtask has its pseudo-release time and pseudo-deadline defined as the following.

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \quad d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{4}$$

A subtask T_i has to be scheduled between $r(T_i)$ and $d(T_i)$ called *window* of T_i. Windows of a task T which has $wt(T) = 8/11$ are shown in Figure 1.

The optimal algorithms, PF, PD, and PD^2, give higher priority to subtasks with earlier deadlines. However, ties are broken with different manners. In these algorithms, PD^2 is most efficient. A valid schedule S exists for a task system τ on M processors iff

$$\sum_{\forall T \in \tau} wt(T) \leq M. \tag{5}$$

3.1. Scheduling Overheads

In Pfair scheduling, schedulers need to schedule all contexts at each slot. Consequently, overheads of Pfair scheduling are higher than traditional algorithms when

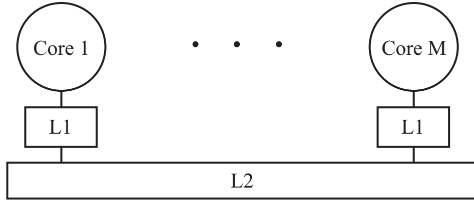


Figure 2. An example of CMP architecture

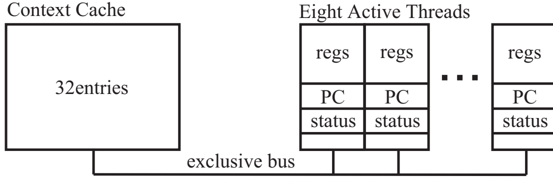


Figure 3. The context cache on RMT Processor

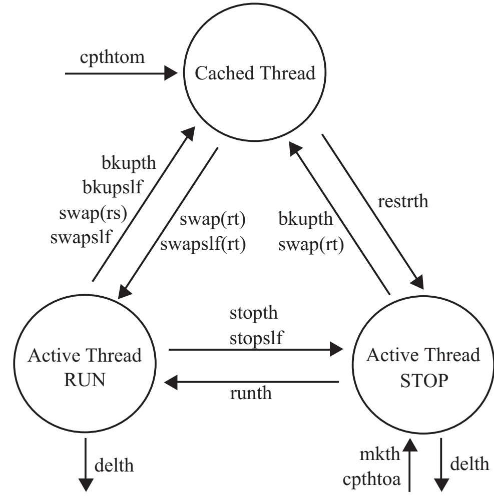


Figure 4. The state transition chart of threads of RMT Processor

Pfair scheduling is applied to process scheduling. Srinivasan et al. [16] showed the formulas of these overheads. Furthermore, all the slots among contexts are assumed to be synchronized, in Pfair scheduling. When the overheads are large, the optimality of Pfair scheduling is dismissed.

There are two operations to schedule processes. First, a scheduler must decide which task to be scheduled. Second, a scheduler must swap hardware contexts. These overheads reduce the time of task executions.In this section, we focus on the overheads of context switches and migrations. These overheads are divided into two origins. First, one of these overheads is context switching itself. To swap hardware context, many memory access instructions which have long latencies are required. Second, task preemptions and migrations cause collisions of hardware resources, like memory cache, TLB, and so on. Figure 2 shows an example of CMP architecture. When task migrations occur, independent resources among contexts (L1 cache in Figure 2) are completely lost. When task preemptions occur, shared resources among contexts (L2 cache in Figure 2) are also competitive.

4. Context Cache

The context cache [19] is an exclusive on-chip memory for saving hardware contexts. It can save and restore hardware contexts by software. On Responsive Multithreaded (RMT) Processor [19], a context switch takes only 4 clocks by the context cache while it takes 590 clocks by software. In our best knowledge, RMT Processor is the only processor in which context cache is implemented. The context cache is not architecture oriented. Therefore, it can be available on many systems.

RMT Processor is the prioritized 8-way SMT processor, which has 8 active threads and 32 cache threads. The context cache of RMT Processor is shown in Figure 3. The registers of active threads are connected to context cache by exclusive bus. The state transition chart of threads of RMT Processor is shown in Figure 4. The state is changed by software instructions. In this paper, we call executable thread as active thread.

An example of context switch with the context cache is shown in Figure 5. At line 19 in this figure, if there is no empty entry of context cache, the evicted entries must be selected. Since a hardware context has a lot of information to be saved, the area size of context cache per 1 entry becomes larger than memory cache. It is difficult to implement many entries to processors. Consequently, the decision which entry should be evicted is important.

There is no need to leave the entry of the task which is executed in context cache. Therefore, it is efficient for context cache to swap active threads and cached threads. This characteristic is different from memory cache and disk cache. It is impossible to discuss the context cache as the same rank with the memory cache and disk cache.

To compare the traditional context switch, the number of context switch by software (i.e. by the *load* and *store* instructions) denoted $n(\text{soft_switch})$, is defined

$$n(\text{soft_switch}) = \frac{n(\text{soft_load}) + n(\text{soft_save})}{2}, \quad (6)$$

where $n(\text{soft_load})$ is the number of restoring hardware context by the *load* instructions and $n(\text{soft_save})$ is the number of saving hardware context by the *store* instructions.

5. A Context Cache Replacement Algorithm

If all tasks are put in context cache, we can be always given the benefit of the context cache. Otherwise, the task assignment to context cache is important. An important thing to remember is that the complexity of the context cache replacement algorithms must be lower than context switch itself.

Algorithm: ContextSwitch

1: Let $\mathcal{A} = \{A_1, \ldots, A_M\}$ be the active threads
2: Let $\mathcal{C} = \{C_1, \ldots\}$ be the cached threads
3: Let P_i be the prev context on A_i
4: Let N_i be the next context on A_i
5:
6: **for all** i such that $1 \leq i \leq M$ **do**
7:   **if** $P_i \neq N_i$ **then**
8:     **if** $N_i \in \mathcal{C}$ **then**
9:       *swap P_i and N_i*
10:     **else if** vacant entries of $\mathcal{C}$ exist **then**
11:       **if** P_i exists **then**
12:         *copy P_i to context cache*
13:       **end if**
14:       **if** N_i exists **then**
15:         restore N_i from memory
16:       **end if**
17:     **else**
18:       **if** P_i exists **then**
19:         **if** P_i goes to $\mathcal{C}$ **then**
20:           *swap P_i and evicted entry*
21:         **end if**
22:         save current context to memory
23:       **end if**
24:       **if** N_i exists **then**
25:         restore N_i from memory
26:       **end if**
27:     **end if**
28:   **end if**
29: **end for**

Figure 5. A concept example of context switches on RMT Processor

5.1. Farthest Weight[1/2]

We propose Farthest Weight[1/2] (FW) context cache replacement algorithm. FW evicts the tasks, in the context cache, which have larger $F(T)$ in the following.

$$F(T) = |wt(T) - 1/2| \tag{7}$$

The most difference between FW and the traditional replacement algorithms, such as LRU and LFU, is that FW takes notice of the task parameter while LRU and LFU takes notice of the cache entry parameters.

The behavior of FW, LRU, and LFU with scheduling PD^2 are shown in Figure 6. The number of contexts, context cache entries, and tasks are 1, 2, and 4, respectively. The weights of the tasks are $1/3$, $1/4$, $1/8$, and $1/10$. The timing of the context switch by software are shown in the lower part of the cached threads as Save(S) and Load(L). Since, in LFU, ties are broken arbitrary if some entries have the same frequency, the worst case is shown in this figure. FW can assign the high priority to the tasks which is frequently executed. In this example, the number of software context switches of FW, LRU, and LFU are 4.5, 5.5, and 6.5, respectively. The reason why LFU can not effectively deal with this problem is that the parameter compared by LFU comes from the value related

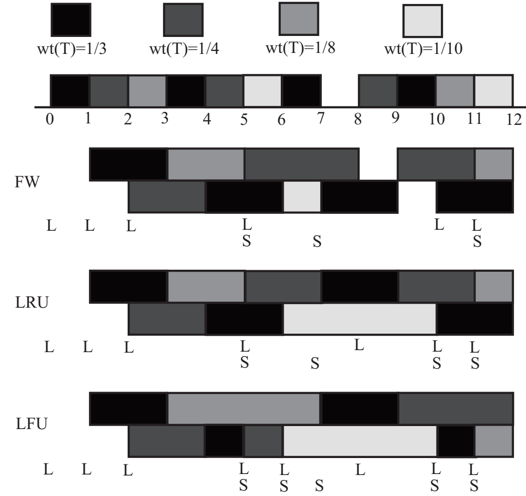


Figure 6. A comparison among FW, LRU, and LFU

to the cache entries (not the tasks) because the cache entries are changed by the frequent context swithing. If the frequency considered by LFU gives to the tasks, we can not dual with aperiodic tasks effectively.

5.2. The Effectiveness of FW

It is difficult to calculate the number of context switches on-line. Therefore, a situation which makes the number of context switches large needs to be supposed. The worst case number of context switches in Pfair scheduling (WCNCSP) is defined as follows.

Definition 1 (WCNCSP) *The worst case number of context switches in Pfair scheduling (WCNCSP) is the largest number of context switches under the all schedule sequences in the time interval $[0, lcm(T.p))$ with supposing that no context switch occurs when the subtasks of one task are assigned to consecutive slots.*

In Pfair scheduling, the scheduling decisions are made by priorities of subtasks. A task T which has higher priority blocks the other tasks which has lower priority than T. We refer this type of blocking as *"schedule restriction"*.

Lemma 1 *$C(T) \leq C'(T)$, where $C(T)$ is the WCNCSP of the task T when the task T suffers schedule restrictions by the other tasks, and $C'(T)$ is the WCNCSP of the task T when the task T does not suffer them.*
Proof *$\mathcal{S} \subseteq \mathcal{S}'$, where the possible schedule set under schedule restrictions is $\mathcal{S}$, and the possible schedule set under norestriction assumptions is $\mathcal{S}'$.* ■

Lemma 1 shows that we need not to consider the other tasks when we estimate the WCNCSP.

Theorem 1 *In Pfair scheduling, the WCNCSP of the task T is maximized when $wt(T) = 1/2$, where $wt(T)$ is the*

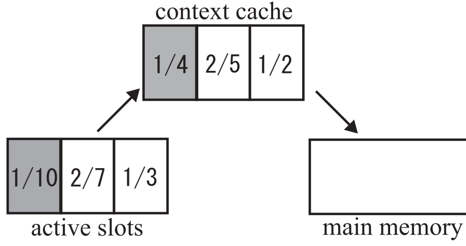


Figure 7. Strict replacement method

weight of the task T.

Proof *Let n be a non-negative integer. Let the task T not suffer schedule restriction from the other tasks.*

When the task T is light, $T.e \leq (T.p - T.e)$. Therefore, there exists a schedule $\mathcal{S}$ which does not schedule subtasks in consecutive slots. The WCNCSP is $2T.e$ in time interval $(n \cdot T.p, (n+1)T.p]$. The WCNCSP per 1 slot is $2T.e/T.p = 2wt(T)$.

When the task T is heavy, $T.e > (T.p - T.e)$. Therefore, there exists a schedule $\mathcal{S}$ which does not make consecutive empty slots by the task. The WCNCSP is $2(T.p - T.e)$ in time interval $(n \cdot T.p, (n+1)T.p]$. The WCNCSP per 1 slot is $2(T.p - T.e)/T.p = 2(1 - wt(T))$. Consequently, the WCNCSP of the task T is maximized when $wt(T) = 1/2$. ■

FW evicts the tasks, in the context cache, which is not preempted frequently.

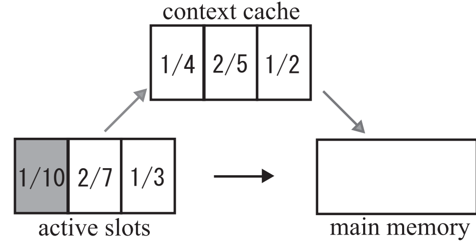
5.3. Replacement Methods

There are two replacement methods to be considered as shown in Figure 7 and 8. The number in the active slots and the context cache is the weight of the task in the entry.

In the first method, the context currently executed is always housed to the cache as shown in Figure 7. This method is called *strict*. In Figure 7, when the task which has the weight $1/10$ in the active slots finishes the execution of its subtask, it is saved to the context cache. The task which has the weight $1/4$ (i.e. the largest $F(T)$ in the context cache) is evicted to the main memory.

In the second method, on the other hand, there exist the possibilities that the context currently executed is housed to memory as shown in Figure 8. This method is called *lazy*. In Figure 8, when the task which has the weight $1/10$ in the active slots finishes the execution of its subtask, it is saved to the main memory because there is no task which has larger $F(T)$ in the context cache than the task. On the other hand, when the task which has the weight $1/3$ finishes the execution of its subtask, it is saved to the context cache. The task which has the weight $1/4$ is evicted to the main memory because $|1/3 - 1/2| < |1/4 - 1/2|$.

In the strict method, $F(T)$ of tasks in the context cache are compared to decide which entry is evicted. On the other hand, in lazy method, $F(T)$ of tasks in the context



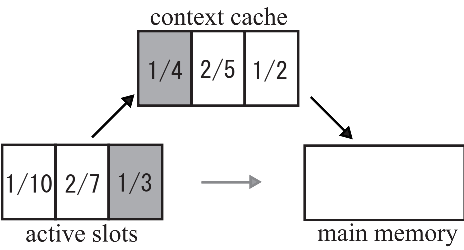


Figure 8. Lazy replacement method

cache *and* the context currently executed are compared. The difference between FW and the traditional algorithms such as LRU or LFU is the place of the parameter which decides the evicted entry. While LRU and LFU compare the parameters of the cache entries, FW compares the parameters of the tasks. The lazy method needs the parameters of tasks because the tasks in the active slots have no value about the cache entries.

In the traditional cache system, strict method is widely used because of its simplicity and efficiency. This is simple and efficient when the cache access has locality. However, in Pfair scheduling, the executions of tasks are dispersed. Consequently, the possibility that the entry of the task which is now finished the execution is low. The strict method make the context house to the context cache. Therefore, in Pfair scheduling, lazy method is effective.

The behavior of strict method and lazy method, shown in Figure 9. The number of the active threads, the context cache entries, and the tasks are 1, 1, and 3, respectively. The weights of the tasks are 1/3, 1/4, and 1/8, respectively. The scheduling and context cache replacement algorithms are PD^2 and FW, respectively. The timing of the context switch by software are shown in the lower part of the cached threads as Save(S) and Load(L). In this example, the number of software context switch of Strict and Lazy are 7.5 and 5.5, respectively.

6. Experimental Results

In this section, we show the effectiveness of the context cache and the proposed algorithm FW. FW and the traditional cache replacement algorithms such as LRU and LFU are compared.

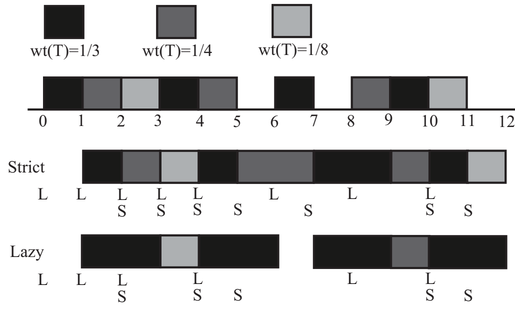


Figure 9. A comparison between Strict and Lazy

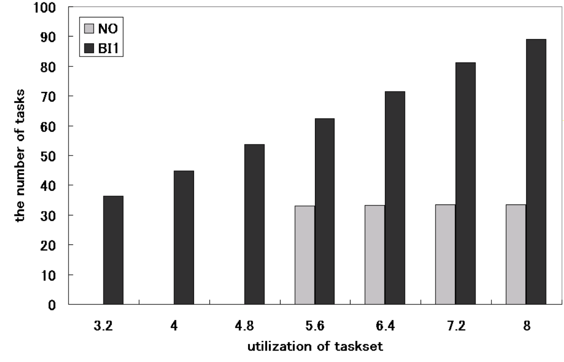


Figure 10. The number of tasks

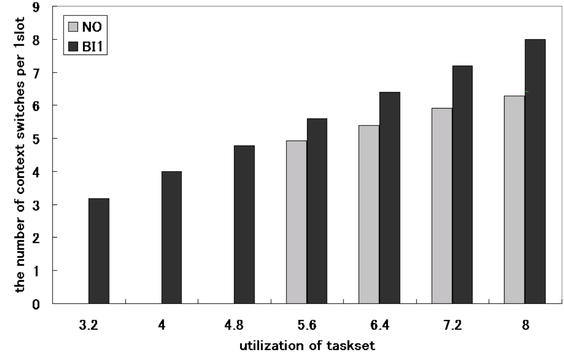


Figure 11. The number of context switches per 1 slot

Table 1. Simulation workloads

	distribution	M	N cache entries	slot length
NO-1	normal	8	32	1ms
NO-01	normal	8	32	0.1ms
BI1-1	bimodal	8	32	1ms
BI1-01	bimodal	8	32	0.1ms

6.1. Experimental Setup

The metrics of experiments are the cache miss ratio and the overhead ratio defined as follows.

$$\text{miss ratio} = \frac{n(\text{soft_switch})}{n(\text{switch})}, \tag{8}$$

where $n(\text{soft_switch})$ and $n(\text{switch})$ are the number of context switch by software (i.e. by the *load* and *store* instructions), and all the context switches (i.e. with cache and without cache), respectively.

$$\text{overhead ratio} = \frac{T(\text{switch})}{L(\text{slot})}, \tag{9}$$

where $T(\text{switch})$ and $L(\text{slot})$ are the duration of a context switch and a slot length, respectively. The overhead ratio changes large by the slot length.

The supposed environment is as follows. To our best knowledge, the processor which is implemented the context cache is only RMT Processor. The simulation parameters are decided by supposing RMT Processor. The frequency of the processor is assumed 100MHz. There are 8 contexts and 32 context cache entries. The number of clocks needed to switch a context is 590 by software and 4 by hardware. The time needed to switch context does not include the operations of task queue.

The simulation uses the workloads shown in Table 1. The simulation is conducted through each workload given tasks which have different weights. The results are shown only when the number of tasks is larger than the number of the context cache entries. The algorithms compared are LRU, LFU, FW, and FW-Lazy. The results which do not use the context cache is shown as Software. The results which are assumed that the number of context cache entries are infinity is shown as Hardware.

The tasksets are generated as follows. The arrival times of all tasks are time 0. The period of tasks are selected in integer [1,100]. The weights of tasks are normal distribution or bimodal distribution. The weight of the taskset exceeds the target weight, then the task is discarded and a new task is generated. When the generation fails 100 times, the last task is generated. When the hyperperiod of the tasksets overflow 24bit, the tasksets are rejected because of time constraints. If the number of tasks are smaller than the number of context cache entries, a new taskset is generated. By these operations, 100 tasksets are generated for each workload.

The tasksets whose weight have normal distribution are generated. The weight of tasks is selected $0 < wt(T) \leq \min(1, V)$, where V is the target weight. The tasksets whose weight have bimodal distribution is generated as follows. The trial whose success ratio is 0.1 is done 100 times, and the number of success is divided 100.

The tasks are executed until the hyperperiod of the tasksets. The hyperperiod of the tasksets is $\text{lcm}(\forall T.p)$. The scheduling algorithm is PD^2. When a task is executed consecutive slots, the task is assigned to the same context. The average of results are calculated for all 100 tasksets.

6.2. Experimental Environments

Since the number of tasks, context switches per 1 slot, and cache miss ratio are the same between NO-10 and

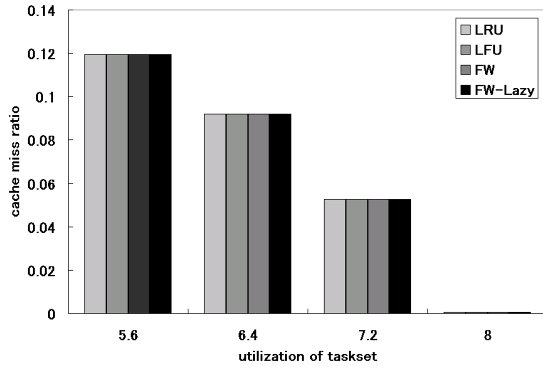


Figure 12. The cache miss ratio on workload NO

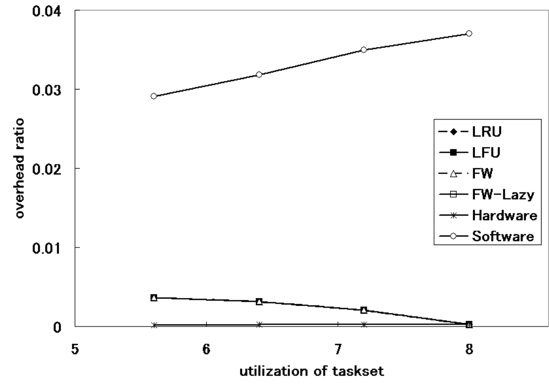


Figure 14. The overhead ratio on work-load NO-10

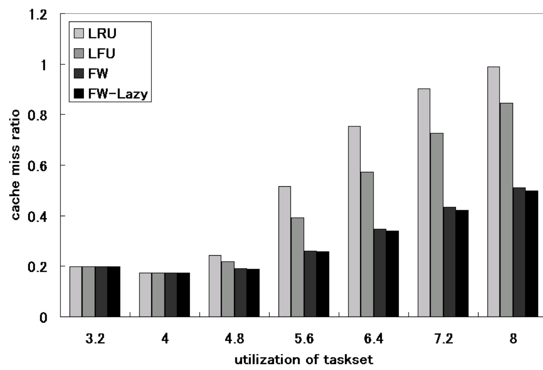


Figure 13. The cache miss ratio on workload BI1

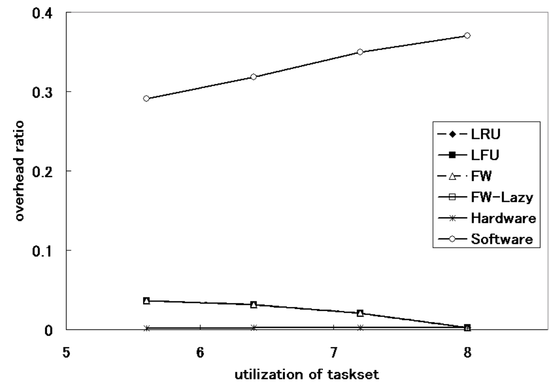


Figure 15. The overhead ratio on work-load NO-01

NO-01, BI1-10 and BI1-01, each result is shown as NO and BI1.

The number of tasks and context switches per 1slot is shown in Figure 10 and 11. The number of tasks is larger and larger while the weight of taskset is large in workload BI1. The other way, in workload NO, the number of tasks hardly changes. The number of context switches per 1 slot is larger and larger while the weight of taskset is large. However, the increase in BI1 is larger than in NO. When the weight of taskset is 8, context switches occur in almost all slots.

6.3. Cache Miss Ratio

The cache miss ratio on workload NO is shown in Figure 12. The miss ratio decreases while the weight of taskset increases. Since the empty slot increases, the consecutive execution increases. The difference among context cache replacement algorithms are none.

The cache miss ratio on workload BI1 is shown in Figure 13. The cache miss ratio increases while the weight of taskset increases. When the weight of taskset is 8, the cache miss ratio of LRU is almost 0.98. In BI1, the executions of tasks are dispersed. The cache miss ratio of FW and FW-Lazy is about 0.51 and 0.50, respectively.

6.4. Overhead Ratio

The overhead ratio of NO-10 and NO-01 are shown in Figure 14 and 15, respectively. Most tasks are put in context cache since the number of tasks are smaller than that of BI1. The decrease of overhead ratio comes from the fact that the empty slot is smaller and smaller while the weight of taskset is large.

The overhead ratio of BI1-10 and BI1-01 are shown in Figure 16 and 17, respectively. The number of context switches per 1 slot is larger than that of NO. Consequently, the overhead under BI1 is larger than that of NO. The results of LRU are closely related to Software. FW can decrease the overhead ratio larger than the other algorithms.

The implementations FW and FW-Lazy is almost same. Therefore, FW-Lazy is the most efficient algorithm compared with LRU, LFU, and FW.

7. Concluding Remarks

In this paper, we propose an effective use of the context cache to reduce the overheads of Pfair scheduling on multi-context environments. Additionally, we propose a new context cache replacement algorithm called Farthest Weight[1/2] (FW). FW evicts the task which is not fre-

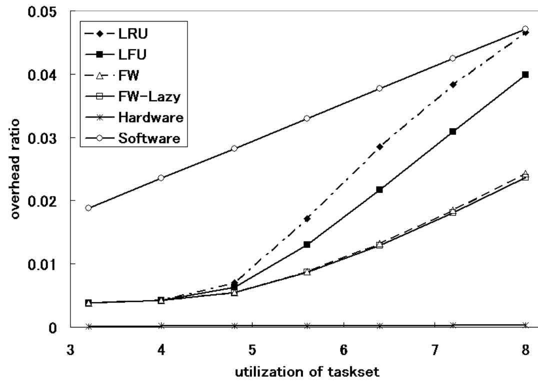


Figure 16. The overhead ratio on work-load BI1-10

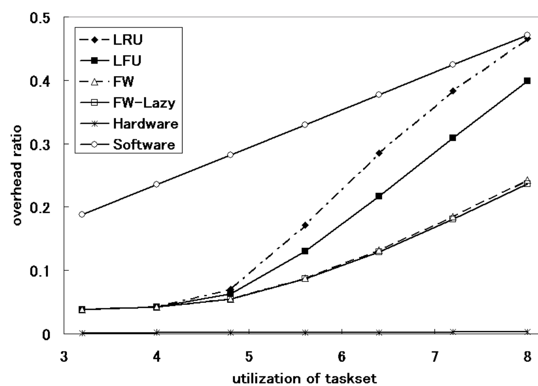


Figure 17. The overhead ratio on work-load BI1-01

quentry preempted. Experimental results shows that FW is effective against to the traditional cache replacement algorithms such as LRU and LFU.

We have some future work. First, we want to combine our approaches with the other works such as spread-cognizant scheduling. Our algorithm is only based on the task utilization. Second, the scheduling algorithm is considered to decide a replacement entry. For example, in PD^2, scheduling decisions are constructed by some factors such as pseudo-deadline, b-bit, and group deadline. Finally, we implement our algorithm to practical systems.

Acknowledgement

This research is supported by CREST, JST.

References

[1] J. Alghazo, A. Akaaboune, and N. Botros. SF-LRU Cache Replacement Algorithms. In *Proc of the Records of the 2004 International Workshop on Memory Technology, Design and Testing*, pages 19–24, Aug. 2004.

[2] J. H. Anderson and J. M. Calandrino. Parallel Real-Time Task Scheduling on Multicore Platforms. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 89–100, Dec. 2006.

[3] J. H. Anderson and A. Srinivasan. Early-Release Fair Scheduling. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[4] J. H. Anderson and A. Srinvasan. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 1996.

[5] B. Andersson, S. Baruah, and J. Jonsson. Static-priority Scheduling on Multiprocessors. In *Proc. of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202, Dec. 2001.

[6] T. P. Baker. An Analysis of EDF Schedulability on a Multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768, Aug. 2005.

[7] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, pages 600–625, 1996.

[8] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proc. of the 9th International Parallel Processing Symposium*, pages 25–28, Apr. 1995.

[9] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, pages 127–140, 1978.

[10] S. Jiang and X. Zhang. Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance. *IEEE Transactions on Computers*, 54(8):939–952, Aug. 2005.

[11] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(122):1352–1361, Dec. 2001.

[12] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, pages 46–61, Jan. 1973.

[13] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 25–33, 2000.

[14] M. Moir and S. Ramamurthy. Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 294–303, Dec. 1999.

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Multiprocessor. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Oct. 1996.

[16] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah. The Case for Fair Multiprocessor Scheduling. In *Proc. of the International Parallel and Distributed Processing Symposium*, page 10, Apr. 2003.

[17] A. Srinvasan and S. Baruah. Deadline-based Scheduling of Periodic Task Systems on Multiprocessors. *Information Processing Letters*, 84:93–98, May 2002.

[18] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[19] N. Yamasaki. Responsive Multithreaded Processor for Distributed Real-Time Systems. *Journal of Robotics and Mechatronics*, 17(2):130–141, Apr. 2005.

Exact Cache Characterization by Experimental Parameter Extraction

Tobias John, Robert Baumgartl
Chemnitz University of Technology
Department of Computer Science
{tobias.john, robert.baumgartl}@cs.tu-chemnitz.de

Abstract

Accurate estimation of worst case execution times (WCET) is an increasingly complex issue in real-time systems research. Unfortunately, many important details of latest processor architectures, e. g. processor caches and branch predictors, are very scarcely documented or completely obscured. It is up to the research community to deduce some of these parameters by performing experiments and interpreting available documentation accordingly.

This paper describes some new and unique techniques on how to obtain parameters of the underlying (processor-) cache architecture by a set of micro-benchmarks. By using performance monitoring registers instead of relying on timing information influenced by the analyzed hardware and employing a real-time executive as operating system environment we are able to obtain very precise measurement results. We describe methods to identify write miss and write hit policies. Further, for the first time, we describe methods to deduce cache replacement strategies. During our experiments, we observed different behavior in initial placement of cache data. Hence, we developed appropriate methods to characterize that aspect.

We apply the micro-benchmarks to a set of processors, discuss the obtained results and conclude underlying policies and strategies.

1. Introduction

Accurate estimation of worst case execution times (WCET) is an increasingly complex issue in real-time systems research. Architectural innovations in modern CPUs do almost always optimize the average case behavior. Technologies as branch prediction, complex caching hierarchies and super-scalar execution render WCET estimation a much more difficult task than it used to be.

Although embedded system CPUs differ in several aspects from CPUs commonly found in off-the-shelf PC systems, a trend of integrating more and more x86-based processors into embedded systems can be observed. PC CPUs of today are embedded CPUs of tomorrow. Therefore, an exact characterization of the execution timing of current processors is a must.

There exist a variety of models and methods to calculate WCET. However, those models incorporate architecture-specific parameters that pose a problem if unknown. Unfortunately, many important architectural details which influence timing are badly or not documented by the processor vendors.

It is up to the research community to deduce some of these parameters by performing experiments and interpreting available documentation accordingly. As has been pointed out before, this process is error-prone and can even lead to contradicting information on certain processor types [14].

This paper describes some new and unique techniques on how to obtain parameters of the underlying (processor-) cache architecture by a set of micro-benchmarks. By using special performance monitoring registers instead of generic cycle counters we are able to obtain very precise measurement results. Of course, the analyzed processor architecture must provide these monitoring registers which somewhat narrows the applicability of our approach to current CPU types. For the first time, we describe methods to deduce cache replacement strategies. During our experiments, we observed different behavior in initial placement of cache data. Hence, we developed appropriate methods to characterize that aspect.

The remainder of this document is structured as follows: In section 2 we describe relevant cache parameters and refer to related work, when necessary. The following section describes our experimental setup in some detail and discusses its advantages over existing solutions. Then, we describe experiments to distinguish between write-allocate and write-no-allocate as well as between write-through and

write-back policies. Further, we describe several experiments to deduce the used cache replacement strategy and the initial filling behavior. In section 4 we apply the described methodology to a range of Intel processors, discuss the obtained results and conclude underlying policies and strategies. Section 5 summarizes our work and gives an outlook on future research.

2. Cache Parameter Basics

Although quite some work has been done for WCET estimation in the large and cache modeling, comparatively few publications exist in the field of experimental processor cache characterization. Therefore, we give a brief overview on relevant caching parameters and list related work, where adequate.

2.1. Structural Cache Parameters

Structural cache parameters as cache size $S = 2^s$, line length $L = 2^l$ and associativity $W = 2^w$ can be estimated experimentally. Several publications describe corresponding methods and algorithms [12, 14]. Besides, information on S, A, L can be gathered through evaluating the bits returned by the `cpuid` instruction as it is done by tools like *cpuid* [13]. Hence, estimation of these parameters is not topic of this paper. We simple assume them to be known and to be powers of two.

2.2. Caching Policies

Depending on a hit or miss, four different policies can be distinguished for write accesses.

Writing a datum which is already in the cache can be performed in two ways. On the one hand, the write could be performed on both the datum in the cache and its copy in main memory ("Write-Through"). The alternative approach is to perform the write operation only on the cache. The main memory copy is not updated until the corresponding cache line is replaced ("Write-Back").

Similarly, writing a datum which is not in the cache can also be performed in two ways. Firstly, the datum could be written to main memory only ("Write-No-Allocate"). Alternatively, the datum could be written to both main memory and cache to accelerate subsequent read accesses ("Write-Allocate"). Table 1 summarizes the policies.

Experimental methods to estimate these policies have been published in [3] but are limited to the first level cache only. Additionally, they are based on timing measurements which are potentially imprecise as we argue in section 3.1.

Line in Cache?	Caching Policy
yes (Hit)	Write-Through
	Write-Back
no (Miss)	Write-Allocate
	Write-No-Allocate

Table 1: Caching Policies

2.3. Replacement Strategies

When the cache is warm and a miss happens, a cache line must be replaced. Several replacement strategies do exist, which are listed in table 2. A thorough description and evaluation of these strategies can be found in [9].

RND	Random
RR/FIFO	Round-Robin
LRU	Least Recently Used
pLRUt	Pseudo Least Recently Used (Tree-based)
pLRUm	Pseudo Least Recently Used (MRU-based)

Table 2: Cache Replacement Strategies

Our paper focuses on LRU and its derivatives, however the presented algorithms are easy to adapt and thus allow the identification of other strategies, too.

To our knowledge, techniques to discover the replacement strategies by experiments have not been described before.

2.4. Initial Fill Policy

After an invalidation the ways of a cache must be filled in a certain order. That order might or might not be based on the replacement policy. For precise cache modeling it is important to understand how this initial placement is performed.

During some experiments we discovered different initial fill patterns for a pLRUt cache. One approach obviously used the tree bits to store the data in a tree-based order (the data is stored the same way it is read). That means the replacement mechanism is applied for initially filling the cache. We refer to that behavior as "tree-based fill". The other approach simply populated the cache in sequential order. In that case the tree bits are ignored. We call this approach "sequential fill".

Figures 1 and 2 illustrate both approaches. They show a set of a 4-way cache before and after a line 'b' has been referenced. The cache contains a line 'a' and is otherwise

empty. The three squares denoted either 'l' or 'r' symbolize the tree bits which determine the path through the tree (symbolized by arrows) pointing to the pLRU cache line of this set. If tree-based filling is used, the tree bits are evaluated, the corresponding line (the 3rd in figure 1) is written and the appropriate tree bits are updated (complemented: l ↔ r). Figure 2 illustrates sequentially filling empty cache lines. The tree bits are neither evaluated not modified, therefore no arrows are drawn. Here the line 'b' is stored in the first free entry next to 'a'.

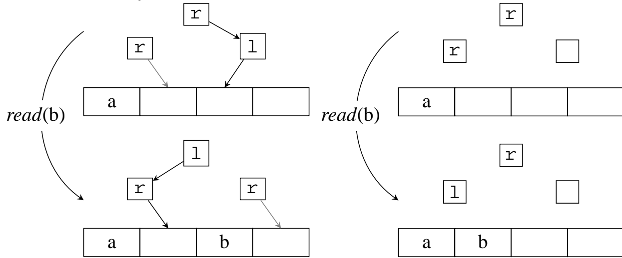


Figure 1: Tree-based Fill Figure 2: Sequential Fill

Of course, more and different initial fill policies may exist.

We describe a methodology to identify the caching hit and miss policies as well as the replacement strategy. Further, we describe, how the initial cache fill can be analyzed. All methodologies are based on a two-level, set-associative cache architecture as it is commonly used in today's PC and server processors.

3. Analysis Concepts

3.1. Experimental Setup

Invariably, previously published methodologies for cache parameter extraction rely on measuring execution times of small code fragments (often called micro-benchmarks) to differentiate between cache hits and misses and stress the good portability of that approach. On the other hand one can argue that using execution timing information for cache parameter extraction without knowing cache internals beforehand is somewhat risky. Influences of the at least partially unknown cache architecture as well as the underlying operating system may spoil the results. Therefore, we propose a different approach: use hardware counters to monitor only the events (e. g. cache hits or misses) which are caused by the hardware to be analyzed. In this way, we do not need to interpret execution times which are potentially influenced by a number of external factors. Of course, our approach is limited to processors which provide those monitoring capabilities. This includes Intel processors starting from the Pentium model ([8]) and AMD processors starting with the Athlon ([1]). Although the events that can be counted differ between architectures, the functionality and configuration is quite similar: one or more so-called Model Specific Registers (MSR) have to be set up to describe the desired event and one of the available performance monitoring counters has to be selected and configured to count the events in the specified way.

The following events must be monitored:

* L1 cache misses
* L2 cache misses
* L2 cache accesses

Using performance monitoring registers has the additional advantage of implicitly serializing execution flow. This prevents inaccuracies introduced by instruction reordering. Obtaining execution timing by accessing the time stamp counter (TSC) register requires additional efforts concerning serialization and is therefore more complex and error-prone.

As operating system environment we used the RTAI real-time executive [2] for the following reasons: The micro-benchmarks are executed as highest-prioritized RTAI tasks. Therefore, timing influences by user-space programs as well as the Linux kernel itself are eliminated. In this system configuration even interrupt processing can be postponed. Further, because RTAI applications execute in kernel mode, no virtual-to-physical address conversion is necessary, accessing physical memory is straightforward, and timing influences of the translation lookaside buffer (TLB) are also eliminated. Accessing hardware, especially manipulation of performance measurement registers is not restricted in kernel mode. To force caches to load a certain data set, it is necessary to access physical main memory at arbitrary locations, which is straightforward in kernel mode.

Existing profiling tools for kernel space are usually based on overflowing counters. The obtained results are too imprecise, hence we implemented routines for manipulating performance monitoring registers by hand. This poses no major problem, because overall implementation complexity is low. Of course, by using RTAI we need to implement all micro-benchmarks as Linux kernel modules which requires a certain amount of system knowledge. We felt that this can be justified.

Collecting and analyzing the obtained measurement data is not time-critical and can therefore be done in user mode by means of standard tools.

The resulting experimental setup ensures very precise results and a reasonable flexibility.

3.2. Notational Conventions

Instead of presenting C source code, we try to illustrate our algorithms a bit more formally. Hopefully, this eases

Identifier	Description	
W_i	Associativity	$(W_i = 2^{w_i})$
S_i	Cache Size	$(S_i = 2^{s_i})$
L	Line Length	$(L = 2^l)$
a_i	Address Width	
$\mathrm{A}_i, \mathrm{B}_i, \ldots$	Data that fills one Cache Way	
$\mathbf{\Omega}_0, \mathbf{\Omega}_1, \ldots$	Data that completely fills L1	

Table 3: Used Identifiers and Symbols

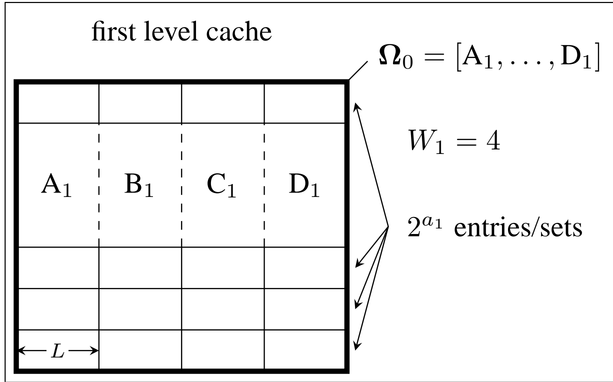


Figure 3: Cache Variables

porting and adapting the methods to newly evolving architectures. Table 3 summarizes the used identifiers. The index i refers to the level of the cache.

$\mathbf{\Omega}_j$ represents an arbitrary data set which completely fills L1. Some of the benchmarks need more than one such data set, therefore we index them by j. The addresses actually referenced are arbitrarily chosen, but it must be guaranteed that for any $\mathbf{\Omega}_j, \mathbf{\Omega}_k$ no single data element is shared between both sets.

It is common, although not necessary, that the line length is identical for both cache levels. This document only covers the case where $l_1 = l_2 = l$. A set-associative cache has $2^{s_i - w_i - l}$ entries and thus an address width $a_i = s_i - w_i - l$. Figure 3 illustrates some of the relevant parameters.

3.3. Analyzing Caching Policies

3.3.1. Cache Miss Policy

The write miss policy is easy to obtain. The idea is to write data to the cache and read that data afterwards. If the data is still in the cache, write-allocate is used. Accordingly, the following steps must be performed:

1. Invalidate caches (at least L1), so that no data is cached by issuing an appropriate machine instruction.

```
wbinvd
```

2. *Write* to as many addresses as fit into L1.

```
write(Ω0)
```

3. *Read* from these addresses and count the L1 misses n_{miss}.

```
reset_counter(L1_MISS)
read(Ω0)
n_miss := read_counter(L1_MISS)
```

If n_{miss} is (approximately) as high as the number of addresses in $\mathbf{\Omega_0}$, write-no-allocate is used. On the other hand, if n_{miss} is exactly or near zero, the data has been stored in L1 and a write-allocate policy has been identified.

3.3.2. Cache Hit Policy

The test for hit policies works with filled caches. If write-through is used, any write operation of a date residing in L1 is performed in both L1 and L2, whereas for write-back only the contents of L1 is modified (the cache line is marked dirty). The algorithmic idea of the micro-benchmark can be described as follows. First, a full clean cache contents is replaced and the number of needed L2 accesses is recorded. Second, the cache is filled again, then the cache contents is modified and afterwards again completely replaced. The needed L2 accesses for that second replacement is again recorded. If both values are approximately equal, write-through is identified. Otherwise, if the replacement of a cache which has been written to requires twice the number of L2 accesses needed for replacing a clean cache we can conclude that a write-back policy is used.

Hence, the structure of the benchmark is as follows:

1. Fill L1 twice by accessing $\mathbf{\Omega}_0$ first and $\mathbf{\Omega}_1$ afterwards, replacing $\mathbf{\Omega}_0$ by $\mathbf{\Omega}_1$ in L1.

```
read(Ω0)
read(Ω1)
```

2. *Read* $\mathbf{\Omega}_0$.
(It must be loaded from L2 and replaces the modified data set $\mathbf{\Omega}_1$). Count the L2 accesses n_{unmod}.

```
reset_counter(L2_ACCESS)
read(Ω0)
n_unmod := read_counter(L2_ACCESS)
```

Because the data has not been modified, it can be replaced regardless of the applied policy and L2 accesses should be nearly zero.

3. Repeat step 1 (fill L1 twice).
   As a result, addresses $\mathbf{\Omega}_1$ are cached in L1.

   read($\mathbf{\Omega}_0$)
   read($\mathbf{\Omega}_1$)

4. *Write* to the addresses $\mathbf{\Omega}_1$ in L1 (if L1 uses write-back then L2 is not updated).

   write($\mathbf{\Omega}_1$)

5. Repeat step 3 (read addresses $\mathbf{\Omega}_0$) and count L2 accesses n_{mod}. Because the data has not modified this time, n_{mod} depends on the cache hit strategy.

   reset_counter(L2_ACCESS)
   read($\mathbf{\Omega}_0$)
   n_{mod} := *read_counter*(L2_ACCESS)

If $n_{\text{mod}} \gg n_{\text{unmod}}$ it can be concluded that write-back is used. If $n_{\text{mod}} \approx n_{\text{unmod}}$ write-through has been identified.

3.4. Replacement Strategy

Identifying replacement strategies is somewhat more complex than the tests described so far. The algorithmic idea is to load the cache with a predefined content, causing replacement by accessing uncached data and analyzing which data has been replaced.

Then we load the cache again with the predefined content but with a different access history, cause again replacement and analyze, which data has been replaced this time. This process is repeated until sufficient knowledge on replacement has been accumulated and the replacement strategy can safely be identified.

The idea is similar to the black box approach in system theory: feed the system to be analyzed with a known input, record the output and draw conclusions on the transfer function of the system that transforms input x_i into output x_o. What we do is to provide several characteristic inputs by varying the number and order of ways that are reloaded, identify which ways should be replaced according to the applied strategy and compare those with the actually replaced ones.

This methodology poses the two challenges: Firstly, although replacement is performed line per line, measuring single replacement events is nearly impossible. Therefore, to obtain reliable results, whole cache ways must be replaced. This can be achieved by carefully crafting the replacing accesses. Secondly, choosing characteristic input data is not straightforward.

Again, for reasons of simplicity the algorithm is explained by means of an exemplary 4-way L1 cache. The detailed structure looks as follows:

1. Invalidate caches.

```
wbinvd
```

2. Fill L1 by *reading* way by way.

   $$\left.\begin{array}{l} \textit{read}(\mathrm{A}_1) \\ \quad\vdots \\ \textit{read}(\mathrm{D}_1) \end{array}\right\} \;\widehat{=}\; \textit{read}(\mathbf{\Omega}_0)$$

3. Reload (*read*) some of the ways already present in L1 to make them the most recently accessed ones. We manipulate L1 history without altering its content.

   read($\{\mathrm{A}_1, \ldots, \mathrm{D}_1\}$)
   $\vdots$

4. Load (*read*) a "new" way E_1 which replaces one of $\{\mathrm{A}_1, \ldots, \mathrm{D}_1\}$ according to the replacement policy and the current access history set up in step three.

   read(E_1)

5. *Read* $\mathbf{\Omega}_0$ way by way and record L1 misses for each way to find out which one has been replaced.

   reset_counter(L1_MISS)
   read(A_1)
   n_{A_1} := *read_counter*(L1_MISS)
   $\vdots$
   reset_counter(L1_MISS)
   read(D_1)
   n_{D_1} := *read_counter*(L1_MISS)

Exactly one of $\{n_{\mathrm{A}_1}, \ldots n_{\mathrm{D}_1}\}$ has a significantly larger value than the other three counters. The associated way has been replaced under the given cache configuration.

For the manipulation of the access history in step three we arbitrarily select the ways $\mathrm{A}_1, \mathrm{B}_1, \mathrm{C}_1$ and access them in different configurations which are depicted in column "reload" of table 4. The selection is thoroughly arbitrary,

different combinations (e. g. B_1A_1) are also possible, of course. Therefore, the complete cache access history for a single benchmark run consists of A_1, B_1, C_1, D_1 (column "load" in table 4) followed by one of the "reload" configurations. In the following discussion we omit the indices of the cache ways because we concentrate solely on level one.

The four rightmost columns of table 4 list the expected way to be replaced in step four of the algorithm under the replacement strategies pLRUt with tree-based fill, pLRUt with sequential fill, pLRUm and strict LRU (from left to right, respectively). Bold rows indicate different results depending on the replacement strategy.

Obviously, not every input row is a characteristic input. For instance, omitting the reload step completely does not allow to identify any replacement strategy, because way A is replaced invariably. Reloading with A, B is equally indifferent.

Clearly, it is possible to identify each of the four replacement strategies. For instance, the first run could reload ABC (row eight) and therefore allows to differentiate between tree-based pLRUt and LRU on one hand and sequential fill pLRUt and pLRUm on the other hand. Depending on the result of that initial benchmark run in the second run either C is reloaded (row five) to distinguish between sequential fill pLRUt and pLRUm, or A is reloaded (row two) to distinguish between sequential fill pLRUt and pLRUm. For our example cache configuration, two benchmark iterations are needed to identify the used replacement strategy.

Table 4: Cache Ways expected to be replaced by different Replacement Strategies

Load	Reload	pLRUt tree-b.fl.	pLRUt seq.fl.	pLRUm	LRU
A B C D	-	A	A	A	A
A B C D	A	**B**	**C**	**B**	**B**
A B C D	B	**A**	**C**	**A**	**A**
A B C D	A B	C	C	C	C
A B C D	C	**B**	**A**	**A**	**A**
A B C D	A C	B	B	B	B
A B C D	B C	**D**	**A**	**A**	**A**
A B C D	A B C	**D**	**A**	**A**	**D**

This technique also allows to identify newly-evolving replacement schemes which differ from well-known behavior.

To apply this micro-benchmark to the 2nd level cache too, the following condition has to be met:

$$a_2 \geq a_1 + w_1$$

Every way of L2 has to be at least as large as the whole L1 cache. Applied to the exemplary 4-way cache used in the preceding discussion that means: If ways have been loaded in the order $A_2, \ldots, D_2$ then L1 contains only D_2 (if $a_2 = a_1 + w_1$) but none of A_2, B_2, C_2. Therefore, reloading a way already present in L2 way really accesses L2 (and accordingly updates its history) and is not influenced by L1 data. Figure 4 illustrates that fact. If the L2 ways have been loaded in the order A_2, B_2, C_2, D_2 and afterwards way C_2 shall be reloaded, it is guaranteed that none of the addresses `C0.0-C31.127` still reside in L1, because they have been overwritten when loading D_2.

Yet there is still another simple method to test the replacement strategy: First the cache is filled with addresses $\mathbf{\Omega}_0$. Afterwards exactly one "new" way is read and it is checked which way it replaces. Then the cache is filled again with $\mathbf{\Omega}_0$ but this time two "new" ways are loaded and it is noted which two ways are replaced by them. This method is repeated until as many "new" ways are loaded as fit into the cache, that means until the whole $\mathbf{\Omega}_0$ has been replaced.
With that incremental replacing of one to $W_i = 2^{w_i}$ ways of a cache it can be observed which ways the pseudo LRU algorithm selects for replacement and this information can again be used to identify the replacement algorithm.

The structure of this benchmark is as follows (again, we use the 4-way-associative cache as demonstration example):

1. Invalidate caches.

```
wbinvd
```

2. Fill L1 by reading way by way.

$$\left.\begin{array}{l} \textit{read}(A_1) \\ \vdots \\ \textit{read}(D_1) \end{array}\right\} \;\widehat{=}\; \textit{read}(\mathbf{\Omega}_0)$$

3. Load (read) *one* "new" way which replaces *one* of $\{A_1, \ldots, D_1\}$.

read(E_1)

4. *Read* $\mathbf{\Omega}_0$ way by way and record L1 misses for each way to find out which one has been replaced.

reset_counter(L1_MISS)
read(A_1)
n_{A_1} := *read_counter*(L1_MISS)
⋮
reset_counter(L1_MISS)
read(D_1)
n_{D_1} := *read_counter*(L1_MISS)

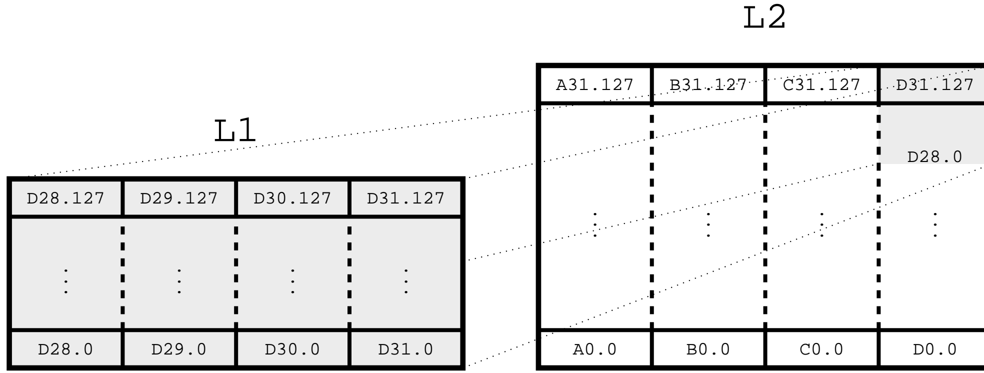


Figure 4: Sizing Relations L1 – L2, based on PII

Exactly one of $\{n_{A_1}, \ldots n_{D_1}\}$ has a significantly larger value than the other three counters. The associated way $\boldsymbol{X}_1$ has been replaced.

5. Repeat step 2.

   read(Ω_0)

6. Load (read) *two* "new" ways which replace *two* of $\{A_1, \ldots, D_1\}$.

   read(E_1)
   read(F_1)

7. Repeat step 4.
   Two ways must experience significant misses. They have been replaced: $X_1, \boldsymbol{X}_2$

   ⋮

10. Repeat step 4.
    Three ways must experience significant misses. They have been replaced: $X_1, X_2, \boldsymbol{X}_3$

    ⋮

13. Repeat step 4.
    All ways were replaced: $X_1, X_2, X_3, \boldsymbol{X}_4$

    $X_k = \{A_1, B_1, C_1, D_1\}, \quad k = \{1, 2, 3, 4\}$

As explained before (cf. figure 4) this test can be applied to L2 too. Only address ranges have to be adapted: $A_2, \ldots, D_2/A_2, \ldots, H_2$[1] have to be read to fill L2 and $E_2, \ldots, H_2/I_2, \ldots, P_2$[1] are those "new" ways to replace present ways in the cache.

[1] 4-way/8-way L2

4. Experimental Results

The algorithms described in the previous section were implemented as RTAI tasks and have been tested on several machines equipped with Intel Pentium II, III and 4 (without Hyperthreading) processors running a ADEOS-patched Linux kernel version 2.6.8.1 with RTAI 3.1. The software is freely available on request.

4.1. L1 Caching Policies

4.1.1. Cache Miss Policy

Results on the Pentium II/III

512 lines can be stored in L1 and we encountered an average of $\bar{n}_{\text{miss}} = 16$ misses when reading L1. This miss ratio of 3.1 % indicates that a write-allocate policy is used.

Results on the Pentium 4

128 lines can be stored in L1 and we measured an average of $\bar{n}_{\text{miss}} = 22$ misses when exhaustively reading L1. Whereas the absolute number of misses can be compared to the preceding result the miss ratio of 17.2 % is not as small as on the Pentium II/III. We believe that the smaller number of lines is the reason for that increase. Because it is still negligible, we can conclude that write-allocation is used.

4.1.2. Cache Hit Policy

Results on the Pentium II/III

The performance monitoring of the Pentium II and III processors provides two different events potentially usable as metric for L2_ACCESS: the cycles the L2 data bus is busy and the number of L2 address strobes. We chose the latter one and implemented micro-benchmarks described in

section 3.3.2 for that event type. Table 5 presents the obtained results.

	L2 addr strobes
n_{mod}	2 057
n_{unmod}	1 063

Table 5: Results on Cache Hit Policy - Pentium II

Table 5 shows clearly that the modified data in the L1 cache has to be written back to L2 first, before it can be replaced: twice as many L2 address strobes are necessary. We can conclude that the first level cache of the Intel P6 family utilizes a write-back policy.

This fact is in accordance with Intel's article [4] which describes differences between the Netburst architecture and "Earlier Pentium Family Processors" (Pentium Pro, II, III). Accordingly, [11] is wrong in assigning a write-through policy to the Pentium III.

Results on the Pentium 4

Unfortunately, the performance monitoring of the Pentium 4 does not allow to count the same events as on earlier processors. Therefore, we decided to take the front side bus (FSB) read and write operations as as indicator for L2 cache accesses. Table 6 presents the measurement results.

	FSB reads	FSB writes
n_{mod}	128	0
n_{unmod}	129	0

Table 6: Results on Cache Hit Policy – Pentium 4

Obviously, there is almost no difference whether the content of L1 has been modified or not and the conclusion must be that the Netburst architecture uses a L1 cache with a write-through policy. This is in accordance with information in [4], [5] and [6].

4.2. Cache Replacement Strategies

Results on the Pentium II/III

Firstly, we applied several characteristic inputs to the replacement benchmark described in section 3.4. The cache is 4-way-associative, therefore we could employ the input data without modification.

Table 7 shows the replaced cache way for every input data pattern. Comparing the results with table 4 we can conclude that the Pentium II/III uses a tree-based pLRU strategy with initial tree-based filling.

Reloaded Way(s)	Replaced Way
–	A
A	B
B	A
A, B	C
C	B
A, C	B
B, C	D
A, B, C	D

Table 7: Results on L1 Replacement Strategy – Pentium II

Secondly, we implemented a similar test for Pentium II/III L2 cache, whose results are presented in table 8.

Reloaded Way(s)	Replaced way
–	A
A	C
B	C
A, B	C
C	A
A, C	B
B, C	A
A, B, C	A

Table 8: Results on L2 Replacement Strategy – Pentium II

Obviously, in contrast to L1, the second level cache of the Pentium II and III uses a pLRUt strategy with sequential initial fill.

The Intel Pentium II Processor Developer's Manual [7] only states that pseudo-LRU is used for both caches, but not which type and initial behavior. Moreover, Sears [10] is wrong with the conclusion that the processor uses a LRU strategy.

Results on the Pentium 4

Identifying the replaced cache line in the L1 cache proved to be more difficult. Figure 5 illustrates the measured L1 misses for the individual reload configurations.

It is nearly impossible to identify a single replaced cache way for certain reload configurations (e. g. AB or AC). We believe, this behavior again results from the comparatively low number of only 32 cache sets.

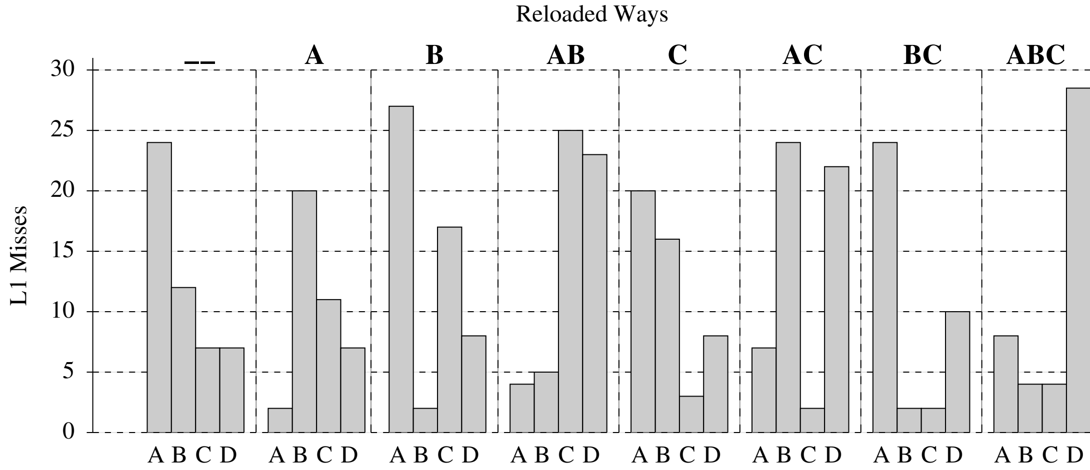


Figure 5: L1 Cache Misses for Replacement Strategy Benchmark – Pentium 4

Nevertheless, lacking a better one, we applied our described methodology and obtained the results depicted in table 4. None of the four replacement strategies fully matches the observed behavior, the closest one is pLRUt with tree-based filling which deviates only in two cases.

Taking the above mentioned inaccuracies into account, it seems nevertheless reasonable to infer pLRUt with tree-based filling for Pentium 4 L1 cache.

Reloaded Way(s)	Replaced Way	Tree-based Fill
–	A	A
A	B	B
B	A	A
A, B	C	C
C	**A**	**B**
A, C	B	B
B, C	**A**	**D**
A, B, C	D	D

Table 9: Results on L1 Replacement Strategy – Pentium 4

L2 cache is 8-way associative on the Pentium 4, therefore the micro-benchmarks have to be adapted accordingly. The cache utilizes 1024 sets, therefore the event miscounts do not play any role. However there are two mysterious results we are not able to explain yet:

* The reloading of two ways A, B and the following reading of a new way I seems to evict *both* ways C *and* D. It is important to note that not half the way C and D are purged from L2 to obtain *one* free way, but *both* ways C, D are freed!
* The reloading of A, B, C and the following reading of a new way I leads to an almost equal distribution of L2 misses between all eight present ways A, . . . , H.

In all other input data configurations, the (single) replaced way can be identified easily by its high number of L2 misses.

The highlighted rows in Table 10 indicate differences in the ways that should be replaced (3rd column) and those that actually are replaced (2nd column). With a reasonable degree of certainty we conclude that the L2 cache of the Pentium 4 uses a pLRUt policy with a tree-based filling, too. Yet we must admit that we still do not understand fully Pentium 4 level 2 cache replacement behavior. This is subject of further research.

Reloaded Way(s)	Replaced Way	Tree-based Fill
–	A	A
A	B	B
B	A	A
A, B	**C, D**	**C**
C	B	B
A, C	B	B
B, C	D	D
A, B, C	**?**	**D**

Table 10: Result on L2 Replacement Strategy – Pentium 4

5. Conclusions and Future Work

We described methods to analyze the first and second level processor caches. In contrast to existing solutions we do not use timing information to differentiate between cache hits and cache misses. Instead we propose to directly count relevant events using performance monitoring registers. Our methodology eliminates timing influences of the operating system and concurring applications.

We described micro-benchmarks to experimentally discover cache miss and cache hit policies. Furthermore our methodology allows to determine the replacement strategy in detail. We are able to obtain information about the initial cache filling.

We successfully applied our methodology to a range of Intel processors. We were able to verify some information on caching behavior and falsified some other statements. We discovered new details on Intel's cache behavior.

Currently we implement micro-benchmarks for a much wider range of microprocessors especially for architectures different from IA-32. We hope to precisely analyze embedded processor's caching soon. Crafting the micro-benchmarks manually is tedious work, therefore another research focus is on (semi-)automatically generating benchmark code.

We feel that statements on branch predictors are equally as fuzzy and contradicting as we experienced for cache behavior. Using our proved methodology first tests on that matter have been carried out already.

The next step is to include cache performance parameters into our experimental setup. That way we are able to quantify and compare cache performance of different architectures. Finally, we hope to contribute towards precise cache models for WCET estimation of complex processors.

References

[1] Advanced Micro Devices. *AMD Athlon Processor x86 Code Optimization Guide*, 2002.

[2] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI Position Paper. In *Proceedings of the 21st IEEE Real-Time Systems Symposium and Real-Time Linux Workshop*, Orlando, FL, Nov. 2000.

[3] L. Enyou and C. Thomborson. Data Cache Parameter Measurements. In *Proceedings of the IEEE International Conference on Computer Design*, pages 376–383, Oct. 1998.

[4] B. Hayes. Differences in Optimizing for the Pentium Processor vs. the Pentium III Processor. Whitepaper 44010.

[5] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 2001. Q1.

[6] Intel. *IA-32 Intel Architecture Optimization Reference Manual*.

[7] Intel. *Pentium II Processor Developer's Manual*, 1997. 243502-001/24333503.

[8] Intel. *IA-32 Intel Architecture Software Developer's Manual*, volume 3. Intel Corporation, 2004.

[9] M. Milenkovic. Performance Evaluation of Cache Replacement Policies. Technical report, University of Alabama in Huntsville, 2004.

[10] C. B. Sears. The Elements of Cache Programming Style. Technical report, Google Inc., 2000. Proceedings of the 4th Annual Linux Showcase & Conference Atlanta.

[11] F. Sebek. Cache Memories and Real-Time Systems. Technical report, Mälardalen Unirversity, 2001. p. 24–25.

[12] C. Thomborson and Y. Yu. Measuring Data Cache and TLB Parameters Under Linux. In *Proceedings of the 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 383–390. Society for Computer Simulation International, July 2000.

[13] A. Todd. *http://www.etallen.com/cpuid.html*, last referenced 05-17-06.

[14] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.

Towards Predictable, High-Performance Memory Hierarchies in Fixed-Priority Preemptive Multitasking Real-Time Systems

E. Tamura
Grupo de Automática y Robótica
Pontificia Universidad Javeriana – Cali
Calle 18 118–250, Cali, Colombia
eutamo@doctor.upv.es

J. V. Busquets-Mataix and A. Martí Campoy
Departamento de Informática de
Sistemas y Computadores
Universidad Politécnica de Valencia
Camino de Vera s/n., 46022 Valencia, España
{vbusque, amarti}@disca.upv.es

Abstract

Cache memories are crucial to obtain high performance on contemporary computing systems. However, sometimes they have been avoided in real-time systems due to their lack of determinism. Unfortunately, most of the published techniques to attain predictability when using cache memories are complex to apply, precluding their use on real applications. This paper proposes a memory hierarchy such that, when combined with a careful pre-existing selection of the instruction cache contents, it brings an easy way to obtain predictable yet high-performance results. The purpose is to make possible the use of instruction caches in realistic real-time systems, with the ease of use in mind. The hierarchy is founded on a conventional instruction cache based scheme plus a simple memory assist, whose operation offers a very predictable behaviour and good performance thanks to the addition of a dedicated locking state memory.

1 Introduction

Contemporary computing systems include cache memories in their memory hierarchy to increase average system performance. In fact, cache memories are crucial to obtain high performance when using modern microprocessors. While trying to minimise the average execution times, the contents of the cache memories vary according to the execution path. General-purpose systems benefit directly from this architectural improvement; however, minimising average execution times is not so important in real-time systems, where the worst-case response time is what matters the most. Thus, due to their lack of determinism, sometimes cache memories have been avoided in fixed-priority preemptive multitasking real-time systems: when they are incorporated in such a system, in order to determine the memory hierarchy access times as well as the delays involved in cache contents replacement it is necessary to know what its contents are.

Using cache memories in fixed-priority preemptive multitasking real-time systems presents two problems. The first problem is to calculate the *Worst-Case Execution Time* (*WCET*), due to intra-task or intrinsic interference. *Intrinsic interference* occurs when a task removes its own instructions from the instruction cache (*I-cache*) due to conflict and capacity misses. When the removed instructions are referenced again, cache misses increase the execution time of the task. This way, the delay caused by the I-cache interference must be included in the WCET calculation. The second problem is to calculate the *Worst-Case Response Time* (*WCRT*) due to inter-task or extrinsic interference. *Extrinsic interference* occurs in preemptive multitask systems when a task displaces instructions of any other lower priority tasks from the I-cache. When the preempted task resumes execution, a burst of cache misses increases its execution time. Hence, this effect, called cache-refill penalty or *Cache-Related Preemption Delay* (*CRPD*) must be considered in the schedulability analysis.

This work proposes

- a memory hierarchy that provides high performance coalesced with high predictability. The solution is to be centred on instruction fetching since it represents the highest number of memory accesses [15];
- the required schedulability analysis for such hierarchy; and
- some evaluation results and its analysis.

Results show that

- the proposed memory hierarchy is predictable and simple to analyse;

- its performance exceeds that of the dynamic use of locking cache as given in [10]; and
- in many cases, its performance is about the same than that obtained when using a conventional instruction cache.

The remainder of the paper is organised as follows. Section 2 introduces the problem and summarises some of the solutions found in the literature. Section 3 describes the proposed memory hierarchy, its requirements, a functional description of its operation and the schedulability analysis. Section 4 assesses the proposed memory hierarchy by comparing it with the dynamic use of locking cache as given in [10]. First predictability and prediction accuracy are examined by comparing estimated and simulated worst-case response times. Performance is evaluated by measuring the worst-case processor utilisation. Some concluding remarks are given in Section 5.

2 Rationale

In order to guarantee that every task in the task set meets its deadline, real-time system designers may opt for three different approaches:

- Use the memory hierarchy in a conventional manner.
- Use the memory hierarchy in a real-time systems suitable manner.
- Use a real-time systems aware memory hierarchy.

Each approach will be briefly summarised according to three different perspectives: architectural viewpoint, implementation viewpoint and, run-time support viewpoint.

2.1 The memory hierarchy is used in a conventional manner.

When using cache memories in a conventional way, the memory hierarchy is the same used in any conventional system with cache memories; therefore, regarding implementation and run-time support, there is no need to implement any additional hardware or software modules. Instead, the real-time system designer does his/her best to determine whether each memory reference causes a cache hit or a cache miss. This is done by using static analysis techniques. Some of the techniques used for WCET calculation are data-flow analysis [13, 22], abstract interpretation [1], integer linear programming techniques [6], or symbolic execution [7]; to tackle the WCRT estimation data-flow analysis is also used. Unfortunately, the complexity of static analysis techniques may preclude their use in practical applications.

2.2 The memory hierarchy is used in a real-time systems suitable manner.

An alternative to fully exploit the inherent performance advantage of cache memories while achieving predictability is to work with unconventional memory hierarchies. In this case, instead of conventional cache memories, the real-time designers favour the use of either locking caches [10, 18, 25, 2, 17] or scratchpad memories [26, 27]. On the one hand, locking caches are caches with the ability to lock cache lines to prevent its replacement; blocks are loaded into the locking cache and then they are locked. They are accessed through the same address space as the main memory. On the other hand, scratchpad memories are an alternative to I- or D-caches (data caches). They are small and extremely fast SRAM memories (since they are usually located on-chip); they are mapped into the processor's address space and are addressed via an independent address space that must be managed explicitly by software.

Regarding implementation, in both cases, during the design phase it is necessary to choose for every task in the task set which instruction blocks will be either loaded and then locked into the locking cache or copied into the scratchpad memory. The number of selected blocks per task must not exceed the capacity of either the locking cache or the scratchpad memory (selecting which information is copied into a scratchpad is very close to deciding which information has to be locked into a locking cache). Once the blocks are chosen, it is possible to know how much time it would take to fetch every instruction in the whole task set; therefore, the access time to the corresponding memory hierarchy is thus predictable. At compile time, the assignment of memory blocks to either the locking cache or the scratchpad has to be handled by hand or automatically using a compiler and/or a linker. However, since scratchpad memories are mapped in the processor's memory space, explicit modifications in the code of tasks may be required to make control flow and address corrections.

To improve the execution performance of more than one task (as is desirable in a fixed-priority preemptive multitasking real-time system), the contents of either the scratchpad or the locking cache memory should be changed at run-time (dynamic use). Thus, in both cases, the subset of blocks selected for every task should be loaded during system execution by a software routine, which is executed each time the real-time system designer judges convenient. Transfers to and from scratchpad memories are under software control while for locking caches this is transparent. While a task is not preempted, it is necessary to ensure that the contents of either the scratchpad or the locking cache will remain unchanged. This way, extrinsic interference is eliminated while intrinsic interference can be bounded. In [10] using locking instruction caches is proposed to cope with both ex-

trinsic and intrinsic interferences; in [25], the use of locking D-caches is proposed to enhance predictability by inserting locking/unlocking instructions: the cache is locked whenever it is not possible to statically determine whether the memory references a datum inside the cache or not. In several cases, the dynamic use of locking I-caches effects the same or better performance than using a conventional I-cache [10]. In [27] by using scratchpads performance gains comparable to that of caches are also obtained. However, since the amount of scratchpad memory available is often small compared to the total amount of cache memory available, intuitively, it is reasonable to think that for task sets with big tasks the scratchpad memory approach may obtain lower performance than the cache memory approach.

No matter

1. which mechanism is used to trigger the execution of a small software routine to either load blocks into the locking cache (at the scheduler level, as proposed originally in [10] or via debug registers by raising exceptions when the program counter reaches specified values [2]) or copy blocks to the scratchpad memory; and,

2. the location of the software routine (e.g., in main memory or even in a scratchpad memory),

the execution of the aforementioned software routine demands valuable processor cycles. Since this execution time must be added to the task's WCRT, the overhead introduced when using either locking caches or scratchpad memories in a fixed priority multitasking real-time system may have severe consequences on performance.

2.3 The memory hierarchy is real-time systems aware.

A third option is to design more predictable memory hierarchies. A memory hierarchy for fixed-priority preemptive multitasking real-time systems must implement mechanisms which in some way address the effects of

- intrinsic interference: it must prevent that the contents of the cache are overwritten by the same task;
- preemption: by allowing the preempting task to overwrite the contents of the cache; and
- extrinsic interference: it must allow that the contents of the cache are restored when the preempted task resumes execution.

To deal with extrinsic interference, some of the approaches use cache partitioning techniques, which allocate portions of the cache to tasks via hardware (I-cache [5], D-cache [14]), software (by locating code and data so they will not map and compete for the same areas in the cache) [28, 12] or a combination of hardware and software [19, 3]. Notice that the technique proposed in [5] introduces unpredictability for blocks that go to the shared pool.

To improve predictability, [4] proposes to extend the cache hardware and to introduce new instructions to control cache replacement (kill or keep cache blocks).

In [24], a custom-made cache controller assigns partitions of set associative caches to tasks so that extrinsic interference is eliminated; cache partitions are assigned to tasks according to their priorities by using a prioritised cache: each partition is assigned dynamically at run time; higher priority tasks can use partitions that were previously allocated to lower priority tasks. A partition allocated to a higher priority task cannot be used for a lower priority task unless the former notifies the cache controller to release the partitions it owns (which is done when the task is completely over). Therefore, it might be possible that the highest priority tasks consumes the whole cache memory and jeopardises the lowest priority tasks response times.

The work presented in this paper is a refinement of previous work [23] and proposes the use of an I-cache and additional hardware information to influence the I-cache replacement decision. This "cache replacement policy" provides a mechanism to increase predictability (time-determinism) without degrading performance, making it suitable for use in fixed-priority preemptive multitasking real-time systems. In this approach, the subset of selected blocks for each task and the instants in which I-cache flushing takes place are fixed: Every time a task begins or resumes its execution, the I-cache is flushed and then it is gradually reloaded with selected blocks as the instructions belonging to the task to be dispatched are being fetched. The selected blocks are inhibited from being replaced until a new context switch takes place. This way, the access time to the memory hierarchy is predictable and on the other hand, each task may use all the available I-cache space in order to improve its execution time.

In contrast to other approaches, the proposed memory hierarchy does not need any software to load the selected blocks into the I-cache at run time and hence it does not introduce penalties in the task's WCRT.

3 Memory hierarchy architecture

Efficient operation of the memory hierarchy requires an efficacious, automatic, on-demand storage control method that frees the software from explicit management of memory addressing space. Furthermore, the resulting architecture should not introduce any additional delays and be as open as possible by using generic components.

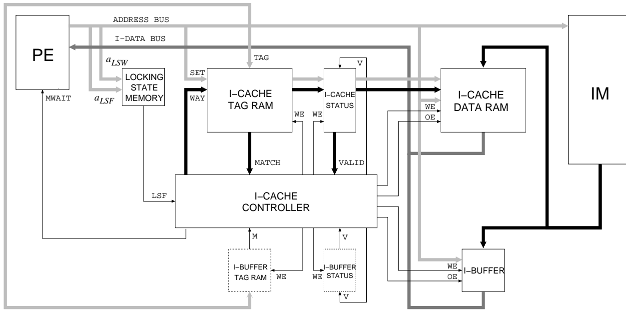


Figure 1. Proposed memory hierarchy

3.1 Description

Figure 1 sketches an architecture that pursues these goals. As can be seen, the figure does not embody any locking I-cache; it resembles a system for a conventional I-cache. There are however three noteworthy differences:

- There is an extra, dedicated, very fast SRAM memory, the *Locking State Memory* (*LSM*), located to the right of the *Processing Element* (*PE*). Its role is to store the status of every instruction block (the *Locking State*, *LS*) in the *Instruction Memory* (*IM*), thus providing a mechanism to discriminate which blocks must be loaded into the I-cache and hence a way to allow for automatic, on-demand loading of the selected instruction blocks. In other words, instead of locking selected blocks into an instruction locking cache, the same effect can be attained by avoiding loading into the I-cache unselected blocks.
- There is also an *Instruction Buffer* (*I-buffer*), with size equal to one cache line, located below the I-cache controller. Having an I-buffer is not essential, rather it is more of a performance assist: its purpose is to take advantage of the sequential locality for those blocks that should not be loaded into the I-cache. Since the I-buffer catches and holds previously used instructions for reuse, it might also contribute with temporal locality by providing look behind support (via the boxes drawn with dashed lines in the bottom part of the figure).
- There is also a subtle difference in the control bits of the I-cache with respect to a locking cache: since locking state information is stored into the LSM, locking status bits are not required.

3.2 Performance requirements

The main goal of the memory hierarchy is to provide deterministic yet high-performance response times.

In order to achieve determinism, each time a task τ_i is dispatched for execution, its corresponding subset of previously selected blocks, SB_i, is loaded into the I-cache as the PE fetches them. Once loaded, the selected blocks must remain in the I-cache and must not be overwritten as long as task τ_i is either not preempted by other, higher priority tasks or it finishes. This policy, which is applied to every task in the task set, eliminates intrinsic interference since the task is not allowed to remove any block previously loaded into I-cache, thus contributing to temporal determinism. Furthermore, extrinsic interference is bounded and can be estimated in advance.

Both temporal locality, the tendency to access instructions that have been used recently, and spatial locality, the tendency to involve a number of instructions that are clustered, are essential to performance. Hence, by keeping the SB_i blocks loaded in the I-cache, temporal locality is mainly captured by the I-cache yet spatial locality is also supported. Besides that, the I-buffer captures spatial locality for those blocks not in SB_i, albeit as it was said before, it might also provide some temporal locality.

With respect to timing issues, the goal is to cause minimum overhead during I-cache (re)load: since the LSM is not in the critical path, IM latency remains the same. LSM access time however must be in the order of a cache hit time to operate in parallel with the I-cache and its controller. This way, the I-cache inner workings are not affected and hence, its timings remain about the same.

3.3 Storage requirements

Storage requirements for the LS are also very important: space consumption should be low. Regarding cost, the most useful measure is to determine how much memory needs to be added to the system.

The minimum amount of memory required to keep track of each selected block is one bit. Hence, there will be as many bits as the number of blocks in the IM. Each of those bits will store a flag, the *Locking State Flag* (*LSF*), which is used to signal whether the corresponding block should be loaded into I-cache or not. For LS packing purposes, however, it is better to group the information into wider, off-the-shelf, fast SRAM memories. Henceforth assume an 8-bit wide LSM; then, the information for eight blocks (a *parcel*) will be stored in one *Locking State Word* (*LSW*) as shown in Figure 2.

Let L be the I-cache line size in bytes and let b_I be the number of bytes per instruction; then each memory block has L/b_I instructions. Given an IM of depth $d_{IM} = mL$, where m is the number of instruction blocks, the required LSM has a depth, d_{LSM}, equal to $m/8$. Then, the number of instructions, I, that corresponds to each LSW is given by $I = 8L/b_I$.

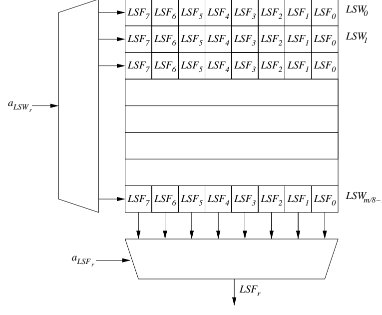


Figure 2. Locking state memory

Let w be the IM width in bits, a be the width of the address bus in bits, K be the degree of associativity and N the number of sets, then S_{IM}, the space required for IM; S_{CM}, the space required for I-cache memory; and S_{LSM}, the space required for LSM, are given (in bits) by:

$$S_{IM} = wmL \quad (1)$$

$$S_{CM} = wLKN + \left(a - \log_2 \frac{L}{N}\right) \times KN + KN \quad (2)$$

$$S_{LSM} = m \quad (3)$$

In the expression for S_{CM}, the first term is related to the I-cache data memory, the second one reflects the space needed for its tag memory and the last one accounts for its status bits (just the valid bits are considered). Notice that lock bits are not necessary since they are grouped into the LSM.

Therefore, the space efficiency, η_s, which measures the fraction of memory dedicated to store LS, can be defined as the ratio of the LSM space, S_{LSM}, to the total amount of memory space:

$$\eta_s = \frac{S_{LSM}}{S_{IM} + S_{CM} + S_{LSM}} \quad (4)$$

3.4 Functional operation

During system design, given a task set, TS, an off-line algorithm selects a subset, SB_{TS}, from the task set instruction memory blocks ($SB_{TS} = \bigcup SB_i, \forall\, \tau_i \in TS$).

The LS associated to TS, LS_{TS}, which reflects the status of every instruction block in TS, must then be loaded into the LSM and it will remain fixed during system execution.

When the system starts operating, the PE must reset the I-cache controller and invalidate all the entries in the I-cache as well as in the I-buffer.

Now, every time that an instruction, I_r, at address a_r is referenced by the dispatched task, τ_i, the LSM needs to be accessed to check the LSW at address a_{LSW_r}. This is the address of the LSW that corresponds to m_r, the memory block that embodies I_r. Hereafter, assume a 32-bit wide instruction size and a byte-addressable IM. Then, a_{LSW_r} is obtained by stripping off the $\log_2 8b_I$ least significant bits of a_r.

Finally, it is necessary to extract LSF_r, the corresponding LSF within LSW_r to drive the `LSF` signal and thus determine whether it is necessary to load m_r in the I-cache. The LSF is indexed by using the 3 bits next to the $\log_2 b_I$ least significant bits of a_r to drive an 8-way multiplexer.

At the same time, the tag for m_r is compared in the I-cache directory thus updating the `MATCH` signal and its corresponding line status is checked via its `VALID` bit. Simultaneously, the data portion of the I-cache is also accessed. Based upon the `LSF`, `MATCH` and `VALID` signals, the I-cache controller may have three possible outcomes:

- The `LSF` signal is `1`, indicating that m_r must be loaded and locked in the I-cache so the I-buffer is disabled; in other words, $m_r \in SB_i$. If the reference causes a miss (because either there is no tag match or the entry is not valid), m_r is loaded from IM into the I-cache, the corresponding tag is updated and its valid bit is set. Afterwards, the I-cache controller, via the `MWAIT` line, signals the PE that the instruction is available so that it can restart fetching.

- The `LSF` signal is `1`, but the reference results in a hit (because the instruction was previously referenced during the current execution). Then, the PE can fetch the instruction from the I-cache without incurring in any further delays.

- The `LSF` signal is `0`, indicating that m_r should not be loaded in the I-cache; in other words, $m_r \notin SB_i$. In this case, the I-cache is disabled and it is necessary to access the IM in order to load m_r in the I-buffer.

Each time a context switch occurs, the scheduler executes an instruction that causes that the entire I-cache contents are purged (its valid bits are reset) and therefore, every line is invalidated; the I-cache controller should also be reset to avoid that it finishes incomplete operations taking place when the context switch happened. Not purging the I-cache might bring better performance but in any case, it is quite difficult to estimate which blocks will remain in the I-cache after several preemptions; furthermore, it is harder to know if those blocks will be used at all once the preempted task resumes execution. Thus, since one of the primary goals is to keep the schedulability analysis simple, it is better to purge the cache on each context switch. Notice however that this may introduce an overestimation in the schedulability analysis.

Using an LSM in the proposed way imposes a constraint: since in a conventional I-cache there is no hardware impediment to replace its lines, the block selection algorithm must

guarantee that for any set in the I-cache there will be no conflict misses. Otherwise, selected blocks, which are already loaded, may be overwritten. This might cause some performance improvements, but at the same time, its predictability will deteriorate and hence, the analyses will turn more complex.

Aside from this restriction, it is important to note that the focal feature of the memory hierarchy is the inclusion of the LSM. With the LSM, the proposed memory hierarchy is able to provide a *Virtual Locking I-cache*. Its key advantage is that it uses a conventional I-cache like a locking I-cache. This approach then, takes advantage of the I-cache intrinsic features while at the same time avoids the overhead required to load instructions into the locking I-cache and the explicit manipulation of its locking mechanism.

3.5 Schedulability analysis

The schedulability analysis is done in two steps: in the first step, the WCET of each individual task is calculated assuming that it is the only task in the system but accounting for the intrinsic interference. Subsequently, the effect of the extrinsic interference is considered in the second phase, the calculation of the WCRT.

Task's WCET is estimated by using a *Cache Aware Control Flow Graph*, *CACFG*, an extended *Control Flow Graph*, *CFG* [21]. In a CACFG, each memory block is mapped to a cache block and assigned a block number and each *basic block* (i.e., each sequence of instructions with a single entry/single exit point) inside the memory block is mapped to a different vertex. Thus, CACFG models not only the flow control of the task through vertices (as it happens in CFG) but also takes into account the presence of the I-cache by modelling how the task is affected from the point of view of the cache structure.

The WCET of tasks may then be easily estimated considering the execution time of each vertex: Let a task τ_i, with selected vertices $V_i \in SV_i \subseteq SB_i$. The execution time of a vertex depends on the number of instructions inside it, k_{V_i}, and the cache state when the instructions inside the vertex are executed. Since

- in the worst case, SB_i, the subset of selected blocks, and hence SV_i, the subset of its corresponding vertices, will always be loaded on-the-fly by the proposed memory hierarchy each time τ_i executes; and,
- each block, once loaded, will remain in the I-cache as long as task τ_i is not preempted (or it finishes),

it is possible to affirm that, in this particular case, the cache state for τ_i is essentially the same during each of its activations. Thus, the execution times for τ_i's vertices are constant across each execution.

Its WCET can then be estimated assuming that all of the vertices in SV_i are already loaded in the I-cache and then adjust this WCET by accounting for the time required to load SB_i. Hence, if the subset of selected blocks is already loaded in the I-cache and the execution time of any instruction (not including the fetch time) is given by t_I, the WCET for a vertex is given by:

$$k_{V_i} \times (t_I + t_{hit}), \quad \forall\, V_i \in SV_i \tag{5}$$
$$k_{V_i} \times (t_I + t_{hit}) + t_{miss}, \quad \forall\, V_i \notin SV_i \tag{6}$$

and C_i, the WCET for any task can be estimated by applying the approach given in [21]. Notice however that Equation 6 introduces an overestimation in the schedulability analysis whenever there is a control transfer from one vertex to any other vertex that belongs to the same memory block.

Nevertheless, the previous assumption makes necessary to adjust the execution time of those instructions contained in every selected block, B_i. Then, for each selected block B_i not loaded into I-cache, task τ_i will incur in an overhead given by t_{miss} (a compulsory miss).

When estimating the WCET for every task τ_i, the worst case scenario regarding the blocks in SB_i implies loading all of its blocks. Thus, this preemption penalty can be accounted for by adding the term $k_{SB_i} \times t_{miss}$ to the previously calculated WCET:

$$C'_i = C_i + L_{SB_i} \tag{7}$$
$$L_{SB_i} = k_{SB_i} \times t_{miss} \tag{8}$$

where k_{SB_i} is the number of selected blocks for task τ_i. Notice that when using a scratchpad memory or a locking cache in a dynamic way (i.e., by modifying its contents at run time, it is necessary to add an extra term to L_{SB_i}: $\Delta_{SW_{rSB_i}}$, that takes into account the time required to execute the software routine in charge of replacing the corresponding memory.)

WCRT is then obtained by using Equation 9, where the I-cache refill penalty due to extrinsic interference is incorporated in parameter γ^i_j.

$$w_i^{n+1} = C'_i + \sum_{\forall\, \tau_j \in hp(\tau_i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times \left(C'_j + \gamma^i_j\right) \tag{9}$$

Computing γ^i_j is not easy because tasks may suffer two kinds of interference: direct interference or indirect interference. *Direct interference* means that a task increases its response time because it is forced to reload its own instructions, previously removed by its preempting tasks. *Indirect interference* means that a task increases its response time because executing any other higher priority tasks increases its response time, due to its own direct and indirect extrinsic interference.

It is hard to know which kind of extrinsic interference a task will suffer during its execution; then, to consider both possibilities, it is safe to use the maximum I-cache refill penalty:

$$\gamma_j^i = \left[\max(k_{SB_j}) + 1\right] \times t_{miss}, \forall\, j \in hp(i) \quad (10)$$

Using the maximum I-cache refill penalty gives a safe, upper bound while keeping the complexity low. This may be somewhat pessimistic: it may happen that not all of the loaded blocks are going to be used before the next preemption. Nevertheless, getting a more precise value in advance will involve complex analyses, since it depends on the number of blocks effectively loaded and the exact preemption instants.

Equation 9 is a recursive equation that is solved iteratively; the resulting WCRT, R_i, is then compared to τ_i's deadline to decide schedulability.

4 Assessing the proposed memory hierarchy

The proposed architecture, when operating in Virtual Locking I-cache mode, is able to guarantee determinism per se (since it is possible to analyse its impact on the WCRT of every task), but system performance strongly depends on the blocks selected to be loaded in the I-cache. Thus, this selection must be carefully accomplished. In fixed-priority preemptive multitasking systems, tasks response times depend on the execution time of higher priority tasks. In addition, indirect interference causes that the response time of tasks depends on the time needed to reload the I-cache contents. Therefore, I-cache contents must be selected considering not the isolated tasks, but all of the tasks in the task set.

Then, the goal is to optimise some temporal metric by selecting a subset of instruction blocks, SB_{TS} from the set of instruction blocks, B_{TS}. Choosing the cache contents in a way that maximises the probability of finding the instructions in cache is a combinatorial problem. In general, the techniques employed to solve combinatorial problems are characterised by looking for a solution from among many potential solutions. Petrank and Rawitz [16] showed that unless $P = NP$ there is no efficient optimised algorithm for data placement or code rearrangement that minimises the number of cache misses. Furthermore, it is not even possible to get close. Therefore, they conclude that the problem pertains to the class of extremely inapproximable optimisation problems and that, consequently, on one hand, it is necessary to use heuristics to tackle the problem, and on the other hand, it is not possible to estimate the potential benefits of an algorithm to reduce cache misses. So, the virtues of a given algorithm must be evaluated by comparing algorithms.

Hence, rather than trying to find only the best (optimal) solution, a good non-optimal (trade-off) solution is sought. Therefore, to solve the problem at hand, it may be a good idea to apply some form of directed search. For this kind of problem, one of the most appealing techniques is using genetic algorithms since they are generally seen as optimisation methods for non-linear functions.

In fact, in [8] a *Genetic Algorithm*, *GA*, has been proposed to solve an equivalent problem. The results presented there show that the use of a genetic algorithm to solve the problem represents a good choice since it provides for each task in the task set, not just the subset of blocks to be loaded, an estimation of the WCET and, the corresponding WCRT considering the estimated WCET, but also because that selection offers good performance. Moreover, results in [9] show that using the genetic algorithm proposed in [8] brings slightly better results than using the pragmatic algorithms given in [18].

In this work, for evaluation purposes, the following cache characteristics are assumed: A direct-mapped I-cache with varying size, a cache line size of 16 bytes (4 32-bit wide instructions); I-buffer is also 16 bytes wide. Fetching an instruction from I-cache or I-buffer takes 1 cycle while fetching an instruction from IM takes 10 cycles. A fixed-priority preemptive scheduler is used in every case. Task priority is assigned according to a Rate Monotonic Policy. Also, notice that in this work, it is assumed that the deadline, D, is equal to the task period, T.

Evaluation results concerning the proposed memory hierarchy must show whether the proposed memory architecture is predictable and if there is any performance loss when using the proposed memory hierarchy (LSM) in front of using a locking I-cache in a dynamic manner (dLC). Therefore, two kinds of results were evaluated to assess the merits of the proposed memory hierarchy. The first set of results is obtained by using a GA to select blocks and estimate processor utilisation when using those selected blocks with the proposed memory hierarchy (U_{LSMe}). The second set of results is obtained by using the same selected blocks and a modified version of SPIM (the freely available, widely used MIPS simulator) which embodies a cache simulator, to execute one hyperperiod of the task set and thus obtain the simulated processor utilisation (U_{LSMs}).

It is not easy to compare the performance of a real-time system running on different architectures. If the same task set is schedulable in every case, there are many characteristics and metrics useful to compare performance. Furthermore, it is highly desirable to use standard benchmark(s) to evaluate the predictability and performance of the proposed memory hierarchy since it makes possible the comparison with other approaches.

Traditional computing benchmarks are inadequate for characterising real-time systems since they are not de-

Table 1. Main characteristics of task sets and cache sizes

Feature	Minimum	Maximum
Number of tasks	3	8
Task Size	1.6 KB	27.6 KB
Task Set Size	12.5 KB	57.6 KB
Instr. executed per task (approx.)	50,000	8,000,000
Instr. executed per tasks (approx.)	200,000	10,000,000
Cache Size	1 KB	32 KB

signed to exhibit behaviour characteristic of such systems, such as periodic, transient and transient periodic activation/deactivation. On the other hand, there are several proposals for embedded/real-time systems benchmarking. Unfortunately, however, the lack of consensus about using a standard benchmark (to the authors' best knowledge) precludes the use of such proposals given that, in general, they are not easily portable. Moreover, it is necessary to notice that the proposed benchmarks are not targeted to measure cache memory effects in real-time systems since they do not cause preemptions[20].

The 26 tasks sets used in this work come from [10]. The code for each task is synthetic; it does nothing useful but it has a mix of instructions such that it is easy to automatically generate different programs, which is adequate for the purpose: each task may have streamlined code, single loops, up to three nested loops, if-then-else constructs.

Table 1 summarises some characteristics of the task sets and cache sizes employed for evaluation purposes.

4.1 Predictability analysis

To verify how predictable the proposed memory hierarchy is, the GA estimated response time of every task in the task set, R_{LSMe}, was compared with the corresponding response time obtained through the simulation R_{LSMs}.

However, instead of using the individual response times for each task, τ_i, in every task set, Processor Utilisation, a measure that involves the whole TS will be used to illustrate the results in a more compact way:

$$U = \sum_{i=1}^{tasks} \frac{C''_i}{T_i} \qquad (11)$$

where C''_i, the computation time of τ_i includes all cache effects (intrinsic and extrinsic interference); i.e., it includes the time required for τ_i to reload the cache after preemptions:

$$C''_i = R_i - \sum_{\forall\, \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C'_j \qquad (12)$$

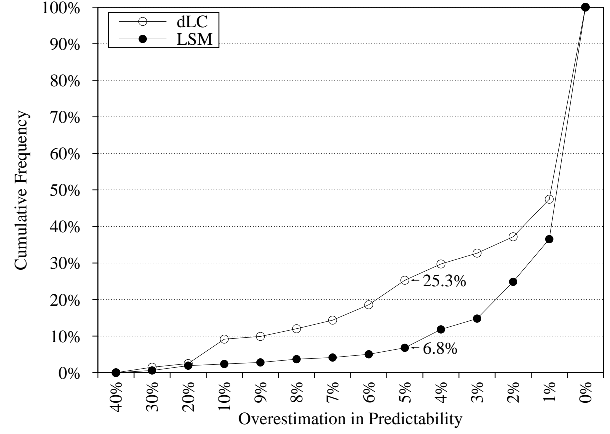


Figure 3. Cumulative frequency curves for the overestimation in predictability

where R_i is the WCRT for τ_i. Since R_i includes not just the CRPD but also the execution time of those tasks with a higher priority than τ_i, it is necessary to deduct the execution time for those tasks.

Then, given the proposed memory hierarchy, the utilisation estimated by the GA (U_{LSMe}), and the utilisation obtained through the simulation (U_{LSMs}), the overestimation in predictability, Ω, is given by $\Omega = U_{LSMe}/U_{LSMs}$. Figure 3 presents the cumulative frequencies for the overestimation when using the proposed memory hierarchy and the dynamic use of locking cache. Cumulative frequencies represent the number of responses in the data set falling into that class or a lower class [11].

The results verify that the proposed memory hierarchy is predictable:

- For every task in the whole set of tasks (676), the estimated response time is always larger than the simulated one ($R_{LSMe} > R_{LSMs}$).
- In the same vein, for every task set, the estimated utilisation is always larger than the one obtained through the simulation ($U_{LSMe} > U_{LSMs}$);
- Furthermore, as can be seen in Figure 3, the proposed memory hierarchy (LSM) provides better predictability than that obtained with the dynamic use of locking cache (dLC). It can be observed that when using the proposed memory hierarchy, the overestimation in utilisation is greater than or equal to 5% in less than 7% of the cases. On the other hand, when employing the locking cache in a dynamic way, the overestimation in utilisation is greater than or equal to 5% in around 25% of the cases.

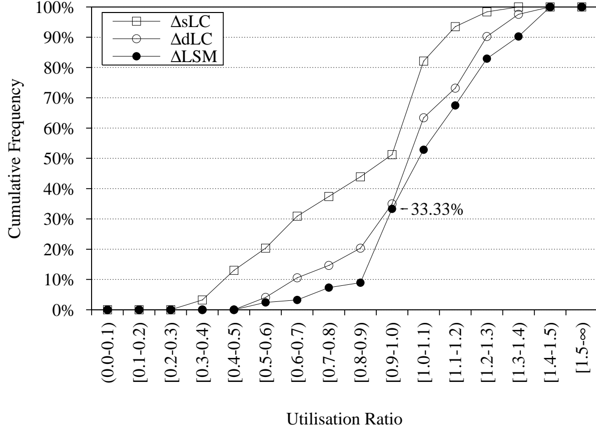


Figure 4. Cumulative frequency curves for the utilisation ratios

4.2 Performance evaluation

Although the effects of using the proposed memory hierarchy can be safely incorporated into the schedulability analysis, the performance advantages obtained from using the proposed memory hierarchy should be analysed.

Since a higher cache hit ratio does not necessarily guarantees that every task in the task set will satisfy its deadline, the approach used in this work to measure the quality of the solution is to use Processor Utilisation. The lower the processor utilisation, the better, since this means that the task set demands less CPU time and thus other tasks might be included in the task set while the system remains schedulable (i.e., all tasks executing on time).

Estimated Processor Utilisations for the system with an LSM (U_{LSMe}), the system with an LC used in a static manner (U_{sLCe}), and the system with an LC used in a dynamic manner (U_{dLCe}), were calculated by using the same GA for block selection and the appropriate I-cache refill penalty.

Afterwards, the different utilisations were normalised against the utilisation obtained when simulating the system with a conventional cache, U_{Cs}, to obtain the Utilisation Ratios ($\Delta U_X = U_{Xe}/U_{Cs}$, where $_X$ is one of $_{LSM}$, $_{dLC}$, $_{sLC}$).

Figure 4 shows that:

- In less than 34% of the cases, $\Delta U_{LSM} > 1$; i.e., $U_{LSMe} > U_{Cs}$, and hence, the proposed memory hierarchy brings about the same or better processor utilisation than that obtained when using a conventional cache in around 66% of the cases;

- In every range, $\Delta U_{LSM} < \Delta U_{dLC} < \Delta U_{sLC}$ and hence, the proposed memory hierarchy brings better processor utilisation than using a locking cache in a dynamic manner; moreover, in the zone with losses (range $[0, 1.0)$), the proposed memory hierarchy provides lower losses and in the zone with gains (range $(1.0, \infty)$), the proposed memory hierarchy provides higher gains.

Furthermore, a statistical analysis of three null hypothesis tests (t-test, sign test, and signed rank test) was done to corroborate that $\Delta U_{LSM} - \Delta U_{dLC} < 0$ (i.e., that the proposed memory hierarchy provides a better Processor Utilisation than the dynamic use of locking cache). The first one establishes whether the mean is zero or not; the remaining two tests allow to determine whether the median is zero or not. The sign test is based on counting the number of values above and below the hypothesized median, while the signed rank test is based on comparing the average ranks of values above and below the hypothesized median. All of the three tests revealed that $\Delta U_{LSM} < \Delta U_{dLC}$ at the 95% confidence level.

5 Concluding remarks

By virtue of including the LSM, any I-cache is transformed into a virtual locking I-cache, independently of its size, associativity and block size, the three main organisation parameters in a cache memory. In addition, parameters like I-cache replacement policy are irrelevant, provided that the algorithm used to select I-cache contents guarantees that there will be no conflict misses.

Results show that the proposed memory hierarchy is predictable and simple to analyse. Moreover, when compared to dynamic use of locking cache, it offers (i) a lower overestimation in predictability; and (ii) a higher performance. Finally, when compared to a conventional cache, in many cases the proposed memory hierarchy performs better or very close to it.

On the other hand, the proposed memory hierarchy does not needs explicit management of the memory hierarchy at run-time, while both scratchpad memories and locking cache memories, do. Moreover, the use of scratchpad memories requires explicit modifications in the application code's control flow.

In short, the memory assist is versatile in its operational aspects, yet it uses generic components; it does not cause any extra overheads to the system; its impact on system programming is negligible; and, it may be embedded in System-on-a-Programmable-Chip designs targeted to current FPGAs, while contributing in a significant way to determinism and performance improvements with respect to dynamic use of a locking I-cache.

All of these advantages are obtained at a fraction of the cost of the original system, thus paving the way to widespread use in realistic real-time systems.

References

[1] Alt M., Ferdinand C., Martin F., and Wilhelm R. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science (LNCS)*, 1145, Sept. 1996.

[2] Arnaud A. and Puaut I. Dynamic instruction cache locking in hard real-time Systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RTNS'06)*, pages 179–188, May 2006.

[3] Jacob B. L. and Bhattacharyya S. S. Real-time memory management: Compile-time techniques and run-time mechanisms that enable the use of caches in real-time systems. Technical report, Institute for Advanced Computer Studies, University of Maryland at College Park, USA, Sept. 2000.

[4] Jain P., Devadas S., Engels D. W., and Rudolph L. Software-assisted cache replacement mechanisms for embedded systems. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, Nov. 2001.

[5] Kirk D. B. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proc. of the 10th IEEE Real-Time Systems Symposium*, pages 229–237, Dec. 1989.

[6] Li Y.-T. S., Malik S., and Wolfe A. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263, Dec. 1996.

[7] Lundqvist T. and Stenstrom P. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2–3):183–207, Nov. 1999.

[8] Martí Campoy A., Pérez Jiménez A., Perles Ivars A., and Busquets Mataix J. V. Using genetic algorithms in content selection for locking-caches. In *Proc. of the IASTED International Symposia Applied Informatics*, pages 271–276. Acta Press, Feb. 2001.

[9] Martí Campoy A., Puaut I., Perles Ivars A., and Busquets Mataix J. V. Cache contents selection for statically-locked instruction caches: an algorithm comparison. In *Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 49–56, July 2005.

[10] Martí Campoy A., Tamura E., Sáez S., Rodríguez F., and Busquets-Mataix J. V. On using locking caches in embedded real-time systems. In *Proc. of the 2nd International Conference on Embedded Software and Systems (ICESS-2005). Lecture Notes in Computer Science (LNCS) vol. 3820*, pages 150–159, Dec. 2005.

[11] G. McPherson. *Applying and Interpreting Statistics: A Comprehensive Guide*. Springer Texts in Statistics. Springer-Verlag New York, Inc., second edition, 2001.

[12] Mueller F. Compiler support for software-based cache partitioning. In *LCTES'95: Proc. of the ACM SIGPLAN 1995 workshop on Languages, Compilers, & Tools for real-time Systems*, pages 125–133, June 1995.

[13] Mueller F. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[14] Muller H., May D., Irwin J., and Page D. Novel caches for predictable computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, Oct. 1998.

[15] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, third edition, 2 Aug. 2004.

[16] Petrank E. and Rawitz D. The hardness of cache conscious data placement. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 101–112, 2002.

[17] Puaut I. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 217–226, July 2006.

[18] Puaut I. and Decotigny D. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 114–123, Dec. 2002.

[19] Sasinowski J. E. and Strosnider J. K. A dynamic-programming algorithm for cache memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, Aug. 1993.

[20] Sebek F. Measuring cache related pre-emption delay on a multiprocessor real-time system. In *IEE/IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, Dec. 2001.

[21] Shaw A. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[22] Staschulat J., Schliecker S., and Ernst R. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 41–48, July 2005.

[23] Tamura E., Rodríguez F., Busquets-Mataix J. V., and Martí Campoy A. High performance memory architectures with dynamic locking cache for real-time systems. In *Proc. of the Work-In-Progress Session of the 16th Euromicro Conference on Real-Time Systems (WIP ECRTS'04). TR-UNL-CSE-2004-0010, Department of Computer Science and Engineering. University of Nebraska-Lincoln*, pages 1–4, June 2004.

[24] Tan Y. and Mooney V. A prioritized cache for multi-tasking real-time systems. In *Proc. of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'03)*, pages 168–175, Apr. 2003.

[25] Vera X., Lisper B., and Xue J. Data cache locking for higher program predictability. In *Proc. of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–282, June 2003.

[26] Wehmeyer L. and Marwedel P. Influence of onchip scratchpad memories on WCET prediction. In *Proc. of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 29–32, June 2004.

[27] Wehmeyer L. and Marwedel P. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, pages 600–605, Mar. 2005.

[28] Wolfe A. Software-based cache partitioning for real-time applications. In *Proc. of the 3rd Workshop on Responsive Computer Systems*, Sept. 1993.

On the sensitivity of WCET estimates to the variability of basic blocks execution times

Hugues Cassé[a], Christine Rochange[a], Pascal Sainrat[a,b]
[a]Institut de Recherche en Informatique de Toulouse, [b]HiPEAC NoE
Université Toulouse III – Paul Sabatier
31062 Toulouse cedex 9, France
{casse, rochange, sainrat}@irit.fr

Abstract

The Implicit Path Enumeration Technique (IPET) is a very popular approach to evaluate the Worst-Case Execution Time (WCET) of hard real-time applications. It computes the execution time of an execution path as the sum of the execution times of the basic blocks weighted by their respective execution counts. Several techniques to estimate the execution time of a block taking into account every possible prefix path have been proposed: the maximum value of the block execution time is then used for IPET calculation. The first purpose of this paper is to analyze the sensitivity of block execution times to prefix paths. Then we show how expanding the IPET model to consider the execution times related to different contexts for each basic block improves the accuracy of WCET estimates.

1. Introduction

In hard real-time systems, the Worst-Case Execution Times (WCETs) of critical tasks have to be estimated as accurately as possible either to analyze the schedulability or to determine a static schedule that makes it possible for every task to meet its deadline. Several approaches to WCET estimation have been investigated these last fifteen years and the problem has been shown to be more and more complex as the architecture of processors is enhanced to provide better performance. Researchers recommend the use of simple hardware but even so the tendency is to consider high-performance processors in order to fit increasing performance requirements: the tasks to be executed are not necessarily more complex but, in the context of some approaches like IMA (Integrated Modular Avionics) or AUTOSAR (Automotive Open System ARchitecture), a single processing node should support several tasks.

Static methods for WCET analysis are usually preferred to measurements because they get round the need of examining every possible execution path. They roughly consist in adding the execution times of the basic blocks along execution paths. To be safe, they must ensure that the execution times of basic blocks are not under-estimated. When considering modern processors, this requires to include the impact of the context (i.e. the execution history) when analyzing the pipelined execution of a block. The result is the maximum execution time (or cost) of the block and is used to compute the WCET of the whole program.

Our purpose, in this paper, is to show that the execution time of a block depends sharply on the context (prefix path) and that the whole WCET would be estimated more tightly if different execution times (for different contexts) were considered for each basic block. In a second step, we introduce context-sensible data into the IPET model. Experimental results show that this helps in getting more accurate WCET estimates. The sensitivity of these results to architectural parameters (scalar/superscalar pipeline, static/dynamic instruction scheduling, size of the instruction window) is also investigated.

The paper is organized as follows. Section 2 provides some background information on timing analysis and pipeline modeling. In Section 3, we discuss experimental results that show the sensitivity to the context of block timings (experimental conditions are detailed in the Appendix). The benefits (in terms of tightness of WCET estimates) of extending the IPET model to include context-related block execution times are shown in Section 4. Section 5 concludes the paper.

2. Background: pipeline modeling for WCET estimation

Modern processors cut the processing of an instruction into several steps that are handled in a pipelined manner: in the absence of stalls, instruction *i* processes through step s in the same time as instruction *i-1* processes through step *s-1* (Figure 1 shows a typical processor pipeline). This means that, while the execution time of a single instruction is the sum of the latencies of all steps, the execution time of a sequence of instructions

is shorter than the sum of their individual execution times due to the overlap in the pipeline. In theory, the execution time of a sequence of N instructions in a P-stage pipeline (each stage having a single-cycle latency) should be $P+N-1$. In practice, the effective execution time would be longer due to stalls that result from resource conflicts and inter-instruction data dependencies.

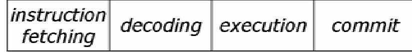


Figure 1. A typical 4-stage processor pipeline.

As said before, static WCET evaluation techniques compute the execution time of a program from the individual execution times of its basic blocks. To estimate the execution time of a basic block tightly, one must take into account the overlap of instructions in the pipeline. When a basic block is fetched in a pipelined processor, the pipeline is generally not empty and still contains instructions from the previous block(s). Then, two kind of effects should be taken into account in order to get an accurate WCET estimate. First, the instructions of the previous blocks use some resources and might be responsible for delaying the processing of the evaluated basic block. This is what we call the sensitivity to the context. Second, the blocks that are simultaneously present in the pipeline overlap, which reduces the execution time of the sequence.

In this section, we will review the various approaches that have been proposed to estimate the execution times of basic blocks in a pipeline. We will first show how the possible interferences between basic blocks can be accounted for while computing the WCET of a program. Then, we will explain how the behavior of a pipelined processor can be modeled and how the interferences between basic blocks can be evaluated.

2.1. Execution times and interferences between basic blocks

In this section, we will show how it is possible to evaluate the contribution of a basic block to the execution time of an execution path it belongs to.

A conservative approach to get an upper bound of this contribution consists in ignoring the overlap of successive blocks in the pipeline and in considering the full execution time of the block in an empty pipeline (i.e. the time between the first instruction is fetched and the last instruction is committed). Then the execution time of the path is computed as the sum of the individual execution times of its blocks. This is illustrated in Figure 2. While being safe, this approach clearly overestimates the execution time.

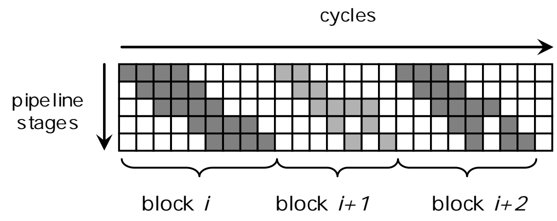


Figure 2. Conservative model to evaluate the execution time of a sequence of blocks.

To take into account the overlap of successive blocks in the pipeline, every possible initial context should be considered to derive time estimates for a given basic block. However, not only the overlap but also the possible block interferences must be analyzed. According to the pipeline characteristics, these interferences might lengthen or shorten the block execution time (in a dynamically-scheduled processor, timing anomalies can reduce or increase the final execution time [13]).

Assuming that the interferences can be properly analyzed (possible approaches will be reviewed in the next section), Figure 3 shows how they can be expressed. In this Figure, t_B is the execution time of block B when it is executed in an empty pipeline while $t_{B/A}$ is the execution time of B when it is executed after A. The cost $c_{B/A}$ is the contribution of block B to the execution time of the sequence [A-B]. The execution time of sequence [A-B] can be written as:

$$t_{[A-B]} = t_A + c_{B/A}$$

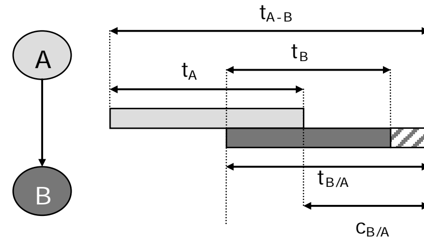


Figure 3. Expressing block interferences and overlapping.

Initially, users of the IPET method [11] to estimate the Worst-Case Execution Time of a program in a pipelined processor did not refer to the cost of blocks but to pipeline gains. The execution time of sequence [A-B] was expressed as:

$$t_{[A-B]} = t_A + t_B + \delta_{[A-B]}$$

which is the same ($c_{B/A}$ stands for $t_B + \delta_{[A-B]}$).

Early implementations of the IPET method did only consider 2-block sequences to determine the costs of blocks. In 2002, Engblom showed that this approach was no longer valid when considering modern processors [4].

To improve performance, modern processors often implement some mechanisms that might be a source of interferences between distant basic blocks (not only adjacent ones). Such features include superscalar execution, dynamic instruction scheduling, long-latency functional units, etc. Figure 4 illustrates how interferences between distant blocks may impact the execution time of a sequence. The execution time of sequence [*A*-*B*-*C*] should be computed as:

$$t_{[A-B-C]} = t_A + c_{B/A} + c_{C/[A-B]}$$

where $c_{C/[A-B]}$ might be shorter than, equal to or longer than $c_{C/B}$ due to the possible interferences from block A.

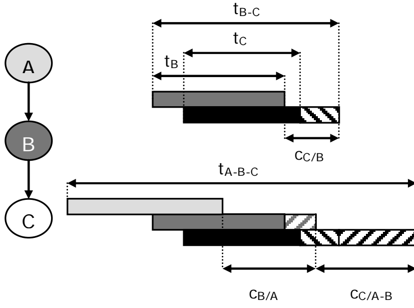


Figure 4. Long timing effects.

The main difficulty comes from the fact that the span of block interferences cannot be bounded, which means that the cost of a basic block on an execution path might be impacted by any other block on the path. In practice, the cost of a block only depends on the few preceding blocks but, for the sake of safety, a possible effect from very distant blocks must be imagined. Existing approaches to take all possible contexts into account to evaluate the cost of a block will be presented in the next section.

2.2. Pipeline model

Early contributions to pipeline modeling made use of reservation tables to find out how every instruction of a basic block would process through the pipeline [10][8]. A reservation table is a simple means to represent the use of the processor internal resources (pipeline stages, functional units, etc.) but its limited semantics is not sufficient to express the behavior of superscalar and dynamically-scheduled processors.

Cycle-level simulators (e.g. SimpleScalar [1]) make it possible to accurately determine the execution time of a sequence of blocks and the cost of each block in the sequence. Unfortunately, the number of possible prefix paths for a basic block in a real-life application is generally huge and it is not possible to simulate each of them. Then it is possible to determine the cost of a block for a set of short possible prefix paths, but not the *maximum* cost since all the possible prefixes cannot be considered.

When all the possible prefix paths cannot be considered one by one, the solution comes from static analysis techniques. The aiT tool of the AbsInt company [1] uses abstract interpretation to define an abstract pipeline state at the beginning of each basic block that includes every possible concrete state [15][16]. It then simulates the execution of the block on this input abstract pipeline state to determine an output abstract state which is propagated to the successors of the block. The tool iterates until a fix point has been found. Then execution times of basic blocks can be derived and included in the ILP (IPET) model to estimate the WCET of the whole program.

Recently, Li *et al.* [12] introduced execution graphs as a means for estimating the worst-case costs of basic blocks. An execution graph expresses precedence constraints between the processing steps of the instructions of a block and computes earliest and latest values of their respective finish times. Some preceding instructions (prologue) can be included in the graph to take the context into account and very conservative hypotheses are made on earlier instructions so that the computed cost is guaranteed to be an upper bound of the real possible costs, whatever the prefix path is. Again, this worst-case cost is used to compute the whole program WCET.

The differences between these two methods reside in the accuracy of the contexts considered for each basic block. With abstract interpretation, the input abstract pipeline state of a block results from the evaluation of effective prefix paths. On the contrary, the execution-graph method examines real prefixes until a given depth (which equals the instruction-window size) and considers that the previous instructions can be any. As a consequence, the overestimation of the costs is higher, which is the price of shorter analysis times.

3. On the sensitivity of block timings to the context

The experimental results reported in this paper were collected in the conditions detailed in the Appendix.

As explained in the previous section, static methods for pipeline modelling derive the worst-case cost of basic blocks. Our purpose here is to analyze the variability of the cost when different contexts (i.e. the prefix paths) are considered.

3.1. Context-related costs

For each basic block, we have examined the different costs when d-block deep contexts are considered, with $0 \leq d \leq 4$. In the rest of this paper, we will refer to the cost of a block evaluated by considering a d-block long prefix of the block as a d-cost.

At each depth level, the cost of a block would be estimated conservatively by a static analysis approach as the ones described in section 2.2. This estimate would be greater than or equal to the maximum of the costs evaluated at the next deeper level. In this work, we have evaluated all the possible costs of each block considering all its possible 8-block long prefixes (see the Appendix for details about the methodology). Then i-cost ($0 \le i \le 4$) of the block was computed as the maximum value of the corresponding ($i+1$)-costs.

To illustrate this, Table 1 lists the different costs of block A in the example CFG given in Figure 5 (in this example, the length of contexts is limited to three blocks). Block A has four possible 3-block contexts ([D_1-C_1-B_1], [D_2-C_2-B_1], [D_3-C_3-B_2], [D_4-C_3-B_2]). In the table, the third column shows the number of different paths represented by each cost. For example, $c_{A/[D_1-C_1-B_1]}$ is the maximum value of the costs of A observed in the two possible paths [F_1-E_1-D_1-C_1-B_1-A] and [F_1-D_1-C_1-B_1-A]. On a mean, each 3-cost represents 1.5 real cost values. We assume that paths [F_1-D_3-C_3-B_2-A] and [F_1-D_4-C_3-B_2-A] have the same cost value for block A. Then $c_{A/[C_3-B_2]}$ represents two paths but a single cost value. At the end, c_A represents five different values and is assigned the highest of them. Using c_A to compute the program WCET (instead of distinguishing between c_{A/B_1} and c_{A/B_2}) is a source of overestimation.

Context depth	Costs	# represented paths/values	mean # represented values
3	$c_{A/[D_1-C_1-B_1]}$	2 / 2	1.5
	$c_{A/[D_2-C_1-B_1]}$	2 / 2	
	$c_{A/[D_3-C_3-B_2]}$	1 / 1	
	$c_{A/[D_4-C_3-B_2]}$	1 / 1	
2	$c_{A/[C_1-B_1]}$	2 / 2	1.67
	$c_{A/[C_2-B_1]}$	2 / 2	
	$c_{A/[C_3-B_2]}$	2 / 1	
1	c_{A/B_1}	4 / 4	2.5
	c_{A/B_2}	2 / 1	
0	c_A	6 / 5	5

Table 1. Context depth and number of represented cost values (see Figure 5)

3.2. Variability of block execution times

For each context depth d, we have recorded the mean number of different represented cost values over the set B of basic blocks in the program (computed with the formula below, where $P_{b(d)}$ is the set of possible d-block long prefixes of block b and A_p is the set of possible paths before prefix p).

$$\frac{1}{|B|}\sum_{b\in B}\left(\frac{1}{|P|}\sum_{p\in P_{b(d)}}\left(\frac{1}{|A_p|}\sum_{a\in A_p} c_{b/[a-p]}\right)\right)$$

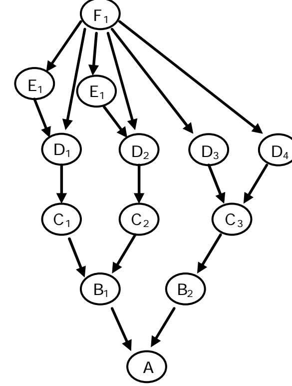


Figure 5. Example CFG.

In this section, we consider the 2-way superscalar processor configuration given in the Appendix. Results for each benchmark are given in Figure 6. As expected, the mean number of represented cost values decreases when deeper contexts are considered. For most of the applications, the variability of costs is noticeable. The case of `jfdctint` is interesting because each of its blocks has a constant cost value whatever the context depth is. This program contains three loops executed in sequence, and each loop has a long (up to 89 instructions) single-block body. These three blocks are the only ones to have two possible contexts and, since they are long, the impact of these contexts is not visible at the end of the blocks.

Considering the maximum of a set of possible costs values instead of each cost value individually might be detrimental to the accuracy of WCET estimates if the values differ noticeably. Table 2 gives additional information about the maximum number of represented cost values (observed over the set of blocks) and about the mean and maximum gap between the different cost values of a block. We observe that several benchmarks (`fft1`, `fir`, `lms`, `minver`, `qurt`) have at least one block for which the 0-cost represents more than 7 values. Moreover, the gap between cost values can be as long as 10 cycles (`minver`). This is likely to impact the whole WCET since a block that has many different timings probably belongs to a loop and might be executed many times on the worst-case path. Note that the maximum number of represented values and the maximum gap between them remains large for 4-costs (even if their means is lower). This indicates that some blocks are impacted by long timing effects.

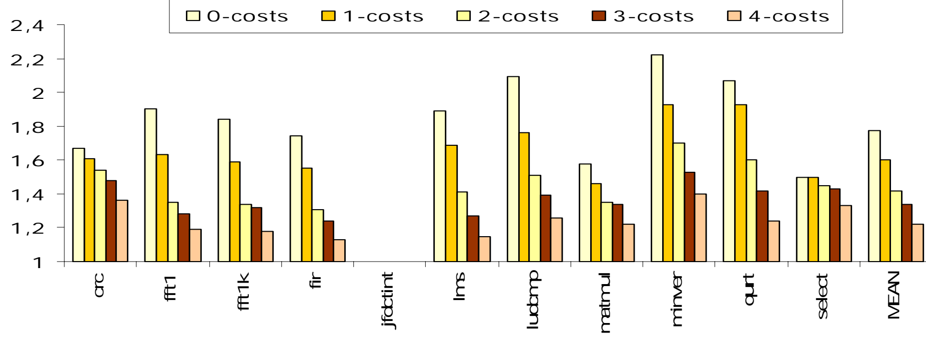


Figure 6. Variability of the costs of blocks (2-way superscalar processor)

	0-costs			4-costs		
benchmarks	**max. # values**	**mean gap**	**max. gap**	**max. # values**	**mean gap**	**max. gap**
crc	3	0.71	3	3	0.43	3
fft1	8	1.16	8	6	0.23	5
fft1k	5	1.02	5	4	0.22	4
fir	7	0.88	7	6	0.16	8
jfdctint	2	0.22	1	1	0	0
lms	7	1.09	8	6	0.2	7
ludcmp	6	1.3	7	4	0.3	4
matmul	3	0.67	3	2	0.22	1
minver	11	1.35	10	6	0.39	10
qurt	7	1.16	8	6	0.25	7
select	4	0.64	4	3	0.41	3

Table 2. Variation of the cost values represented by each 0-cost and 4-cost (2-way processor)

4. Context-sensitivity of blocks timings and WCET estimates accuracy

4.1. Including context-related timings in the IPET model

As said before, the execution time of an execution path is usually computed as:

$$T = \sum_{b \in B} x_b . c_b$$

where B is the set of blocks along the paths, xb is the execution count of block b in the path and c_b is the maximum contribution of block b to the execution time (c_b is evaluated taking any possible context into account).

This estimation can be refined by using deeper costs. With depth d, $x_b.c_b$ can be expanded into:

$$x_b . c_b = \sum_{p \in P_{b(d)}} x_{b/p} . c_{b/p}$$

where $P_{b(d)}$ is the set of d-block prefixes of block b.

As shown in earlier work by Ermedahl [7], a set of constraints must be added to the IPET model to bound the execution count of each prefix path. For each block, and for each path p in the set P of the possible prefixes of the block, with p being the sequence of blocks $[p_0$-p_1-...-$p_{n-1}]$, the constraints to be added are:

$$x_b = \sum_{p \in P_{b(d)}} x_{b/p}$$

$$\forall i, 0 \le i < n-1, \ x_{b/p} \le x_{[p_i - p_{i+1}]}$$

where $x_{[p_i - p_{i+1}]}$ is the execution count of sequence $[p_i$-$p_{i+1}]$.

4.2. Experimental results: improvement of WCET estimates

Figure 7 shows how the WCET can be improved when d-costs are used (the gain is computed against the WCET estimated with 0-costs). On a mean, the WCET estimation is improved by 5.5% with 1-costs, by 7% with 2-costs, by 10% with 3-costs and by 11% with 4-costs. Some of the benchmarks (`jfdctint`, `crc`, `fft1`, `select`) do not exhibit noticeable improvements. This was expected for `jfdctint` since the number of contexts is very limited. Other benchmarks (`fftk1`, `ludcmp`, `matmul`, `minver`, `qurt`) benefit from higher gains.

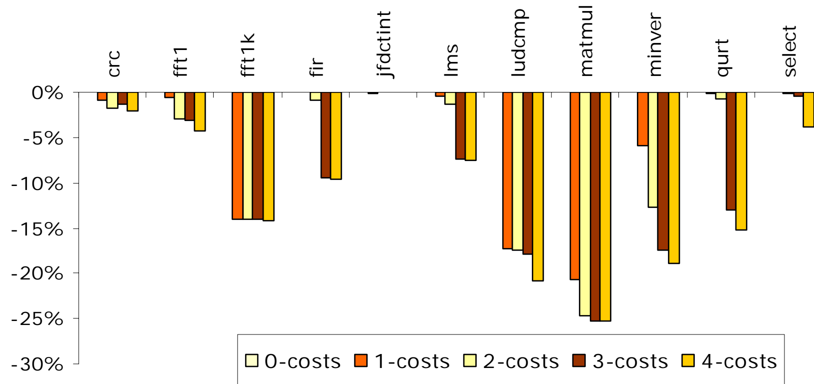


Figure 7. Improvement of the estimated WCET with deeper analysis (2-way superscalar).

4.3. Impact of the architecture parameters on the WCET improvement

In this section, we analyze the impact of the processor parameters on the WCET improvement obtained by using 4-costs. Figure 8 shows how the gains (against a WCET estimated with 0-costs) vary with the pipeline width (with out-of-order execution). It can be observed that very small gains are to be expected from considering deeper contexts for a scalar processor (only 2.3% on a mean). But the gains increase rapidly with the pipeline width. For several benchmarks, the improvement is higher than 20% for a 4-way processor (up to 43% for `ludcmp`) which is considerable. For these benchmarks, we have recorded up to 15-cycle gaps between the cost values represented by some 0-costs. This means that the execution time of a block is very dependent on the execution history. This is not surprising since a large pipeline with dynamic instruction scheduling can produce a large number of instructions interleaves, which generates inter-block effects.

In Figure 9, we observe the impact of the window size on the results. Measurements were made for a 4-way out-of-order processor, with different reorder buffer sizes, and the WCET was computed with 0-costs and 4-costs (the diagram plots the improvement). For most of the benchmarks, better gains are obtained when the instruction window is deeper. This is not true for `matmul` and select: they include some repeated small loops that do not fit well in a small reorder buffer (which generates many possible contexts for some blocks). This is why they exhibit high gains with a small instruction window.

Finally, we have considered in-order pipelines. The results given in Figure 10 were obtained for a 4-way processor with a 32-instruction window. As it could be expected, it appears that taking deeper contexts into account does only slightly improve WCET estimates for a statically-scheduled processor. This is due to the fact that the number of possible instruction interleaves in the pipeline is smaller, which limits the number of possible cost values for a basic block.

5. Conclusion

Estimating the WCET of a program using a static approach requires evaluating the individual execution times of basic blocks. When executing on a high-performance processor, the execution time of a basic block is likely to depend on the execution history (also referred to as the context). Some techniques to take into account all the possible contexts have been proposed and they provide the maximum execution time (or cost) of each basic block.

In this paper, we have presented some experimental results that show how much block execution times are sensitive to the context. For most of the benchmarks, some blocks have many different cost values related to different possible prefix paths (as many as 11 values) and the gap between the minimum and maximum value is large (up to 10 cycles). Considering the maximum value instead of distinguishing each of them is likely to lead to a large over-estimation of the WCET.

We have shown how taking context-related execution times could improve the tightness of WCET estimates. Experimental results prove that considering 4-block contexts tightens the WCET estimates by up to 25% for a 2-way superscalar out-of-order processor (11% on average) and up to 43% (18% on average) for a 4-way processor. As far as we know, most of the existing tools (e.g. aiT) mainly consider 1-costs (even if they might use loop unrolling techniques to take one part of the context-sensitivity into account. Our results show that considering deeper contexts is beneficial for most of the applications. Additional results show that the gain is slightly lower when the instruction window (reorder buffer) is less deep (4-way out-of-order processor) and that is negligible when instructions are scheduled in order.

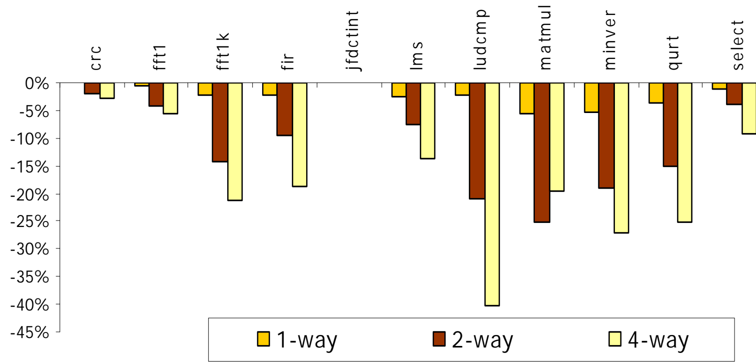


Figure 8. Improvement of the estimated WCET as a function of the pipeline width.

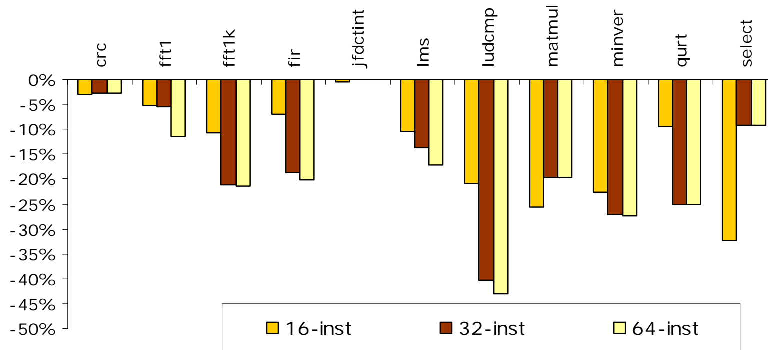


Figure 9. Impact of the ROB size on the WCET improvement with 4-costs (4-way out-of-order).

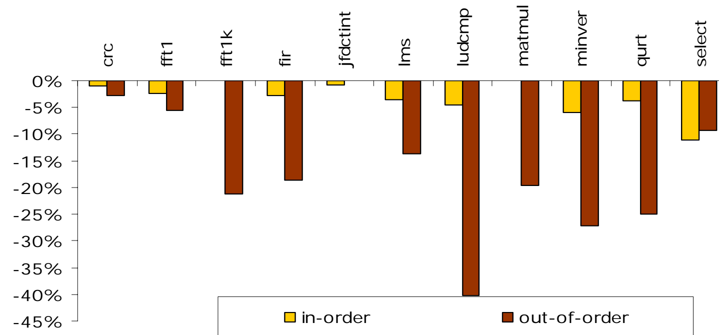


Figure 10. Impact of the scheduling policy on the WCET improvement with 4-costs (4-way superscalar, 32-inst. ROB)

References

[1] http://www.absint.com

[2] T. Austin, E. Larson, D. Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*, IEEE Computer, 35(2), 2002.

[3] H. Cassé, P. Sainrat, *OTAWA, a framework for experimenting WCET computations*, 3rd European Cong. on Embedded Real-Time Software, 2006.

[4] J. Engblom, *Processor Pipelines and and Static Worst-Case Execution Time Analysis*, Ph.D. thesis, University of Uppsala, 2002.

[5] J. Engblom, A. Ermedahl, *Pipeline Timing Analysis using a Trace-Driven Simulator*, 6th Int'l Conference on Real-Time Computing Systems and Applications (RTCSA), 1999.

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, H. Hansson, *Towards Industry-Strength Worst Case Execution Time Analysis*, ASTEC 99/02 Report, 1999.

[7] A. Ermedahl, *A Modular Tool Architecture for Worst-Case Execution Time Analysis*, Ph.D. thesis, Uppsala University, 2003.

[8] C. Healy, R. Arnold, F. Mueller, D. Whalley, M. Harmon, *Bounding pipeline and instruction*

cache performance, IEEE Transactions on Computers 48(1), 1999.
[9] Y. Hur, Y.H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S.-L. Min, C.Y. Park, H. Shin, C.S. Kim, *Worst case timing analysis of RISC processors: R3000/R3010 case study*, IEEE Real-Time Systems Symposium, 1995.
[10] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, C. S. Kim, *An accurate worst case timing analysis technique for RISC processors,* Real-Time Systems Symposium, 1994.
[11] Y.-T. S. Li, S. Malik, *Performance Analysis of Embedded Software using Implicit Path Enumeration,* Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.
[12] X. Li, A. Roychoudhury, T. Mitra, *Modeling out-of-order processors for WCET analysis*, Real-Time Systems, 34(3), 2006
[13] T. Lundqvist, P. Stenström, *Timing Anomalies in Dynamically Scheduled Microprocessors*, IEEE Real-Time Systems Symposium (RTSS), 1999.
[14] http://archi.snu.ac.kr/realtime/benchmark/
[15] S. Thesing, *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, PhD thesis, Universität des Saarlandes, 2004.
[16] H. Theiling, C. Ferdinand, *Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis*, 19th IEEE Real-Time Systems Symposium (RTSS), 1998.

Appendix

The experiments reported in this paper were carried out using the OTAWA framework [3]. OTAWA[1] implements an infrastructure to support several kinds of analyses that can be combined to estimate the WCET of an application. In this work, we used a basic flow analyzer (it builds the CFG from the object code and retrieves user-specified flow facts), a cycle-level simulator (it estimates the execution times of sequences of blocks) and a module that generates the constraints for WCET estimation with IPET and calls an ILP solver.

Evaluating the costs of basic blocks from the simulation of sequences of limited length is questionable since some long timing effects might not be captured this way. However, as mentioned in Section 2, block interferences do not span over more than a few blocks in practice. Then, the costs considered in this work cannot be guaranteed as 100%-safe but can be thought as very close to the real costs.

The simulator models a pipelined processor and accepts different parameters: pipeline width, instruction-scheduling policy (in-order/out-of-order), fetch queue and reorder buffer sizes, functional units parameters (width, latency, pipeline). It includes perfect caches (every access is a hit) and a perfect branch predictor. The configurations used in this work are summarized in Table 3. The simulator accepts PowerPC object code as input.

		CONFIGURATIONS		
		1-way	**2-way**	**4-way**
instruction scheduling		out-of-order		
fetch queue size		4	8	8
reorder buffer size		4	8	32
functional units	**latency**			
integer ALU	1	1	2	2
memory unit	2	1	1	1
fp ALU	6	1	1	1
multiply unit	3	1	1	1
divide unit	15	1	1	1

Table 3. Processor configurations

The benchmarks come from the SNU-suite [14] and are listed in Table 4. They were compiled to PowerPC code using gcc with the -O0 optimization level option.

	# blocks	**mean block length**	**Function**
crc	52	5	CRC (Cyclic Redundancy Check)
fft1	151	7	FFT (Fast Fourier Transform) using Cooly-Turkey algorithm
fft1k	48	8	FFT (Fast Fourier Transform) for 1K array of complex numbers
fir	94	7	FIR filter with Gaussian number generation
jfdctint	10	22	JPEG slow-but-accurate integer implementation of the forward DCT
lms	85	7	LMS adaptive signal enhancement
ludcmp	47	6	LU decomposition
matmul	14	4	Matrix product
minver	75	5	Matrix inversion
qurt	77	7	Root computation of quadratic equations
select	30	4	N-th largest number selection

Table 4. Benchmarks

[1] OTAWA is supported by the French Agence Nationale pour la Recherche (MasCoTte project)

Scheduling 1

Efficient computation of response time bounds under fixed-priority scheduling

Enrico Bini
Scuola Superiore Sant'Anna
Pisa, Italy
e.bini@sssup.it

Sanjoy K. Baruah
University of North Carolina
Chapel Hill, NC
baruah@cs.unc.edu

Abstract

All algorithms currently known for computing the response time of tasks scheduled under fixed-priority scheduling have run-time pseudo-polynomial in the representation of the task system. We derive a formula that can be computed in polynomial time for determining an upper bound on response times; our upper bound on response time has the added benefit of being continuous in the task system parameters. We evaluate the effectiveness of our approximation by a series of simulations; these simulations reveal some interesting properties of (exact) response time, which give rise to an open question that we pose as a conjecture.

Finally, the proposed upper bound of the response time can be used to test effectively the schedulablity of task sets in time linear with the number of tasks.

1. Introduction

In many real-time systems specific jobs are expected to complete by specified deadlines. Basically, two main categories of algorithms have been proposed for determining the response times of tasks in DM-scheduled systems: Rate Monotonic Analysis (RMA) [13] and Response Time Analysis (RTA) [10, 2].

RTA computes, for each task, the *worst-case response times* — the maximum amount of time that may elapse between the instant that a job is released for execution and the instant it completes execution. If, for all tasks, the response time is shorter than the deadline, then the task set is feasible. Instead, RMA searches, for each task, any instant earlier than the deadline, large enough to accommodate the computational requirement of the task itself and all the higher priority tasks. If such an instant exists for all tasks then the task set is feasible.

Both approaches are known to have pseudo-polynomial worst-case time complexity, and it is currently unknown whether the task set feasibility can be computed in time polynomial in the representation of the task system.

Despite the pseudo-polynomial time complexity, both RMA and RTA have very efficient implementations in practice that render them suitable for feasibility analysis of Fixed Priority (FP) systems. However, these algorithms may not be particularly well-suited for use in interactive real-time system design environments. When using such design environments, the system designer typically makes a large number of calls to a feasibility-analysis algorithm during a process of interactive system design and rapid system prototyping, since proposed designs are modified according to the feedback offered by the feasibility-analysis algorithm (and other analysis techniques). In such scenarios, a pseudo-polynomial algorithm for computing the task set feasibility may be unacceptably slow; instead, it may be acceptable to use a faster algorithm that provides an approximate, rather than exact, analysis.

Moreover, there are some circumstances in the real-time system design, such as in control systems [8] and in holistic analysis [19], where it is required to know the response time of the tasks, and not only the system feasibility provided by RMA. For this reason in this paper, we propose an algorithm for computing efficiently an approximate upper bound of the response time. In addition to computation efficiency, our algorithm has the benefit of representing the (bound on) response time as a continuous function of the task system parameters, thereby facilitating optimisation of system design in applications, such as some control systems, where task parameters may be tweaked locally without causing catastrophic changes to application semantics. (Response time is not in general a continuous function of system parameters; hence, no exact algorithm for computing response times can possibly make a similar guarantee.)

There are many scenarios in which efficient computation of (exact or approximate) response times is desirable.

- In distributed systems, tasks may be activated after the completion of some other task [22, 19]. In such cases it is necessary to know the response time of the first task in order to analyse the scheduling of the second. This task model is called *transaction model* [19], and the analysis is performed by means of the *holistic analy-*

sis [22].

- In control systems, the response time of a task measure the delay between the instant where the input are read from the sensors and the output are written to the actuators. The performance of the control system depends upon this value [8] hence the response time has a direct impact on the system performance. Moreover, as our provided bound of the response time is a differentiable function, it is possible to estimate the effect of the variation of any system parameter.

- Finally, when the relative deadline parameters are permitted to be larger than the periods, current algorithms for the exact computation of response time require the evaluation of the response times of each and every job within the busy period [12, 23]. The resulting complexity may be unacceptably high, especially in all those design environments where the response time routine is largely invoked.

1.1. Related work

The problem of reducing the time complexity of feasibility tests has been largely addressed by the real-time research community. The Rate Monotonic Analysis, after the first formulation by Lehoczky et al. [13], has been improved by Manabe and Aoyagi [17] who reduced the number of points where the time demand needs to be checked. Bini and Buttazzo [4] proposed a method to trade complexity vs. accuracy of the RMA feasibility tests.

The efforts in the simplification of the Response Time Analysis has been even stronger, probably due to the greater popularity of RTA. Sjödin and Hansson [21] proposed several lower bounds to the response time so that the original response time algorithm [10] could start further and the time spent in computing the response time is reduced. Bril [7] proposed a similar technique to reduce the time complexity of the exact RTA. Starting from the idea of Albers and Slomka [1], who developed an estimate of the demand bound function for EDF scheduled tasks, Fisher and Baruah [9] have derived a fully polynomial time approximation scheme (FPTAS) of the RTA. Very recently, Richard and Goossens [20] have extended the task model of a previous FPTAS [9] to take into account release jitter. Finally, Lu et al. [16] proposed a method to reduce the number of iterations for finding the task response times.

The remainder of this paper is organised as follows. In Section 2 we formally state our task model, and reduce the problem of bounding the response time of each task in a task system to a problem of bounding the total workload generated by the task system. In Section 3 we derive a bound on the workload, which immediately yields the desired response time bound. We describe a series of simulation experiments in Section 4 for determining the "goodness" of our upper bound. We conclude in Section 5 with a brief summary of the main results presented in this paper.

2. The Response Time Bound

We assume that a real-time system is modelled as being comprised of a pre-specified number n of independent *sporadic* tasks [18, 3] $\tau_1, \tau_2, \ldots, \tau_n$, executing upon a single shared preemptive processor. Each sporadic task τ_i is characterised by a worst-case execution time (WCET) C_i; a relative deadline parameter D_i; and a period/ minimum inter-arrival separation parameter T_i. Notice that the deadlines are arbitrary, meaning that no particular relationship is assumed between D_i and T_i. Each such task generates an infinite sequence of jobs, each with execution requirement at most C_i and deadline D_i time-units after its arrival, with the first job arriving at any time and subsequent successive arrivals separated by at least T_i time units. We assume that the system is scheduled using a fixed-priority (FP) scheduling algorithm such as the Deadline-Monotonic (DM) scheduling algorithm [14], which is known to be an optimal fixed-priority algorithm when all the sporadic tasks have their relative deadline parameters no larger than their periods.

We will use the term *utilisation* of τ_i (denoted by U_i), to represent the ratio C_i/T_i, and let U denote the *system utilisation*: $U = \sum_{i=1}^{n} U_i$. We assume that *tasks are indexed according to priorities*: task τ_1 is the highest-priority task, and τ_{i+1} has lower priority than τ_i for all i, $1 \leq i < n$. Notice that we do not assume any specific priority assignment.

We start with some notations and definitions. Let us define the *worst-case workload* as follows:

Definition 1 *Let $W_i(t)$ denote the* worst-case workload *of the i highest priority tasks over an interval of length t, which is the maximum amount of time that a task τ_j, with $1 \leq j \leq i$ can run over an interval of length t.*

As proved by Liu and Layland in their seminal paper [15], the worst-case workload $W_i(t)$ occurs when all the tasks $\tau_1, \ldots, \tau_i$ are simultaneously activated, and each task generates subsequent jobs as soon as legally permitted to do so (i.e., consecutive jobs of τ_i arrive exactly T_i time units apart, for all i) – this sequence of job arrivals is sometimes referred to as the *synchronous arrival sequence.* Thus, $W_i(t)$ equals the maximum amount of time for which the CPU may execute some task from among $\{\tau_1, \ldots, \tau_i\}$, over the time interval $[0, t)$, for the synchronous arrival sequence.

We highlight that our definition of worst-case workload is different than the *worst-case demand*, which is expressed by the "classical ceiling" expression $\sum_i \left\lceil \frac{t}{T_i} \right\rceil C_i$. The

worst-case workload is the fraction of the demand which can be executed in $[0, t)$, under the synchronous arrival sequence hypothesis, whereas the demand is the maximum amount of work which can be *demanded* in $[0, t)$.

A closely-related concept is that of the *worst-case idle time*:

Definition 2 *Let $H_i(t)$ denote the* worst-case idle time *of the i highest priority tasks over an interval of length t.*

This is the minimum amount of time that the CPU is not executing some task in $\{\tau_1, \ldots, \tau_i\}$ over the time interval $[0, t)$. It is straightforward to observe that

$$H_i(t) = t - W_i(t) \tag{1}$$

Let us define the *(pseudo) inverse* of the idle time, as follows:

Definition 3 *The (pseudo) inverse function $X_i(c)$ of $H_i(t)$ is the smallest time instant such that there are at least c time units when the processor is not running any tasks in $\{\tau_1, \ldots, \tau_i\}$, over every interval of length $X_i(c)$. That is,*

$$X_i(c) = \min_t \{t : H_i(t) \geq c\}$$

We note that $H_i(t)$ is not an invertible function, since there may be several time-instants t for which $H_i(t)$ is constant — that is why we refer to $X_i(c)$ as a pseudo inverse. In the remainder of this paper we will abuse notation somewhat, and use the following notation:

$$X_i(c) = [H_i(t)]^{-1} \tag{2}$$

Based upon this definition of the inverse of the idle time, we obtain the following alternative representation of task response time. (Observe that this relationship holds regardless of whether task deadlines are lesser than, equal to, or greater than periods.)

Lemma 1 *The worst-case response time R_i of task τ_i is given by:*

$$R_i = \max_{k=1,2,\ldots} \{X_{i-1}(k\, C_i) - (k-1)\, T_i\} \tag{3}$$

Proof. $X_{i-1}(k\, C_i)$ is the instant when the first $i-1$ tasks have left $k\, C_i$ units of time available for the lower priority tasks. Hence it is also the finishing time of the kth job of τ_i in the busy period. $(k-1)\, T_i$ is the activation of such a job. The proof hence follows directly as in [23]. $\square$

Notice that if $\sum_{j=1}^{i} U_i > 1$ then the we clearly have $R_i = +\infty$. For this reason in realistic cases we assume $\sum_{j=1}^{i} U_i \leq 1$.

Some further notation: for any function $f(x)$, $f^{\mathsf{ub}}(x)$ denotes an upper bound, and $f^{\mathsf{lb}}(x)$ denote a lower bound on the function $f(x)$, so that we have $f^{\mathsf{lb}}(x) \leq f(x) \leq f^{\mathsf{ub}}(x)$ for all x.

Theorem 1 *For any upper bound $W_i^{\mathsf{ub}}(t)$ on the workload $W_i(t)$, there is a corresponding upper bound R_i^{ub} on the worst-case response time R_i.*

Proof. Since $W_i^{\mathsf{ub}}(t)$ is an upper bound of $W_i(t)$ we have by definition

$$W_i^{\mathsf{ub}}(t) \geq W_i(t)$$

from which it follows the obvious relationship for the idle time

$$H_i^{\mathsf{lb}}(t) = t - W_i^{\mathsf{ub}}(t) \leq t - W_i(t) = H_i(t)$$

which gives us a lower bound of the idle time. From this relationship it follows that for any possible value c we have

$$\{t : H_i^{\mathsf{lb}}(t) \geq c\} \subseteq \{t : H_i(t) \geq c\}$$

Now it is possible to find a relationship between the pseudo-inverse functions. In fact we have

$$X_i^{\mathsf{ub}}(c) = \min_t \{t : H_i^{\mathsf{lb}}(t) \geq c\} \geq \min_y \{t : H_i(t) \geq c\} = X_i(c)$$

from which it follows that

$$R_i^{\mathsf{ub}} = \max_{k=1,2,\ldots} \{X_{i-1}^{\mathsf{ub}}(k\, C_i) - (k-1)\, T_i\} \geq R_i$$

as required. $\square$

3. The workload upper bound

As stated above, it was proved by Liu and Layland [15] that the worst-case workload $W_i(t)$ occurs for the synchronous arrival sequence of jobs — i.e., when all the tasks $\tau_1, \ldots, \tau_i$ are simultaneously activated, and consecutive jobs of τ_i arrive exactly T_i time units apart, for all i. Hence the function $W_i(t)$ may be expressed by the sum of the individual workload of each task τ_j. If we let $w_j(t)$ denote the maximum amount of time that the processor executes task τ_j over the interval $[0, t)$ in this worst-case scenario, we can write:

$$W_i(t) = \sum_{j=1}^{i} w_j(t)$$

This is shown in Figure 1.

Letting $w_j^o(t)$ denote the maximum amount of time that the processor executes task τ_j in any interval of length t, **when task τ_j is the only task in the system**, clearly we have:

$$\forall j \quad \forall t \qquad w_j^o(t) \geq w_j(t)$$

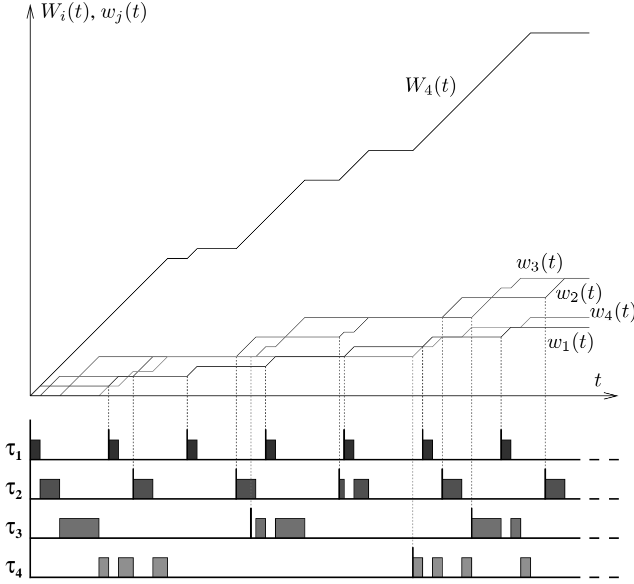


Figure 1. An example of the $W_i(t)$ and $w_j(t)$

since the presence of additional jobs may only delay the execution of τ_j's jobs.

The workload $w_j^o(t)$, which is equal to $\min\left\{t - (T_j - C_j)\left\lfloor\frac{t}{T_j}\right\rfloor, \left\lceil\frac{t}{T_j}\right\rceil C_j\right\}$, can be conveniently upper bounded by the linear function as shown in Figure 2. The equation of the linear bound is $U_j\, t + C_j(1 - U_j)$.

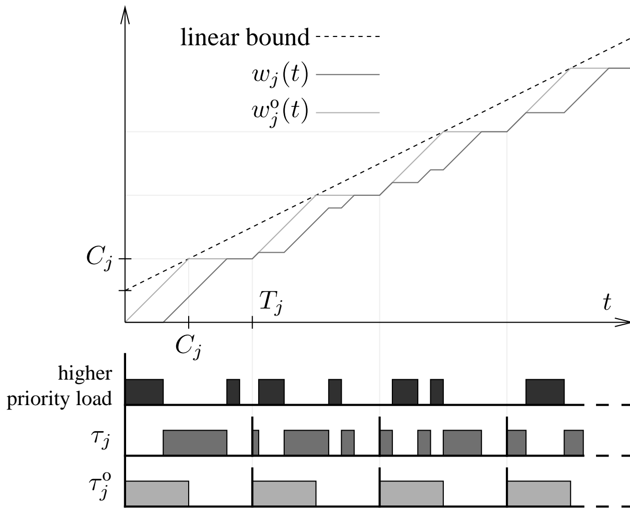


Figure 2. The upper linear bound of $w_j(t)$

Using these relationships found for the workload $w_j(t)$ of each task, if we sum over j from 1 to i we obtain an upper bound on the workload function $W_i(t)$:

$$W_i(t) = \sum_{j=1}^{i} w_j(t) \leq \sum_{j=1}^{i} w_j^o(t) \leq$$

$$\leq \sum_{j=1}^{i} (U_j\, t + C_j\,(1 - U_j)) \quad (4)$$

We have so obtained the upper bound we were looking for. The property of this bound is that we can compute conveniently its inverse function and then apply the Theorem 1 to finally find the bound of the response time.

Theorem 2 *The worst-case response time R_i of task τ_i is bounded from above as follows:*

$$R_i \leq \frac{C_i + \sum_{j=1}^{i-1} C_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j} = R_i^{\mathsf{ub}} \quad (5)$$

Proof. The proof of this theorem is obtained by applying Theorem 1 to the workload bound provided by the Eq. (4). So we have:

$$W_i^{\mathsf{ub}}(t) = \sum_{j=1}^{i} (U_j\, t + C_j(1 - U_j))$$

$$H_i^{\mathsf{lb}}(t) = t\left(1 - \sum_{j=1}^{i} U_j\right) - \sum_{j=1}^{i} (C_j(1 - U_j))$$

Since $H_i^{\mathsf{lb}}(t)$ is invertible, it can be used to compute $X_i^{\mathsf{ub}}(h)$.

$$X_i^{\mathsf{ub}}(h) = \frac{h + \sum_{j=1}^{i} C_j(1 - U_j)}{1 - \sum_{j=1}^{i} U_j}$$

Then the response time is bounded by:

$$\max_{k=1,2,\ldots}\left(\frac{kC_i + \sum_{j=1}^{i-1} C_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j} - (k - 1)T_i\right) \quad (6)$$

We will now prove that the maximum in the Eq. (6) occurs for $k = 1$. Let us consider this function on the real extension $[1, +\infty)$. On this interval we can differentiate with

respect to k. Doing so we get:

$$\frac{\mathrm{d}}{\mathrm{d}k}\left(\frac{kC_i + \sum_{j=1}^{i-1} C_j(1-U_j)}{1-\sum_{j=1}^{i-1} U_j} - (k-1)T_i\right) =$$
$$\frac{C_i}{1-\sum_{j=1}^{i-1} U_j} - T_i =$$
$$T_i\left(\frac{U_i}{1-\sum_{j=1}^{i-1} U_j} - 1\right) =$$
$$T_i\left(\frac{\sum_{j=1}^{i} U_j - 1}{1-\sum_{j=1}^{i-1} U_j}\right)$$

which is always negative (or zero). In fact, if $\sum_{j=1}^{i} U_j > 1$ the response time is known to be arbitrarily long, and so unbounded. Then, since the function is decreasing (or constant), its maximum occurs in the left bound of the interval, which means $k = 1$. Finally, by substituting $k = 1$ in Eq. (6), we get:

$$R_i^{\mathsf{ub}} = \frac{C_i + \sum_{j=1}^{i-1} C_j(1-U_j)}{1-\sum_{j=1}^{i-1} U_j} \tag{7}$$

as required. □

Moreover we can divide by T_i to normalise the bound and we get:

$$r_i^{\mathsf{ub}} = \frac{R_i^{\mathsf{ub}}}{T_i} = \frac{U_i + \sum_{j=1}^{i-1} a_j\, U_j(1-U_j)}{1-\sum_{j=1}^{i-1} U_j} \tag{8}$$

where $a_j = T_j/T_i$.

The time complexity of computing the response time upper bound R_i^{ub} of task τ_i is $O(i)$. Hence the complexity of computing the bound for all the tasks seems to be $O(n^2)$. However, it can be noticed that the computation of R_{i+1}^{ub} can take advantage of the completed computation of R_i^{ub}. In fact the two sums involved in Equation (5) can be simply computed by adding only the values relative to the last index to the sum values of the previous computation. This observation allows us to say that the computation of the response time upper bound of all the tasks in $O(n)$.

There are other techniques to bound the response time. Similarly as suggested by Sjödin and Hansson [21], a different upper bound on the worst-case response times may be obtained from the recurrence used in response-time analysis [10, 2] by replacing the ceiling function $\lceil x \rceil$ with $x + 1$. We have:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$
$$R_i \le C_i + \sum_{j=1}^{i-1} \left(\frac{R_i}{T_j} + 1\right) C_j$$
$$R_i - R_i \sum_{j=1}^{i-1} U_j \le C_i + \sum_{j=1}^{i-1} C_j$$
$$R_i \le \frac{\sum_{j=1}^{i} C_j}{1-\sum_{j=1}^{i-1} U_j}$$

Observe that this is a looser bound than the one we have obtained above, in Theorem 2.

We conclude by reiterating the benefits of using the response time upper bound presented in Theorem 2 above:

- it can be computed in $O(n)$ time;
- it is continuous and differentiable in all the variables;
- the bound holds even for deadlines greater than the period. In this case the exact algorithm for the response time calculation [23] requires to check all the jobs within the busy period;
- the bound has a closed formulation, instead that an iterative definition. Hence it is possible to adopt some feedback on task parameters (C_j or T_j) so that the response time is modified in some desired direction.

3.1. A sufficient schedulability test

In the same way as the exact values of the response times allow to formulate a necessary and sufficient schedulability test, the response time upper bound R_i^{ub} allows to express a sufficient schedulability condition for the fixed priority algorithm. It is then possible to enunciate the following $O(n)$ sufficient schedulability condition for tasks scheduled by fixed priority with arbitrary deadline.

Corollary 1 *A task set $\tau_1, \ldots, \tau_n$ is schedulable by fixed priorities **if**:*

$$\forall i \quad R_i^{\mathsf{ub}} = \frac{C_i + \sum_{j=1}^{i-1} C_j(1-U_j)}{1-\sum_{j=1}^{i-1} U_j} \le D_i \tag{9}$$

Proof. From Theorem 2 it follows that $R_i \le R_i^{\mathsf{ub}}$. From the hypothesis it follows that $R_i^{\mathsf{ub}} \le D_i$. Then it follows that $R_i \le D_i$, which means that all the tasks do not miss their deadlines. □

Corollary 1 provides a very efficient means for testing the feasibility of task sets. This condition can also be restated as a utilisation upper bound, and compared with many existing schedulability tests [15, 12, 11, 6]. Since some of these results are achieved assuming deadlines equal to periods, we also provide the following corollary in this hypothesis although this restriction doesn't apply to our response time upper bound.

Corollary 2 *A task set $\tau_1, \ldots, \tau_n$, with deadlines equal to periods ($D_i = T_i$) is schedulable by fixed priorities **if**:*

$$\forall i \quad \sum_{j=1}^{i} U_j \leq 1 - \sum_{j=1}^{i-1} a_j\, U_j(1 - U_j) \tag{10}$$

where $a_j = T_j/T_i$.

Proof. From Equation (9) it follows that the task τ_i is schedulable if

$$\frac{U_i + \sum_{j=1}^{i-1} a_j\, U_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j} \leq 1$$

where $a_j = \frac{T_j}{T_i}$. Also notice that if tasks are scheduled by RM then $a_j \leq 1$ always. From the last equation we have

$$U_i + \sum_{j=1}^{i-1} a_j\, U_j(1 - U_j) \leq 1 - \sum_{j=1}^{i-1} U_j$$

$$\sum_{j=1}^{i} U_j \leq 1 - \sum_{j=1}^{i-1} a_j\, U_j(1 - U_j)$$

which proves the corollary, when ensured for all tasks. □

It is quite interesting to observe that when the periods are quite large compared to the preceding one — meaning that $a_j \to 0$ — then the test is very effective. On the other hand, when all the periods are similar each other then the right hand side of Eq. (10) may also become negative, making the condition impossible. This intuition will be confirmed in the next section dedicated to the experiments.

4. Experiments

The major benefits of the response time upper bound that we have computed in Section 3 above lie in **(i)** the *time complexity* which, at $O(n)$ where n denotes the number of tasks, is linear in the representation of the task system; and **(ii)** the fact that the upper bound is *continuous* with respect to the task system parameters (and hence more useful in interactive system design). It is however, also important to evaluate the quality of the bound. Clearly, the tightness of the approximation depends upon the task set parameters. In order to estimate the distance between the exact value of the response time and of our derived upper bound (thereby determining the "goodness" of our upper bound), we performed a series of experiments that explored the impact of the different task characteristics.

4.1. Effect of task periods

In the first set experiments we evaluate the impact of task periods on the response time upper bound. For this purpose, we use a system comprised of only 2 tasks. The period of the higher-priority task is set $T_1 = 1$, whereas the period T_2 of the low priority task is calculated so that the ratio T_1/T_2 ranges in the interval $[0, 1]$. The task computation times C_1 and C_2 are chosen such that:

- the relative utilisations of the two tasks does not change in the experiments. This is achieved by setting $U_1/U_2 = 0.25$ always;
- the total utilisation $U = U_1 + U_2$ is equal to one of the four values $\{0.2, 0.4, 0.6, 0.8\}$ (we run four classes of experiments, one for each value).

We leave the values of D_1 and D_2 unspecified, since these parameters have no effect on either the exact response time, or our computed upper bound, under FP scheduling.

For each simulation, we computed the exact response time R_2 and our upper bound R_2^{ub} for the task τ_2. Notice that both the tasks will have response times smaller than or equal to their respective periods since the Liu and Layland utilisation bound for two tasks is $2(\sqrt{2} - 1) \approx 0.828$, which is greater than all the total utilisations assumed in this experiment. Hence the maximum response time occurs in the first job of τ_2. Both the response time and the upper bound are normalised with respect to the period T_2, so that the comparison between different values of the period T_2 is easier. The results are shown in Figure 3. Black lines are the normalised R_2^{ub} values, gray plots are the exact response times.

It may be noticed that the approximation is very good when $T_2 \gg T_1$ (i.e. when the ratio T_1/T_2 is close to zero). In fact, in this condition the workload estimate, upon which the response time bound is built, becomes very tight. The discontinuities in the response times occur when an additional job of τ_1 interferes with the response time of τ_2. Finally, it may be noticed that the approximation degrades as the total utilisation increases. This can be explained by reiterating that the upper estimate of the workload is tight for low utilisations, as can be observed from Figure 2.

Given this last observation, it becomes quite interesting to test the case when $U = 1$. In this condition of heavy load, the task system utilisation is no longer $\leq$ the Liu and

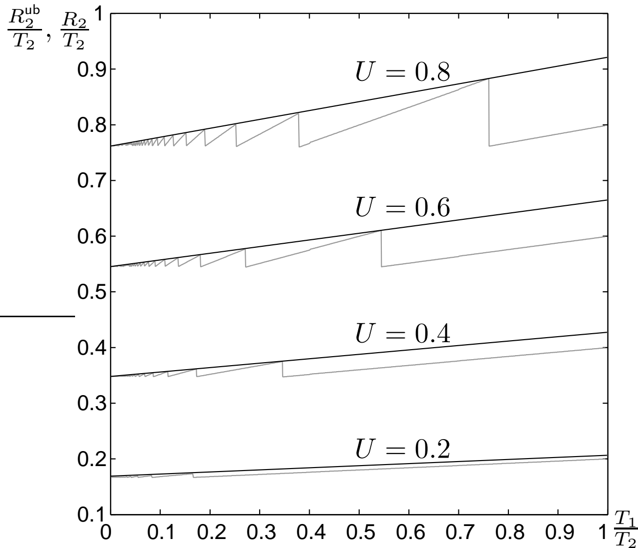


Figure 3. Effect of task periods

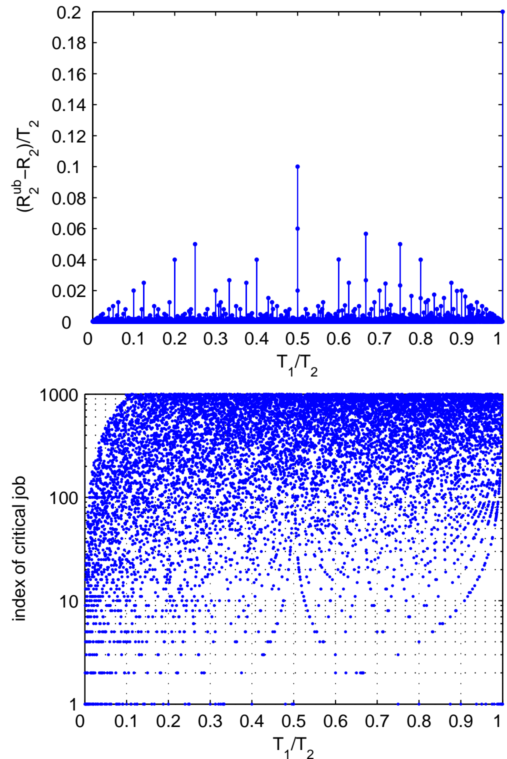


Figure 4. Response time bound, when $U = 1$

Layland utilisation bound, and hence it is not guaranteed that both tasks' response times will be $\leq$ their respective period parameters. Furthermore, the response time R_2 does not necessarily occur at the first job, and hence all the jobs within the first busy period must be checked. (In fact, under the condition $U = 1$ the processor is always busy and the busy period never ends, but the response time can still be computed by checking all the jobs up to hyperperiod — the least common multiple of all the periods.) A second — more serious — problem is related to the nature of the experiment: since we are running simulations as the period T_2 varies from T_1 to infinity, the hyperperiod can be extremely large! (Indeed, the hyperperiod does not even exist if T_1/T_2 is irrational, although this phenomenon is not encountered with machine representable numbers.) Hence, in our simulation setting the computation of the response time is stopped after 1000 jobs of τ_2. In the top part of Figure 4 we report the difference $R_2^{\text{ub}} - R_2$ normalised with respect to T_2 as usual. The result is quite surprising.

From the figure we see that the upper bound is a very tight approximation of the exact response time, unless *some harmonic relationship exists* between T_1 and T_2. Moreover, the stronger the harmonicity the greater the difference between the bound and the exact value (for example when $\frac{T_1}{T_2} \in \{1, \frac{1}{2}, \frac{1}{4}, \frac{2}{3}\}$.) When the periods are poorly harmonic the upper bound is extremely tight.

In these experiments we observed that in poorly harmonic periods, the response time routine needs to be conducted much further than in more harmonic conditions. The bottom part of Figure 4 reports, on a log scale, the index of the job of τ_2 that experiences the maximum response time (the *critical job*). When the periods are in some harmonic relationship the critical job occurs relatively early. However, when the harmonic relationship is poor we often stop our computation because of our job limit at 1000 jobs.

This observation motivated the third and last set of experiments exploring the influence of periods. We want to evaluate what the critical job is, when the periods are poorly harmonic. For this purpose, we set $\frac{T_2}{T_1} = \sqrt{2}$ so that the notion of hyperperiod doesn't exist (clearly on machine representable numbers, T_1 and T_2 are still rational.) We set the ratio $\frac{U_1}{U_2} = 0.25$ (meaning that the τ_1 has a significantly lower load than τ_2, although this setting did not seem to significantly affect the simulation results). The experiments are carried out varying the total utilisation in the proximity of $U = 1$. Again, we stopped the computation of response time after 10000 jobs. Figure 5 reports the index of the critical job in log scale.

It may be noticed clearly that as the total utilisation approaches 1 the index of the critical job progressively increases, until the computation is artificially interrupted at job 10000. Actually when the utilisation is exactly 1, we believe that *there always exists some future job with longer response time*. Observing this phenomenon has lead us to formulate the following conjecture.

Conjecture 1 *When $U = 1$ and the ratio $\frac{T_2}{T_1}$ is irrational then the index of the critical job is unbounded. Moreover*

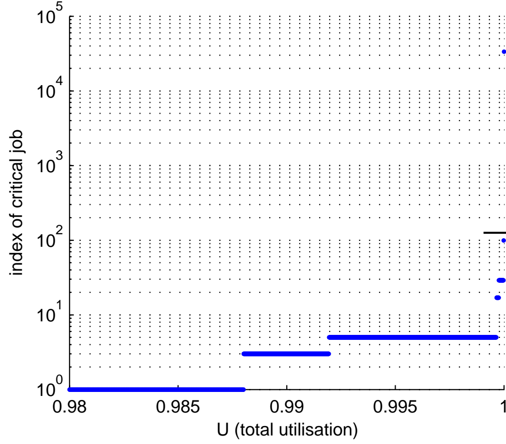


Figure 5. The index of the critical job.

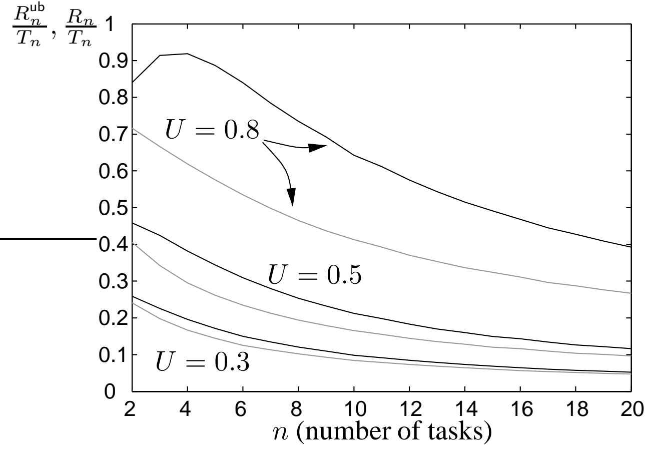


Figure 6. Response bound and tasks number

we have

$$\limsup_{k} R_{2,k} = R_2^{\mathsf{ub}} \tag{11}$$

where $R_{2,k}$ *denotes the response time of the* k^{th} *job of task* τ_2.

4.2. Effects of the number of tasks

In this set of experiments we focus on the influence of the number of tasks both on the actual response time and on the upper bound derived by us in Section 3. The number of tasks ranges from 2 to 20. The experiment is run under three different total load condition represented by $U = 0.3$ (light load), $U = 0.5$ (average load) and $U = 0.8$ (heavy load). The total load is uniformly distributed among the single tasks using the simulation routine suggested by Bini and Buttazzo [5]. Notice that as the number of tasks increases all the individual utilisations U_i tend to decrease because the total utilisation U is kept constant. The period T_1 of τ_1 is set equal to one, and the remaining periods are randomly selected such that T_{i+1}/T_i is uniformly distributed in $[1, 3]$. For each pair (number of tasks n, total utilisation U) we ran 10000 simulations and computed the normalised response time R_n/T_n — drawn in gray — and the normalised upper bound R_n^{ub}/T_n — in black. Figure 6 reports the average value of all the simulations. The figure shows three pairs of plots, relative to the three different values of utilisation simulated.

It may seem quite unexpected that the response times does not increase with the number of tasks. However, we must remember that we are plotting values normalised with the period T_n. To confirm the validity of the experiments we can compute the limit of the normalised response time, reported in Eq. (8), as n grows to infinity. In order to compute the limit we assume that all the tasks utilisations are the same (i.e., each is equal to U/n) and all the period ratios a_j are equal to a common value a. The asymptotic value of the response time then is

$$\begin{aligned}\lim_{n\to\infty} r_n^{\mathsf{ub}} &= \frac{\frac{U}{n} + (n-1)a\frac{U}{n}(1-\frac{U}{n})}{1-(n-1)\frac{U}{n}} \\ &= \frac{U + (n-1)a\,U(1-\frac{U}{n})}{n-(n-1)U} \\ &= \frac{U + (n-1)a\,U}{n-(n-1)U} \\ &= \frac{a\,U}{1-U}\end{aligned}$$

which is constant.

4.3. The sufficient test

In the final experiments we evaluated the number of tasks sets accepted by the sufficient test stated in Corollary 2. This test is compared with other simple sufficient tests: the Hyperbolic Bound [6] and the utilisation RBound [11]. We remind that the complexity of the test presented here and the Hyperbolic Bound in $O(n)$, whereas the complexity of the utilisation RBound is $O(n \log n)$, where n denotes the number of tasks.

First we investigated the effect of the period on the quality of the sufficient tests. We arbitrarily set the number of tasks equal to 5 and the total utilisation $U = 0.8$ so that the random task sets are not trivially schedulable. The periods are randomly extracted as follows: **(i)** T_1 is set equal to 1 and **(ii)** the other periods T_i are uniformly extracted in the interval $[T_{i-1}, r\,T_{i-1}]$. The parameter r, denoted by *period dispersion* in Figure 7, measures how close each other are the periods. For example if $r = 1$ then all the periods are the same, if r is large then the next random period tends to be large compared with the previous one. The experiments

are conducted for r varying from 1 to 7, and for each setting we extracted 5000 task set. The quality of the tests is measured by the *acceptance ratio*, which is the percentage of schedulable task sets accepted by each of the three sets [5]. The results are shown in Figure 7.

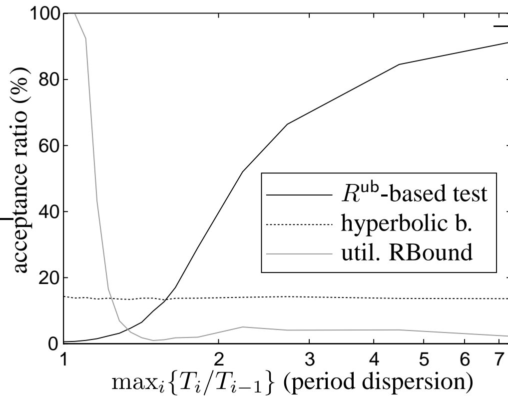


Figure 7. Acceptance ratio and periods

First, the figure confirms that the Hyperbolic Bound is not affected at all by the variation of the periods. In fact, this test is performed only on task utilisations which are left unchanged. Then we observe that when the periods are close each other (period dispersion close to 1) the RBound dominates, whereas for large periods the test based on the response time bound performs better than the others. The possible explanation is that the RBound is built starting from the Liu and Layland [15] worst-case periods which are all very close each other.

Finally, we evaluated the acceptance ratio as the number of tasks varies from 2 to 20. The total utilisation is equal to 0.75 so that a considerable number of task sets are schedulable also when the number of tasks is maximum. The period dispersion r, as defined previously, is set equal to 1.4 so that we work in an area where all the three tests seem comparable from Figure 7. The acceptance ratio is reported in Figure 8.

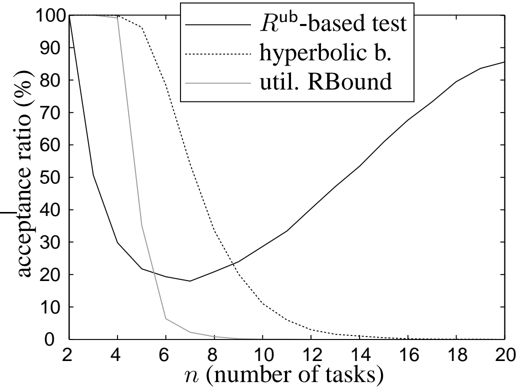


Figure 8. Acceptance ratio and tasks number

In this case the Hyperbolic Bound is always superior to the RBound, although this may happen because the period dispersion r is chosen too high. Anyhow, the most interesting aspect is that when the number of tasks grows beyond 8, the quality of the R^{ub}-based test starts increasing. This phenomenon is justified by observing that as the number of tasks grows, all the individual utilisations become smaller and smaller. Under this condition, as discussed previously, the workload upper bound — and the test based on it — is very tight.

5. Conclusions

Response time analysis (RTA) is an important approach to feasibility analysis of real-time systems that are scheduled using fixed-priority (FP) scheduling algorithms. Two drawbacks of RTA are: **(i)** computing response times takes time pseudo-polynomial in the representation of the task system; and **(ii)** response times are not in general continuous in task system parameters.

In this paper, we have derived an upper bound on the response times in sporadic task systems scheduled using FP algorithms. Our upper bound can be computed in polynomial time, and has the added benefit of being continuous and differentiable in the task system parameters. We have designed and conducted a series of simulation experiments to evaluate the goodness of our approach. These simulations have had the added benefit of giving rise to an interesting theoretical conjecture concerning response times for systems in which all parameters need not be rational numbers.

References

[1] Karsten Albers and Frank Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the* 16th *Euromicro Conference on Real-Time Systems*, pages 187–195, Catania, Italy, June 2004.

[2] Neil C. Audsley, Alan Burns, Mike Richardson, Ken W. Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[3] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the* 11th *IEEE Real-Time Systems Symposium*, pages 182–190, Lake Buena Vista (FL), U.S.A., December 1990.

[4] Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, November 2004.

[5] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1–2):129–154, May 2005.

[6] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[7] Reinder J. Bril, Wim F. J. Verhaegh, and Evert-Jan D. Pol. Initial values for on-line response time calculations. In *Proceedings of the* 15th *Euromicro Conference on Real-Time Systems*, pages 13–22, Porto, Portugal, July 2003.

[8] Anton Cervin and Johan Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 1(2):209–224, 2005.

[9] Nathan Fisher and Sanjoy Baruah. A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems. In *Proceedings of the* 17th *Euromicro Conference on Real-Time Systems*, pages 117–126, Palma de Mallorca, Spain, July 2005.

[10] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

[11] Sylvain Lauzac, Rami Melhem, and Daniel Mossé. An improved rate-monotonic admission control and its applcations. *IEEE Transactions on Computers*, 52(3):337–350, March 2003.

[12] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadline. In *Proceedings of the* 11th *IEEE Real-Time Systems Symposium*, pages 201–209, Lake Buena Vista (FL), U.S.A., December 1990.

[13] John P. Lehoczky, Lui Sha, and Ye Ding. The rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the* 10th *IEEE Real-Time Systems Symposium*, pages 166–171, Santa Monica (CA), U.S.A., December 1989.

[14] Joseph Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[15] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[16] Wan-Chen Lu, Jen-Wei Hsieh, and Wei-Kuan Shih. A precise schedulability test algorithm for scheduling periodic tasks in real-time systems. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1451–1455, Dijon, France, April 2006.

[17] Yoshifumi Manabe and Shigemi Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2):171–181, March 1998.

[18] Aloysius Ka-Lau Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Boston (MA), U.S.A., May 1983.

[19] José Carlos Palencia and Michael González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the* 19th *IEEE Real-Time Systems Symposium*, pages 26–37, Madrid, Spain, December 1998.

[20] Pascal Richard and Joël Goossens. Approximating response times of static-priority tasks with release jitters. In 18th *Euromicro Conference on Real-Time Systems, Work-in-Progress*, Dresden, Germany, July 2006.

[21] Mikael Sjödin and Hans Hansson. Improved response-time analysis calculations. In *Proceedings of the* 19th *IEEE Real-Time Systems Symposium*, pages 399–408, Madrid, Spain, December 1998.

[22] Ken Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50:117–134, April 1994.

[23] Ken W. Tindell, Alan Burns, and Andy Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–152, March 1994.

Approximate Feasibility Analysis and Response-Time Bounds of Static-Priority Tasks with Release Jitters

Pascal Richard
LISI/ENSMA
University of Poitiers (France)
pascal.richard@univ-poitiers.fr

Joël Goossens
Computer Science Department
Université Libre de Bruxelles
joel.goossens@ulb.ac.be

Nathan Fisher
Department of Computer Science
University of North Carolina, Chapel Hill
fishern@cs.unc.edu

Abstract

We consider static-priority tasks with constrained-deadlines that are subjected to release jitter. We define an approximate worst-case response-time analysis *and propose a polynomial-time algorithm. For that purpose, we extend the Fully Polynomial-Time Approximation Scheme (*FPTAS*) presented in [6] to take into account release jitter constraints; this feasibility test is then used to define a polynomial time algorithm that approximate worst-case response times of tasks. Nevertheless, the approximate worst-case response time values have not been proved to have any bounded error in comparison with worst-case response times computed by an exact algorithm (with pseudo-polynomial time complexity).*

1 Introduction

Guaranteeing that tasks will always meet their deadlines is a major issue in the design of hard-real time systems. We consider the problem of ensuring that periodic tasks scheduled by a preemptive static-priority scheduler upon a uniprocessor platform meet all deadlines. Every execution of a given task is called a job. We consider tasks that have constrained-deadlines (i.e., deadlines are less than or equal to task periods) and are subjected to release jitter. A *release jitter* models an interval of time in which a task waits the next tick of the RTOS in order to start or is pending due to input communications.

Tasks are scheduled at run-time using a static-priority scheduling policy. Every task has a static priority and at any time the executed job has the highest priority among tasks awaiting execution. The feasibility problem consists of proving that tasks will always meet their deadlines at run-time. For the considered real-time systems, the feasibility problem is not known to be NP-hard, but only pseudo-polynomial time tests are known. However, pseudo-polynomial time complexity is too computationally expensive for performing on-line task admission or for analysing large distributed systems using classical methods such as the holistic analysis [19].

For a static-priority system, a task set is *feasible* on a given processing platform, if every task will always meet all deadlines when scheduled according to its given static-priority on the given platform. A feasibility test is an algorithm used to check if a task set is feasible or not. One can distinguish several approaches to designing a feasibility test for real-time task sets: *(i)* an exact feasibility test, *(ii)* a sufficient feasibility test (also known as pessimistic feasibility test) and *(iii)* an approximate feasibility test. We briefly describe their main characteristics.

An *exact feasibility test* can always correctly categorize task sets as either *feasible* or *infeasible* upon a specific hardware platform [10, 13, 15]. An exact test will label a periodic task set as "infeasible" if and only if the task set will miss a deadline at run-time. Neither a polynomial-time test nor NP-hardness result are known for static-priority tasks having constrained-deadlines.

A *sufficient feasibility test* always leads to an exact positive decision: if the test concludes that a task set is feasible then no deadline will be missed at run-time. But, when it concludes that a task is infeasible, then it may be a rather pessimistic decision (i.e., tasks may meet their deadlines at run-time). Sufficient feasibility tests have a lower computation complexity than corresponding exact feasibility tests. Numerous sufficient feasibility tests are known in the literature (e.g. [16, 11, 3, 9, 1, 4]).

An *approximate feasibility test* is based on the approximability theory of NP-hard optimization problems [7]. It reduces the gap between the two previous approaches to control the "unused processor capacity" for tests based on the processor-demand analysis. It runs in *polynomial*-time according to an accuracy parameter ϵ. An approximate feasibility test allows to conclude that a task set is [6, 5]:

- feasible (upon a unit-speed processor).

- infeasible upon a $(1-\epsilon)$-speed processor. That is, "we must effectively ignore ϵ of the processor capacity for the test to become exact" [6]. So, the pessimism introduced by the feasibility test is kept bounded by a constant multiplicative factor.

In [17], some numerical experiments are presented that show the practical interest of several approximate feasibility analysis in comparison with exact feasibility tests.

Most of feasibility tests produce a boolean decision: feasibility or infeasibility. However, an important qualitative measure for a task is its *worst-case response time* (i.e., the maximum size interval of time between a release of a task and its completion). Response-Time Analysis is often used to quantify the maximum earliness or tardiness of tasks and to bound release jitter of dependent tasks or messages in a distributed system. For synchronous static-priority systems, worst-case response times of tasks can be computed in *pseudo*-polynomial time.

This research. As far as we know, no approximation algorithm is known for approximating worst-case response times of tasks with a constant performance guarantee (i.e., upper bounds of worst-case response times). The aim of this paper is to introduce such an analysis and to try to show its relationship with approximate feasibility analysis. We present an FPTAS for analysing the feasibility of static priority tasks with release jitter constraints. We then show feasibility tests can be used to define upper bounds of worst-case response times based on a polynomial time algorithm. Lastly, we show that there exists some task systems such that ratio between the exact worst-case response time and the approximate worst-case response time is not bounded.

Organization. The remainder of this paper is organized as follows. We first define a preliminary result for computing worst-case response times while performing a processor demand analysis (e.g., [13]), then we extend the FPTAS presented in [6] with release jitter constraints. These results are then combined to define for computing approximate worst-case response times. Nevertheless, we show via a counter-example that the computed approximate worst-case response times values are not guaranteed to be close to actual worst-case response times (i.e., with a bounded error).

2 Task Model and Exact Analysis

2.1 Task Model

In this paper, we assume that all tasks share a processor upon which all jobs must execute. Every job can be preempted and resumed later at no cost or penalty. Without loss of generality, we also assume the the rate of the processor is exactly one, since if it is not the case all processing requirements can be normalized to the processor speed.

A task τ_i, $1 \leq i \leq n$, is defined by a worst-case execution requirement C_i, a relative deadline D_i and a period T_i between two successive releases. Every task occurrence is called a job. We assume that deadlines are constrained: $D_i \leq T_i$. Such an assumption is realistic in many real-world applications and also leads to simpler algorithms for checking feasibility of task sets [12]. Moreover, we define the utilization factor of the periodic tasks as follows: $U \stackrel{\text{def}}{=} \sum_{i=1}^{n} C_i/T_i$. We consider a discrete scheduling model and thus we assume that all parameters are integers.

In order to model delay due to input data communications of tasks, we also consider that jobs are subjected to release jitter. A release jitter J_i of a task τ_i is a interval of time after the release of a job in which it waits before starting its execution. In the following, we assume that $0 \leq J_i \leq D_i$ (otherwise the system is obviously not schedulable). Release jitter constraints model delays introduced by the RTOS in presence of system ticks or input communications. For this latter case, dependencies among distributed tasks are modeled using the parameters J_i, $1 \leq i \leq n$. Using such a model, a distributed system can be analysed processor by processor, separately using for instance an holistic based schedulability analysis [19].

For a given processor, we assume that all tasks are independent and synchronously released. All tasks have static priorities that are set before starting the application and are never changed at run-time. At any time, the highest priority task is selected for execution among ready tasks. Without loss of generality, we assume next that tasks are indexed according to priorities: τ_1 is the highest priority task and τ_n is the lowest priority one.

2.2 Known Results

2.2.1 Request-Bound and Workload Functions

In presence of release jitter constraints, the request-bound function of a task τ_i at time t (denoted $\text{RBF}(\tau_i, t)$) and the cumulative processor demand (denoted $W_i(t)$) of tasks at time t of tasks having priorities greater than or equal to i are respectively (see [19] for details):

$$\text{RBF}(\tau_i, t) \stackrel{\text{def}}{=} \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i \tag{1}$$

$$W_i(t) \stackrel{\text{def}}{=} C_i + \sum_{j=1}^{i-1} \text{RBF}(\tau_j, t) \tag{2}$$

Informally, the request-bound function for a task τ_i and positive t is the maximum execution requirement of jobs of τ_i released in any continuous interval of length t.

Using these functions, two distinct (but linked) exact feasibility tests can be defined. We restate both results that will be reused in the remainder.

2.2.2 Time-Demand Analysis

The time-demand approach checks that the processor capacity is always less than or equal to the processor capacity required by task executions. [13] presents a processor-

demand schedulability test for constrained-deadline systems (but the test was extended for arbitrary deadline systems in [12]). It can be also easily extended to tasks subjected to release jitter as stated in the following result (a proof can be found in [8]):

Theorem 1 *[13, 15] A static-priority system with release jitter contraints is feasible iff* $\max_{i=1\ldots n}\left\{\min_{t\in S_i}\frac{W_i(t)}{t}\right\} \leq 1$*, where* S_i *is the set of scheduling points defined as follows:* $S_i \stackrel{\text{def}}{=} \{aT_j - J_j \mid j = 1\ldots i, a = 1\ldots \left\lfloor\frac{D_i-J_i+J_j}{T_j}\right\rfloor\} \cup\{D_i - J_i\}$.

Note that schedulability points correspond to a set of time instants in the schedule where a task can start its execution, after the delay introduced by its release jitter. From a computational complexity point of view, for any integer k, there is a task system with two tasks such that the time complexity of the time-demand analysis is at least $O(k)$ (Lemma 1, [15]).

2.2.3 Response-Time Analysis

An alternative approach for checking the feasibility of a static-priority task set is to compute the worst-case response time R_i. The worst-case response time of τ_i is formally defined as:

Definition 1 *The worst-case response time of a task* τ_i *subjected to a release jitter is:* $R_i \stackrel{\text{def}}{=} (\min\{t > 0 \mid W_i(t) = t\}) + J_i$.

Note that for infeasible tasks R_i does not necessarily correspond to the worst case response time, but instead only corresponds to the worst-case response time of the first job of τ_i.

Exact algorithms for calculating the worst-case response time of periodic tasks are known (e.g., see [10] for a recursive definition of the following method). Using successive approximations starting from a lower bound of R_i, we can compute the smallest fixed point of $W_i(t) = t$ with the following sequence. By Definition 1, this smallest fixed point is the worst-case response time for feasible task τ_i.

$$W_i^{(0)} = \sum_{j=1}^{i} C_j$$

$$W_i^{(k+1)} = C_i + \sum_{j=1}^{i-1} \text{RBF}(\tau_j, W_i^{(k)})$$

The recursion terminates (assuming that $U \leq 1$) for the smallest integer k such that: $W_i^{(k+1)} = W_i^{(k)}$ (i.e., the smallest fixed point of the equation $W_i(t) = t$ has been reached).

The processor-demand analysis and the response-time analysis are both based on the cumulative request-bound function (i.e., Equation 2).

Nevertheless, to the best of our knowledge, no direct link is known between these methods for validating static-priority task sets. In this section, we propose combining the aforementioned analysis techniques in an algorithm that calculates the response time of a periodic task in the presence of release jitter constraints. The initial value (e.g., $W_i^{(0)}$) plays an important role to limit the number of required iterations to reach the smallest fixed point of equation $W_i(t) = t$. Different approaches have been proposed in [18, 2] and are quite useful in practice to reduce computation time. Nevertheless, such improvements are not necessary to present our results and for that reason are not developed in the remainder.

As in the processor-demand approach, the worst-case response-time computation can be done in pseudo-polynomial time. Furthermore, for any integer k, there is a task system with two tasks such that the time complexity of the response-time analysis is at least $O(k)$ (Lemma 2, [15]).

2.3 A Preliminary Result

We show that worst-case response times of tasks can be computed using a Time-Demand Analysis (i.e., using Theorem 1), for every feasible task set. For a feasible task τ_i, it is sufficient to check the following testing set [13]:

$$S_i \stackrel{\text{def}}{=} \{aT_j - J_j \mid j = 1\ldots i, a = 1\ldots \left\lfloor\frac{D_i-J_i+J_j}{T_j}\right\rfloor\} \cup\{D_i - J_i\}$$

We first define the critical scheduling point that facilitates the computation of the worst-case response time of τ_i (under the assumption that the task τ_i will meet its deadline at execution time).

Definition 2 *The critical scheduling point for a feasible task* τ_i *is:* $t^* \stackrel{\text{def}}{=} \min\{t \in S_i \mid W_i(t) \leq t\}$.

We now prove if t^* exists, then $W_i(t^*) + J_i$ defines the worst-case response time of τ_i.

Theorem 2 *The worst-case response time of a feasible task* τ_i *is exactly* $R_i = W_i(t^*) + J_i$.

Proof:

Since task τ_i is feasible then we verify that $W_i(t^*) \leq t^*$. Let $S_i = \{t_{i1}, t_{i2}, \ldots, t_{i\ell}\}$ with $t_{i1} < t_{i2} < \cdots < t_i^* < \cdots < t_{i\ell} = D_i - J_i$. By Definition 2, there exists $t^* = t_{ij}$, where $1 \leq j \leq \ell$, is the first scheduling point verifying $W_i(t^*) \leq t^*$: $W_i(t) > t$ for all $t \in \{t_{i1}, \ldots, t_{ij-1}\}$ and $W_i(t_{ij}) \leq t_{ij}$.

Since $W_i(t)$ is non-decreasing between subsequent scheduling points $\{t_{ia}, t_{ia+1}\}$, $1 \leq a \leq \ell - 1$, then there exists a time $t \in (t_{ij-1}, t_{ij}]$ such that $W_i(t) = t$. Since scheduling points in S_i corresponds to task releases, then no new task is released between t and t^* and as a consequence we have $W_i(t) = W_i(t^*)$. The worst-case response time of τ_i is then defined as $W_i(t^*) + J_i$. ■

Tasks	C_i	D_i	T_i	J_i
τ_1	1	3	3	2
τ_2	2	5	5	1
τ_3	1	12	12	2

Table 1. Static-priority task set with release jitter constraints

$t \in S_i$	1	4	7	9
$W_1(t)/t$	1			
$W_2(t)/t$	3	1		
$W_3(t)/t$	4	1.25	1.14	1

Table 2. Exact Time-Demand Analysis

Thus, for all feasible tasks, we can compute their worst-case response times. But, t^* is not defined for an infeasible task τ_i , thus there is no scheduling point $t \in S_i$ such that $W_i(t) \leq t$. For this latter case, the presented method cannot be used to compute a worst-case response time (i.e., some scheduling points after the deadline must be considered).

Since the size of S_i depends on $\sum_{j=1}^{i-1} \lfloor \frac{D_i - J_i + J_j}{T_j} \rfloor$, then the algorithm runs in *pseudo*-polynomial time. Note that computing the smallest fixed-point $W_i(t) = t$ using successive approximation is also performed in pseudo-polynomial time.

Let us take an example, consider the task set presented in Table 1. The utilization factor is $U = 0.81$. The computations associated with the exact tests are given in Table 2. Figure 1 presents $W_3(t)$ and the processor capacity (i.e., $f(t) = t$). Notice that for every task $\tau_i, 1 \leq i \leq n$ the first value such that $W_i(t)/t \leq 1$ leads to its exact worst-case response time:

- for τ_1, $R_1 = W_1(1) + J_1 = 1 + 2 = 3$,
- for τ_2, $R_2 = W_2(4) + J_2 = 4 + 1 = 5$,
- for τ_3, $R_3 = W_3(9) + J_3 = 9 + 2 = 11$.

3 A FPTAS for Feasibility Analysis of a Task

3.1 Approximating the Request-Bound Function

For synchronous task systems without release jitter, the worst-case activation scenario for the tasks occurs when they are simultaneously released [14]. When tasks are subjected to release jitter, then the worst-case processor workload occurs when all higher-priority tasks are simultaneously available after J_i units of time (e.g., when their input data are available). Notice that deadline failures of τ_i (if any) occur necessarily in an interval of time where only tasks with a priority higher or equal to i are running. Such an interval of time is defined as a level-i busy period [13]. When analysing a task τ_i, if we assume that the analysed processor busy period starts at time 0, then the worst-case workload in that busy period is defined by the release of task τ_j at time $-J_j$, $j \leq i$. According to such a scenario, the total execution time requested at time t by a task τ_i is defined by [19]: $\text{RBF}(\tau_i, t) \stackrel{\text{def}}{=} \left\lceil \frac{t+J_i}{T_i} \right\rceil C_i$.

The RBF function is a discontinuous function with a "step" of height C_i every T_i units of time. In order to approximate the request bound function according to an error bound ϵ (accuracy parameter, $0 < \epsilon < 1$), we use the same principle as in [6, 5]: we consider the first $(k-1)$ steps of $\text{RBF}(\tau_i, t)$, where k is defined as $k = \lceil 1/\epsilon \rceil - 1$ and a linear approximation, thereafter. The approximate request bound function is defined as follow:

$$\delta(\tau_i, t) = \begin{cases} \text{RBF}(\tau_i, t) & \text{for } t \leq (k-1)T_i - J_i, \\ C_i + (t + J_i)\frac{C_i}{T_i} & \text{otherwise.} \end{cases} \tag{3}$$

Notice that up to $(k-1)T_i - J_i$ the approximate request-bound function is equivalent to the exact request-bound function of τ_i, and after that it is approximated by a linear function with a slope equal to the utilization factor of τ_i. The next subsection describes how we use the approximation to the request-bound function to obtain an approximation scheme for feasibility analysis of static-priority tasks subjected to release jitter constraints.

3.2 Approximation Scheme

[19] shows that a static-priority task system with release jitter constraints is feasible, iff, worst-case response times of tasks are not greater than their relative deadlines. This problem is known as the *release jitter problem*. An alternative way is to define a time-demand approach for solving the release jitter problem using the principles of the well-known exact feasibility test presented for the rate monotonic scheduling algorithm in [13].

As presented in Theorem 1, the cumulative request bound function at time t is defined by: $W_i(t) \stackrel{\text{def}}{=} C_i + \sum_{j=1}^{i-1} \text{RBF}(\tau_j, t)$. A task τ_i is feasible (with a constrained relative deadline) iff, there exists a time t, $0 \leq t \leq D_i - J_i$, such that $W_i(t) \leq t$. Since request bound functions are step functions, then $W_i(t)$ is also a step function that increases for every scheduling point in the following set $S_i = \{t = bT_a - J_a; a = 1 \ldots i, b = 1 \ldots \left\lfloor \frac{D_i - J_i + J_a}{T_a} \right\rfloor\} \cup \{D_i - J_i\}$. The feasibility test can then be formulated as follows: if there exists a scheduling point $t \in S_i$, such that $W_i(t)/t \leq 1$ then the task is feasible.

To define an approximate feasibility test, we define an approximate cumulative request bound function as:

$$\widehat{W_i}(t) \stackrel{\text{def}}{=} C_i + \sum_{j=1}^{i-1} \delta(\tau_j, t)$$

According to the error bound ϵ leading to $k = \lceil 1/\epsilon \rceil - 1$, we define the following testing set $\widehat{S_i} \subseteq S_i$:

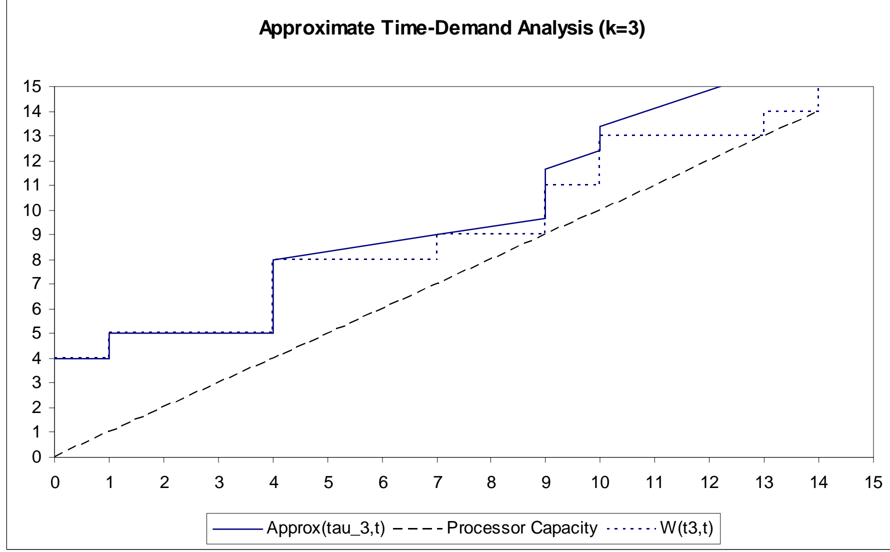


Figure 1. Exact and approximate cumulative request bound functions $W_3(t)$ and $\widehat{W_3}(t)$ with $\epsilon = 0.3$ leading to $k = 3$. Steps occurs at time $aT_i - J_i$ where $0 < a \leq k - 1$ and $0 \leq i \leq n$ before starting linear approximations. The approximate test concludes that τ_3 is not feasible upon a $(1-\epsilon)$-speed processor.

$$\widehat{S_i} \stackrel{\text{def}}{=} \{t = bT_a - J_a; a = 1 \dots i-1, b = 1 \dots k-1\} \cup \{D_i - J_i\}$$

We consider the task set presented in Table 1, the cumulative request bound function $\widehat{W_3}(t)$ is presented in Figure 1 using $\epsilon = 0.3$. This means exactly three steps will be considered for every task (i.e., $k = 3$) before approximating the request bound function using a linear function. We indicate without providing computation details that worst-case response times of τ_1 and τ_2 can be exactly computed since they are achieved before approximating request bound functions. But as shown in Figure 1, the approximate feasibility test concludes that τ_3 is not feasible because $\widehat{W_3}(t) > t$ for all scheduling points (i.e., for all $t \in \widehat{S_3}$).

This is a FPTAS since the algorithm is polynomial according to the input size and the input parameter $1/\epsilon$. We now prove the correctness of this approximate feasibility test.

3.3 Correctness of Approximation

The key point to ensure the correctness is: $\delta(\tau_i, t)/\text{RBF}(\tau_i, t) \leq (1 + \epsilon)$. This result will then be used to prove that if a task set is stated infeasible by the FPTAS, then it is infeasible under a $(1 - \epsilon)$ speed processor.

Theorem 3 *$\forall t \geq 0$, we verify that:* $\text{RBF}(\tau_i, t) \leq \delta(\tau_i, t) \leq (1 + \frac{1}{k})\text{RBF}(\tau_i, t)$ *where* $k = \lceil \frac{1}{\epsilon} \rceil - 1$.

Proof: We first prove the first inequality: for all $t \in [0, (k-1)T_i - J_i]$

$$\delta(\tau_i, t) = \text{RBF}(\tau_i, t)$$

For $t > (k-1)T_i - J_i$:

$$\delta(\tau_i, t) = C_i + (t + J_i)\frac{C_i}{T_i} = C_i\left(1 + \frac{t + J_i}{T_i}\right)$$

As a consequence:

$$\delta(\tau_i, t) \geq \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i = \text{RBF}(\tau_i, t)$$

We now prove the second inequality of the statement: If $\delta(\tau_i, t) > \text{RBF}(\tau_i, t)$ then since $t > (k-1)T_i - J_i$ then $k - 1$ steps before approximating the request bound function, we verify:

$$\text{RBF}(\tau_i, t) \geq kC_i \tag{4}$$

Furthermore,

$$\delta(\tau_i, t) - \text{RBF}(\tau_i, t) \leq C_i$$

This is obvious if $t \in [0, (k-1)T_i - J_i]$ since $\delta(\tau_i, t) = \text{RBF}(\tau_i, t)$, and if $t > (k-1)T_i - J_i$, then:

$$\begin{aligned} \delta(\tau_i, t) - \text{RBF}(\tau_i, t) &= C_i + (t + J_i)\frac{C_i}{T_i} - \left\lceil \frac{t + J_i}{T_i} \right\rceil C_i \\ &\leq C_i \end{aligned}$$

As a consequence: $\delta(\tau_i, t) \leq \text{RBF}(\tau_i, t) + C_i$ and using inequality (4), we obtain the result:

$$\delta(\tau_i, t) \leq (1 + \frac{1}{k})\text{RBF}(\tau_i, t)$$

As a consequence, both inequalities are verified. ■

Using the same approach presented in [6, 5], we can establish the correctness of approximation.

Theorem 4 *If there exists a time instant* $t \in (0, D_i - J_i]$, *such that* $\widehat{W_i}(t) \leq t$, *then* τ_i *is feasible (i.e.,* $W_i(t) \leq t$*).*

Proof: Directly follows from Theorem 3. ■

Theorem 5 *If* $\forall t \in (0, D_i - J_i]$, $\widehat{W_i}(t) > t$, *then* τ_i *is infeasible on a processor of* $(1-\epsilon)$ *capacity.*

Proof: Assume that $\forall t \in (0, D_i - J_i]$, $\widehat{W_i}(t) > t$, but τ_i is still feasible on a $(1-\epsilon)$ speed processor. Since assuming τ_i to be feasible upon a $(1-\epsilon)$ speed processor, then there must exist a time t_0 such that τ_i: $W_i(t_0) \leq (1-\epsilon)t_0$. But, using Theorem 3 we verify that $\widehat{W_i}(t) \leq (1 + \frac{1}{k})W_i(t)$, where $k = \lceil \frac{1}{\epsilon} \rceil - 1$, then for all $t \in (0, D_i - J_i]$, the condition $\widehat{W_i}(t) > t$ implies that $\forall t \in (0, D_i - J_i]$:

$$W_i(t) > \frac{t}{1 + \frac{1}{k}} > \frac{k}{k+1} t \geq (1-\epsilon)t.$$

As a consequence, a time t_0 such that $W_i(t_0) \leq (1 - \epsilon)t_0$ cannot exist and τ_i is infeasible. ■

To conclude the correctness, we must prove that scheduling points are sufficient.

Theorem 6 *For all* $t \in \widehat{S_i}$ *such that* $\widehat{W_i}(t) > t$, *then we also verify that:* $\forall t \in (0, D_i - J_i], \widehat{W_i}(t) > t$.

Proof: Let t_1 and t_2 be two *adjacent* points in $\widehat{S_i}$ (i.e., $\nexists\ t \in \widehat{S_i}$ such that $t_1 < t < t_2$). Since $\widehat{W_i}(t_1) > t_1$, $\widehat{W_i}(t_2) > t_2$ and the fact that $\widehat{W_i}(t)$ is an non-decreasing step left-continuous function we conclude that $\forall t \in (t_1, t_2)$ $\widehat{W_i}(t) > t$ (see Figure 2 for details). The property follows. ■

4 Approximate Response-Time Analysis with Release Jitter

4.1 Approximate worst-case response time upper bound

According to a accuracy parameter ϵ, we define approximate worst-case response times as in the classical Combinatorial Optimization Problem theory [7]:

Definition 3 *Let* ϵ *be a constant and* R_i *be the worst-case response time of a task* τ_i, *then the approximate worst-case responses time* $\widehat{R_i}$ *satisfies:* $R_i \leq \widehat{R_i} \leq (1+\epsilon)R_i$.

We shall combine results presented in Sections 2 and 3, in order to define approximate worst-case response times. Using the FPTAS presented in Section 3, we can check that a task is feasible or not. If it is feasible, then we are able to compute an upper bound of the worst-case response time of a task as presented in Section 2.

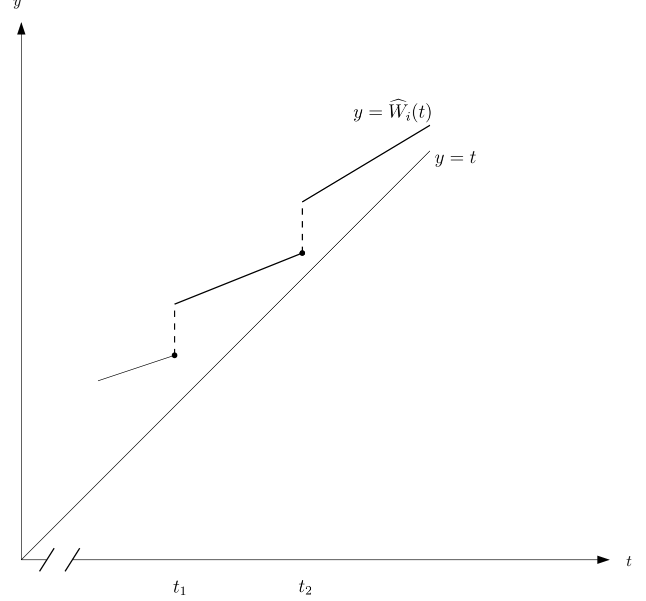


Figure 2. The scheduling points $\widehat{S_i}$ are sufficient

Definition 4 *Consider a task* τ_i *such that there exists a time* t *satisfying* $\widehat{W_i}(t) \leq t$, *then an approximate worst-case response time is defined by:*
$t^* \stackrel{\text{def}}{=} \min \left(t \in \widehat{S_i} \mid \widehat{W_i}(t) \leq t \right)$ *and* $\widehat{R_i} \stackrel{\text{def}}{=} \widehat{W_i}(t^*) + J_i$.

We now prove that such a method defines an upper bound of the worst-case response time of task τ_i.

Theorem 7 *For every task* τ_i *such that there exists a time* t *satisfying* $\widehat{W_i}(t) \leq t$, *then:* $R_i \leq \widehat{R_i}$

Proof: Let t be a scheduling point such that $\widehat{W_i}(t) \leq t$. From the approximate feasibility test, we verify that τ_i is feasible: there exists a time t^* such that $W_i(t^*) \leq t^*$ and $t^* \leq t$. Since $R_i = W_i(t^*) + J_i$ and $\widehat{R_i} = \widehat{W_i}(t) + J_i$ then, it follows from properties of the approximate feasibility test that $R_i \leq \widehat{R_i}$. ■

4.2 The Algorithm

The complete algorithm for computing approximate worst-case response time of a task τ_i is presented in Algorithm 1. The algorithm contains three nested loops. The first loop and the last one are bounded by n (i.e., the number of tasks). The second one is related to k, thus on the value $1/\epsilon$. Thus, this implementation of the approximate feasibility test for a given task leads to a $O(n^2/\epsilon)$ algorithm. This algorithm is eligible to be a FPTAS since it is polynomial in the size of the task set and the accuracy parameter $1/\epsilon$. But, as we will prove in the next section, it does not lead to bounded performance guarantee on computed response times in comparison with an exact response time analysis (performed with a pseudo-polynomial time algorithm).

Algorithm 1. Approximate worst-case response time of τ_i

```
input :
    ε : real                                  /* The FPTAS accuracy parameter */;
    i : integer                               /* Index of the analysed task */;
    n : integer                               /* Size of the task set */;
    C[n], T[n], D[n], J[n] : array of integers /* Task parameters */;
output: Approximate response time of τ_i or 'not feasible upon a (1 − ε)-speed processor';
k = ⌈1/ε⌉ − 1                 /* k is the number of steps considered in rbf(τ_i, t) */;
for j = 1 to i − 1 do
    for ℓ = 1 to k do                                  /* for each scheduling pointt */
        if (ℓ = k and j = i − 1) then t = D[i] − J[i] ; /* t is the last scheduling point */
        else t = ℓ × T[j] − J[j] ;                     /* t is another scheduling points */
        w = C_i                                         /* w is δ(τ_i, t) */;
        for m = 1 to i − 1 do                          /* for all higher priority tasks */
            if (t ≤ (k − 1)T[m] − J[m]) then w+ = C[m]⌈t/T[m]⌉ ;  /* compute rbf(τ_m; t) */
            else w+ = C[m] + (t + J[i])C[m]/T[m] ;     /* compute linear approximation */
        end
        if (t ≥ w)) then return (w + J[i]);            /* approximate response time of τ_i */
    end
end
return ("not feasible upon a (1 − ε)-speed processor");
```

4.3 Worst-case analysis of the algorithm performance guarantee

We now show that this method does not lead to an approximation algorithm (i.e., with the expected bounded error presented in Definition 3) even if the approximate feasibility analysis returns a positive answer.

Theorem 8 *There exist some task systems for which $cR_i \leq \widehat{R_i}$ for any integer c.*

Proof: Let us consider a task system with two tasks with the following parameters: τ_1 with $C_1 = 1-\lambda$ and $T_1 = 1$ and τ_2 with $C_2 = k\lambda$ and $T_2 = k+1/\lambda$, where $0 < \lambda < 1$ and k is an arbitrary integer. (Both tasks have their jitter parameter equal to zero). With these parameters and the Rate-Monotonic scheduling policy, the task τ_2 can only be executed λ unit of time within any interval of length one in the schedule. The τ_2 completes at time k. The approximate feasibility analysis leads to the following computations:

$$\begin{aligned}\widehat{W_2}(t) &= k\lambda + \delta(\tau_1, t)\\ &= k\lambda + (1-\lambda) + t(1-\lambda)\end{aligned}$$

The corresponding approximate worst-case response time will be achieved for $\widehat{W_2}(t) = t$. The approximation switches to a linear approximation at time $(k-1)T_1 = k-1$. The corresponding fixed-point t is:

$$\begin{aligned}t &= k\lambda + (1-\lambda) + t(1-\lambda)\\ t &= k - 1 + \frac{1}{\lambda}\end{aligned}$$

As a consequence the approximate worst-case response time is: $\widehat{R_2} = k - 1 + 1/\lambda$. The approximate feasibility analysis always predicts that the task system is feasible for any integer k since the approximate worst-case response time is strictly less than the deadline of task τ_2. Therefore, the approximate response time is strictly larger than the exact, and can be made arbitrarily large: the ratio between the exact worst-case response time and the approximate one is exactly:

$$\frac{\widehat{R_2}}{R_2} = 1 - \frac{1}{k} + \frac{1}{\lambda k}$$

This ratio increases without any bound as λ approaches zero. So, for any arbitrary integer c, we can find a lambda sufficiently small such that $\widehat{R_2}/R_2 \geq c$. ■

The Figure 3 presents an example of this counterexample with $k = 10, \lambda = 0.1, \epsilon = 0.33$. The exact worst-case response time of τ_2 is 10 and the approximate worst-case response time is 19 (thus τ_2 completes by its deadline equal to 20). Note that the slope of the approximated cumulative request bound function tends to one when λ tends to zero. and thus becomes nearly parallel to the line representing the processor capacity. That is why a performance guarantee can not be achieved using our method.

5 Conclusion and Further Work

We presented a method for approximating worst-case response times of static-priority tasks with release jitter constraints. The method is based on a FPTAS performing a feasibility test based on a Time-Demand Analysis. According to an accuracy parameter ϵ, if the approximate feasibility test concludes that a task τ_i is feasible (i.e., meets its deadline) then we can compute an approximate

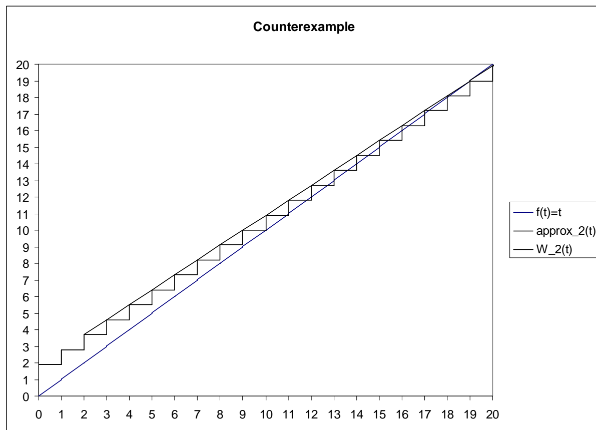


Figure 3. Counterexample with $k = 3, \lambda = 0.1, \epsilon = 0.3$. The exact worst-case response time is 10 and the approximate one is achieved when lines intersect at time 19. Thus, the approximate value is near 2 times greater than the exact wors-case response time. Reducing λ to an arbitrary small value lead to an unbounded performance ratio.

worst-case response time, but without any constant performance guarantee. But, when the approximate feasibility test cannot conclude that τ_i is feasible, we know that τ_i will not be feasible under a processor with capacity $(1 - \epsilon)$; however, the proposed approach cannot guarantee that the approximate worst-case response times are within a constant multiplicative factor of the actual worst-case response time. Even if our results are not complete, they allow to define a sufficient feasibility analysis that can be used for analysing a component in a QoS Optimization method or encapsulated within a holistic analysis for analysing distributed real-time systems.

The existence of an approximation scheme (or weakly an approximation algorithm) is still an interesting open issue. If such a result exists for the worst-case response time analysis, it will exactly quantify the pessimism of the corresponding sufficient feasibility test.

Acknowledgments

The authors would like to thank anonymous reviewers for their helpful comments that allow to improve the presentation of this paper.

References

[1] E. Bini, G. Buttazzo, and G. Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 2003.

[2] R. Bril, W. Verhaege, and E. Pol. Initial values for on-line response time calculations. *proc. Int Euromicro Conf. on Real-Time Systems (ECRTS'03), Porto*, 2003.

[3] A. Burchard and J. Liebeherr. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 1995.

[4] D. Chen, A. Mok, and T. Kuo. Utilization bound revisited. *IEEE Transactions on Computers*, 2003.

[5] N. Fisher and S. Baruah. A polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines. In I. C. Society, editor, *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 117–126, 2005.

[6] N. Fisher and S. Baruah. A polynomial-time approximation scheme for feasibility analysis in static-priority systems with bounded relative deadlines. *Proceedings of the 13th International Conference on Real-Time Systems,Paris, France*, pages 233–249, 2005.

[7] M. Garey and D. Johnson. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman and Company*, 1979.

[8] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. *Technical Report 2966, Institut National de Recherche en Informatique et Automatique (INRIA), France*, 1996.

[9] C. Han and H. Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithm. *proc 18th IEEE Real-Time Systems Symposium (RTSS'97)*, 1997.

[10] M. Joseph and P. Pandya. Finding response times in a real-time systems. *The Computer Journal*, 29(5):390–395, 1986.

[11] S. Lauzac, R.Melhem, and D. Mossé. An improved rate-monotonic admission control and its applications. *IEEE Transactions on Computers*, 2003.

[12] J. Lehoczky. Fixed-priority scheduling of periodic task sets with arbitrary deadlines. *proc. Real-Time System Symposium (RTSS'90)*, pages 201–209, 1990.

[13] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *proc. Real-Time System Symposium (RTSS'89)*, pages 166–171, 1989.

[14] J. C. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[15] Y. Manabee and S. Aoyagi. A feasible decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems Journal*, pages 171–181, 1998.

[16] D.-W. Park, S. Natarajan, A. Kanavsky, and M. Kim. A generalized utilization bound test for fixed-priority real-time scheduling. *proc. 2sd Workshop on Real-Time Systems and Applications*, 1995.

[17] P. Richard. Polynomial time approximate schedulability tests for fixed-priority real-time tasks: some numerical experimentations. *14th Real-Time and Network Systems, Poitiers (France)*, 2006.

[18] M. Sjodin and H. Hansson. Improved response time analysis calculations. *proc. IEEE Int Symposium on Real-Time Systems (RTSS'98)*, 1998.

[19] K. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, 1994.

Schedulability Analysis using Exact Number of Preemptions and no Idle Time for Real-Time Systems with Precedence and Strict Periodicity Constraints

Patrick Meumeu Yomsi
INRIA Rocquencourt
Domaine de Voluceau BP 105
78153 Le Chesnay Cedex - France
Email: patrick.meumeu@inria.fr

Yves Sorel
INRIA Rocquencourt
Domaine de Voluceau BP 105
78153 Le Chesnay Cedex - France
Email: yves.sorel@inria.fr

Abstract

Classical approaches based on preemption, such as RM (Rate Monotonic), DM (Deadline Monotonic), EDF (Earliest Deadline First), LLF (Least Laxity First), etc, give schedulability conditions in the case of a single processor, but assume the cost of the preemption to be negligible compared to the duration of each task. Clearly the global cost is difficult to determine accurately because, if the cost of one preemption is known for a given processor, it is not the same for the exact number of preemptions of each task. Because we are interested in hard real-time systems with precedence and strict periodicity constraints where it is mandatory to satisfy these constraints, we give a scheduling algorithm which counts the exact number of preemptions for each task, and thus leads to a new schedulability condition. This is currently done in the particular case where the periods of all the tasks constitute an harmonic sequence.

1 Introduction

We address here hard real-time applications found in the domains of automobiles, avionics, mobile robotics, telecommunications, etc, where the real-time constraints must be satisfied in order to avoid the occurrence of dramatic consequences [1, 2]. Such applications based on automatic control and/or signal processing algorithms are usually specified with block-diagrams. They are composed of functions producing and consuming data, and each function has a strict period in order to guarantee the input/output rate as it is usually required by the automatic control theory. Consequently, in this paper we study the problem of scheduling tasks onto a single computing resource, i.e. a single processor, where each task corresponds to a function and must satisfy precedence constraints in addition to its strict period. This latter constraint implies that for such a system, any task starts its execution at the beginning of its period. We assume here that no jitter is allowed at the beginning of each task.

Traditional approaches based on preemption, such as RM (Rate Monotonic) [3], DM (Deadline Monotonic) [4], EDF (Earliest Deadline First) [5], LLF (Least Laxity First) [6], etc, give schedulability conditions but always assume the cost of the preemption to be negligible compared to the duration of each task [7, 8]. Indeed, this assumption is due to the Liu & Layland model [9], also called "the classical model", which is the pioneer model for scheduling hard real-time systems. With this model, the authors showed that a system of independent periodic preemptive tasks with the periods of all tasks forming an harmonic sequence [10] [1], is schedulable if and only if:

$$\sum_{i=1}^{n} \frac{C_i^{'}}{T_i} \leq 1 \tag{1}$$

T_i denotes the period and $C_i^{'}$ the inflated worst case execution time (WCET) with the approximation of the cost of the preemption for task τ_i. It is worth noticing that most of the industrial applications in the field of automatic control, image and signal processing consist of tasks with periods forming an harmonic sequence. For example, the automatic guidance algorithm in a missile falls within this case. Actually, expression (1) takes into account the cost due to preemption inside the value of $C_i^{'}$. Thus, $C_i^{'} = C_i + \varepsilon_i^{'}$ where C_i is the value of the WCET without preemption, and $\varepsilon_i^{'}$ is an approximation of the cost ε_i of the preemption for this task, as explicitly stated in [9]. Thus, expression (1) becomes:

$$U + \varepsilon^{'} \leq 1 \tag{2}$$

where

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}, \quad \text{and} \quad \varepsilon^{'} = \sum_{i=1}^{n} \frac{\varepsilon_i^{'}}{T_i}$$

The cost of the preemption for task τ_i is $\varepsilon_i = N_p(\tau_i) \cdot \alpha$, where α denotes the temporal cost of one preemption and $N_p(\tau_i)$ is the exact number of preemptions of task τ_i.

[1] A sequence $(a_i)_{1 \leq i \leq n}$ is harmonic if and only if there exists $q_i \in \mathbb{N}$ such that $a_{i+1} = q_i a_i$. Notice that we may have $q_{i+1} \neq q_i \quad \forall i \in \{1, \cdots, n\}$.

$N_p(\tau_i)$ may depend on the instance of the task according to the relationship between the periods of the other tasks in the system. For example, in the case where the periods of the tasks form an harmonic sequence $N_p(\tau_i)$ does not depend on the instance of τ_i. Therefore, since $\varepsilon_i^{'}$ is an approximation of ε_i and T_i is known, $\varepsilon^{'}$ is an approximation of the global cost ε due to preemption, defined by:

$$\varepsilon = \sum_{i=1}^{n} \frac{N_p(\tau_i) \cdot \alpha}{T_i}$$

If the temporal cost α of one preemption is known for a given processor, it is not the same for the exact number of preemptions $N_p(\tau_i)$ for each task τ_i during a period T_i. Consequently, it becomes difficult to calculate the global cost of the preemption, and thus to guarantee that expression (2) holds. Obviously the approximation of this latter may lead to a wrong real-time execution whereas the schedulability analysis concluded that the system was schedulable. To cope with this problem the designer usually allows margins which are difficult to assess, and which in any case lead to a waste of resources. Note that the worst-case response time of a task is the greatest time, among all instances of that task, it takes to execute each instance from its release time, and it is larger than the WCET when an instance is preempted. A. Burns, K. Tindell and A. Wellings in [11] presented an analysis that enables the global cost due to preemptions to be factored into the standard equations for calculating the worst-case response time of any task, but they achieved that by considering the maximum number of preemptions instead of the exact number. Juan Echagüe, I. Ripoll and A. Crespo also tried to solve the problem of the exact number of preemptions in [12] by constructing the schedule using idle times and counting the number of preemptions. But, they did not really determine the execution overhead incurred by the system due to these preemptions. Indeed, they did not take into account the cost of each preemption during the scheduling. Hence, this amounts to considering only the minimum number of preemptions since some preemptions are not considered: those due to the increase in the execution time of the task because of the cost of the preemptions themselves.

For such a system of tasks with strict periodicity and precedence constraints, we propose a method to calculate on the one hand the exact number of preemptions and thus the accurate value of ε, and on the other hand the schedule of the system without any idle time, i.e. the processor will always execute a task as soon as it is possible to do so. Although idle time may help the system to be schedulable, when idle time is forbidden it is easier to find the start times of all the instances of a task according to the precedence relation.

The proposed method leads to a much stronger schedulability condition than expression (1). Moreover, we do this in the case where tasks are subject to precedence and strict periodicity constraints, using our previous model [13] that is well suited to the applications we are interested in. Afterwards, to clearly distinguish between the specification level and its associated model, we shall use the term *operation* instead of the commonly used "*task*" [14] which is too closely related to the implementation level.

The paper is structured as follows: Section 2 describes the model and gives notations used throughout this paper. Section 3 restricts the study field thanks on the one hand to properties on the strict periods, and on the other hand to properties on WCETs. Section 4 proposes a scheduling algorithm which counts the exact number of preemptions, and derives a schedulability condition, in the case where the periods of all operations constitute an harmonic sequence. We conclude and propose future work in section 5.

2 Model

The model depicted in figure 1 is an extension, with preemption, of our previous model [13] for systems with precedence and strict periodicity constraints executed on a single processor.

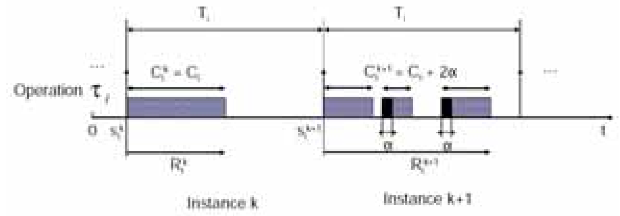


Figure 1. Model

Here are the notations used in this model assuming all timing characteristics are non-negative integers, i.e. they are multiples of some elementary time interval (for example the "CPU tick", the smallest indivisible CPU time unit):

$\tau_i = (C_i, T_i)$: An operation
T_i: Period of τ_i
C_i: WCET of τ_i without preemption, $C_i \leq T_i$
τ_i^k: The kth instance of τ_i
α: Temporal cost of one preemption for a given processor
$N_p(\tau_i^k)$: Exact number of preemptions of τ_i in τ_i^k
$C_i^k = C_i + N_p(\tau_i^k) \cdot \alpha$: Exact WCET of τ_i including its preemption cost in τ_i^k
s_i^0: Start time of the first instance of τ_i
$s_i^k = s_i^0 + (k-1)T_i$: Start time of the kth instance of τ_i
R_i^k: Response time of the kth instance of τ_i
R_i: Worst-case response time of τ_i
$T_i \wedge T_j$: The greatest common divisor of T_i and T_j, when $T_i \wedge T_j = 1$, T_i and T_j are co-prime
$\tau_i \prec \tau_j$: $\tau_i \longrightarrow \tau_j$, τ_i precedes τ_j

We denote by V the set of all systems of operations. Each system consists in a given number of operations, with precedence and strict periodicity constraints. Each

operation τ_i of a system in V consists of a pair (C_i, T_i): C_i its WCET and T_i its period.

The precedence constraints are given by a partial order on the execution of the operations. $\tau_i \prec \tau_j$ means that the start time s_j^0 of the first instance of τ_j cannot occur before the first instance of τ_i, started at s_i^0, is completed. This precedence relation between operations also implies that $s_i^k \leq s_j^k, \forall k \geq 1$ thanks to the result given in [15]. In that paper it has been proven that given two operations $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$:

$$\tau_i \prec \tau_j \Longrightarrow T_i \leq T_j$$

Regarding the latter relation from the practical point of view, it is worth noticing that when the precedence relations are due to data transfers and the periods of the operations exchanging data constitute an harmonic sequence, the number of operations producing data between two consecutive operations consuming the corresponding data, is constant. Consequently, the number of buffers used to actually achieve the data exchange is bounded, i.e. it cannot increase indefinitely.

The strict periodicity constraint means that two successive instances of an operation are **exactly** separated by its period: $s_i^{k+1} - s_i^k = T_i \quad \forall k \in \mathbb{N}, \quad \forall i \in \{1, \cdots, n\}$, and no jitter is allowed. In this model the start time is always equal to the release time, in contrast to Liu & Layland's classical model. A great advantage of the strict periodicity constraint for each task is that it is only necessary to focus on the start time of the first instance, the other being directly obtained from it.

It is fundamental to note that, because of the strict periodicity constraint and the fact that we are dealing with the single processor case, any two instances of any two operations of the system cannot start their executions at the same time.

3 Study field restriction

Firstly, we eliminate all the systems where the start times of any two instances of any two operations are identical. This will be achieved thanks to properties on the strict periods of the operations, using the *Bezout theorem*. This is formally expressed through both theorems given in section 3.1. Secondly, we eliminate all the systems where the start time of any instance of an operation occurs while the processor is occupied by a previously scheduled operation thanks to properties on WCETs of the operations. This is formally expressed through the theorem given in section 3.2. These three theorems give sufficient non-schedulability conditions. For the remaining systems of operations, we adopt a constructive approach which consists in building, i.e. in predicting, all the possible preemptive schedules without any idle time. In so far, as we are dealing with hard real-time systems whose main feature is predictability, constructive techniques are better suited than simulation techniques based on tests that are seldom exhaustive. In addition, an exhaustive simulation assumes that there exists a scheduling algorithm, e.g. RM or DM, which is used to perform the simulation. In our case we propose a scheduling algorithm which determines if the system is schedulable and provides the schedule.

3.1 Restriction due to strict periodicity

Theorem 1

Given a system of n operations in V, if there are two operations $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$ with $(\tau_i \prec \tau_j)$ starting their executions respectively at the dates s_i^0 and s_j^0 such that

$$T_i \wedge T_j = 1 \tag{3}$$

then the system is not schedulable. Moreover, any additional assumption (for example preemption and idle times) on the system intending to satisfy all the constraints is of no interest in this case.

Proof: The proof of this theorem uses the Bezout theorem and is detailed in [16]. ■

Theorem 2

Given a system of n operations in V, if there are two operations $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$ with $(\tau_i \prec \tau_j)$ starting their executions respectively at the dates s_i^0 and s_j^0 such that

$$T_i \wedge T_j \mid (s_j^0 - s_i^0) \tag{4}$$

then the system is not schedulable. Moreover any additional assumption on the system intending to satisfy all the constraints is of no interest in this case.

Proof: The proof of this theorem also uses the Bezout theorem and is detailed in [16]. ■

Theorems 1 and 2 give non-schedulability conditions for systems with strict periodicity constraints when both previous relations on the strict periods hold. Moreover, any additional assumption on the system would be useless because of the identical start times of two instances of at least two operations.

We denote by Ω_λ the sub-set of V excluding the cases where the strict periods of the operations verify both previous relations.

$$\Omega_\lambda = \{\{(C_i, T_i)\}_{1 \leq i \leq n} \in V \,/\, \forall i, j \in \{1, \cdots, n\} \\ \exists \lambda > 1, T_i \wedge T_j = \lambda \text{ and } \lambda \nmid (s_j^0 - s_i^0)\}$$

3.2 Restriction due to WCET

The scheduling analysis of a system of preemptive tasks (operations) has shown its importance in a wide range of applications because of its flexibility and its relatively easy implementation [17]. Although preemptions may allow schedules to be found that could not be found without it, it can, unfortunately, cause non schedulability of the system due to its global cost.

Since, given two operations $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$ we have $\tau_i \prec \tau_j \Longrightarrow T_i \leq T_j$ thus, the operations must be scheduled in an increasing order of their periods

corresponding to classical fixed priorities. In other words the smaller the period of an operation is, the greater its priority is, like in the RM scheduling. Note that the scheduling analysis of a system of preemptive tasks with fixed priorities has been a pivotal basis in real-time application development since the work of *Liu* and *Layland* [9]. Now, we assume that any operation of the system may only be preempted by those previously scheduled, and that any operation is scheduled as soon as the processor is free, i.e. no idle time is allowed between the end of the first instance of an operation and the start time of the first instance of the next operation relatively to $\prec$. This assumption about no idle time allows the greatest possible utilization factor of the processor to be achieved. Therefore, to schedule an operation τ_i relatively to those previously scheduled, amounts to filling available spaces in the scheduling with corresponding slices of the exact WCET of τ_i. Consequently, from the point of view of operation τ_i the start time s_i^0 of its first instance is yielded by the end of the first instance of τ_{i-1}. Thus, the notion of release time of τ_i is not relevant in this paper, or is equal to s_i^0.

A *potential schedule* S of a system is given by a list of the start times of the first instance of all the operations:

$$S = \{(s_1^0, s_2^0, \cdots, s_n^0)\} \quad (5)$$

The start times s_i^k ($k \geq 1$, $i = 1 \cdots n$) of the other instances of operation τ_i are directly deduced from the first one, and this advantage derives directly from the model. The *response time* R_i^k of the kth instance of operation $\tau_i = (C_i, T_i)$ is the time elapsed between its start time s_i^k and its end time. This latter takes into account the preemption thus,

$$R_i^k \geq C_i \quad \forall k.$$

We call R_i the *worst response time* of operation τ_i, defined as the maximum of the response times of all its instances.

These definitions enable us to say that, in order to satisfy the strict periodicity, any operation $\tau_i = (C_i, T_i)$ of a potentially schedulable system in Ω_λ must satisfy:

$$R_i \leq T_i \quad \forall i \in \{1, \cdots, n\} \quad (6)$$

We say that a system in Ω_λ has one *overlapping* when the start time of any instance of a given operation occurs while the processor is occupied by a previously scheduled operation. Such systems are not schedulable, as expressed in the following theorem.

Theorem 3

Given a system of n operations in Ω_λ, if there are two operations $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$ with $(\tau_i \prec \tau_j)$ starting their executions respectively at the dates s_i^0 and s_j^0 such that for $k \geq 1$

$$\exists\, \beta < k \text{ and } 0 \leq (s_j^0 + \beta T_j) - (s_i^0 + (k-1)T_i) < R_i^k \quad (7)$$

then the system is not schedulable. Moreover any additional assumption on the system intending to satisfy all the constraints is of no interest in this case.

Proof: The proof of this theorem derives directly from the assumption that an operation may only be preempted by those previously scheduled, and it is detailed in [16]. An example is given below (see figure 2). ∎

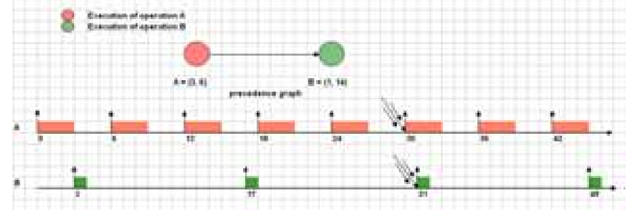


Figure 2. System with an overlapping

Now we can partition Ω_λ into the three following disjoint subsets: the subset V_c of systems with overlappings which are not schedulable thanks to theorem 3, the subset V_r of systems with regular operations, i.e. where the periods of all the operations constitute an harmonic sequence, and the subset V_i of systems with irregular operations. Thus, since the subset of operations where $T_i \wedge T_j = 1$ holds, the subset of operations where $T_i \wedge T_j \mid (s_j^0 - s_i^0)$ holds, and the subset V_c are not schedulable, only the remaining subsets V_r and V_i are potentially schedulable (see figure 3).

$$V_c = \{\{(C_i, T_i)\}_{1 \leq i \leq n} \in \Omega_\lambda \,/\, \exists i \in \{1, \cdots, n-1\}, \exists j \in \{i+1, \cdots, n\} \text{ and } 0 \leq (s_j^0 + \beta T_j) - (s_i^0 + (k-1)T_i) < R_i^k, \quad k \geq 1; \beta \in \mathbb{N}\}$$

$$V_r = \{\{(C_i, T_i)\}_{1 \leq i \leq n} \in \Omega_\lambda \,/\, T_1 \mid T_2 \mid \cdots \mid T_n\}$$

$$V_i = \Omega_\lambda \backslash (V_c \cup V_r)$$

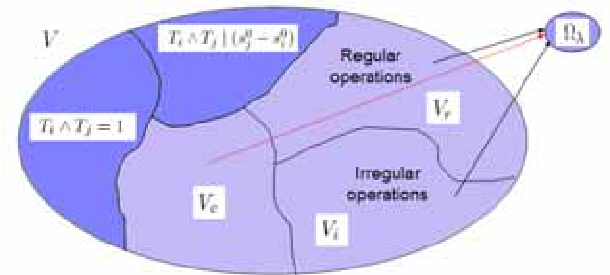


Figure 3. Ω_λ**-partitioning**

In the remainder of this paper, we restrict our scheduling analysis to the subset V_r.

4 Scheduling analysis for V_r

Given any system in V_r, both the exact WCET C_i^k and the response time R_i^k of the kth instance of a given operation τ_i are the same for all its instances, $C_i^k = C_i^* =$

$C_i + N_p(\tau_i) \cdot \alpha$ and $R_i^k = R_i$ (equal to the worst response time R_i of the operation) because the number of available spaces left in each instance does not depend on the instance itself. Therefore it is worth, in this case, noticing that it is sufficient to give a schedulability condition for the first instance of each operation.

We call U_p (respectively U_p^*) the pth *temporary load factor* (respectively the *exact* p^{th} *temporary load factor*) of the processor ($1 \leq p \leq n$) for a system of n operations $\{\tau_i = (C_i, T_i)\}_{1 \leq i \leq n}$ in V_r.

$$U_p = \sum_{i=1}^{p} \frac{C_i}{T_i} \quad \text{and} \quad U_p^* = \sum_{i=1}^{p} \frac{C_i^*}{T_i} = U_p + \sum_{i=1}^{p} \frac{N_p(\tau_i) \cdot \alpha}{T_i}$$

This system will be said to be *potentially schedulable* if and only if:

$$U_n \leq 1 \tag{8}$$

and *schedulable* if and only if:

$$U_n^* \leq 1 \tag{9}$$

Notice that in (8), C_i is the WCET of operation τ_i without preemption. From now on, we assume (8) is always satisfied.

We say that the exact WCET $C_i^* = C_i + N_p(\tau_i) \cdot \alpha$ of an operation $\tau_i = (C_i, T_i)$ of a system in V_r is a *critical* WCET if its scheduling causes a temporal *delay* to the start time of the first instance of operation $\tau_{i+1} = (C_{i+1}, T_{i+1})$, $\tau_i \prec \tau_{i+1}$. In other words, this means from the point of view of operation τ_i that C_i^* is critical when $s_{i+1}^0 > s_i^0 + R_i^1$, see figure 4. Indeed, in this case the last slice of the exact WCET of τ_i exactly fits the next available space in the scheduling, and thus the first instance of the next operation relatively to $\prec$ cannot start exactly at the end of the first instance of τ_i.

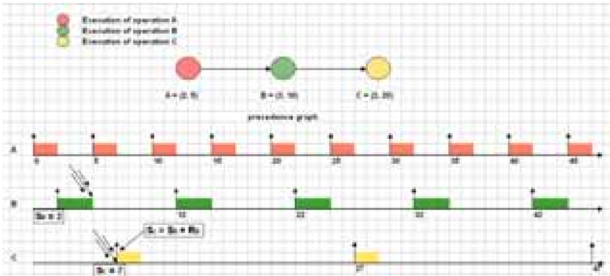


Figure 4. Operation with a critical WCET

In order to make it easier to understand the general case, we first study the simpler case of only two operations. Both cases are based on the same principle which consists, for an operation, in filling available spaces left in each instance with slices of its exact WCET taking into account the cost of the exact number of preemptions necessary for its scheduling.

4.1 System with two operations

We consider $\tau_1 = (C_1, T_1)$ and $\tau_2 = (C_2, T_2)$ to be a system with two operations in V_r such as $T_1 \mid T_2$.

To be consistent with what we have presented up to now, we will first schedule τ_1, and then τ_2, $\tau_1 \prec \tau_2$. Hence, since no idle time is allowed between the end of the first instance of τ_1 and the start time of the first instance of τ_2, we have:

$$C_1^* = C_1 \quad \text{and thus} \quad R_1 = C_1 \quad \text{and} \quad s_2^0 = s_1^0 + R_1 \tag{10}$$

Without any loss of generality, we assume in the remainder of this paper that $s_1^0 = 0$. Because the system is potentially schedulable, we have:

$$\left(\left\lceil \frac{R_1 + T_2}{T_1} \right\rceil - 1 \right) \cdot C_1^* + C_2 \leq T_2, \tag{11}$$

i.e. operation τ_2 is schedulable without taking into account the cost of the preemption.

Now, on the one hand, if:

$$C_1 + C_2 \leq T_1$$

then operation τ_2 is schedulable without any preemption, and we have:

$$C_2^* = C_2 \quad \text{and} \quad R_2 = C_2 \tag{12}$$

On the other hand, if:

$$C_1 + C_2 > T_1 \tag{13}$$

then the system requires at least one preemption of operation τ_2 to be schedulable. To compute the exact number of preemptions $N_p(\tau_2)$, we perform the algorithm below, using a sequence of Euclidean divisions.

We denote $e = T_1 - C_1$ and we initialize $C^1 = C_2$. The Euclidean division of C^1 by e gives:

$$C^1 = q_1 \cdot e + r_1 \text{ with } q_1 = \left\lfloor \frac{C^1}{e} \right\rfloor \text{ and } 0 \leq r_1 < e$$

For all $k \geq 0$, we compute

$$C^{k+1} = r_k + q_k \cdot \alpha \tag{14}$$

and at each step, we perform the Euclidean division of C^{k+1} by e which gives:

$$C^{k+1} = q_{k+1} \cdot e + r_{k+1} \text{ with } q_{k+1} = \left\lfloor \frac{C^{k+1}}{e} \right\rfloor \text{ and } 0 \leq r_{k+1} < e$$

We stop the algorithm as soon as: either there exists $m_1 \geq 1$ such that $\sum_{i=1}^{m_1} q_i \cdot e > T_2(1 - U_1^*)$, and thus the operation τ_2 is not schedulable in this case, or

$$\exists m_2 \geq 1 \quad \text{such that} \quad C^{m_2} \leq e \tag{15}$$

and thus, $N_p(\tau_2)$ is given by:

$$N_p(\tau_2) = \sum_{i=1}^{m_2 - 1} q_i \tag{16}$$

Hence

$$C_2^* = C_2 + N_p(\tau_2) \cdot \alpha \quad (17)$$

and the worst response time R_2 of the operation τ_2 is given by:

$$R_2 = R_2^0 - s_2^0 \quad (18)$$

where:

$$R_2^0 = C_2^* + \left\lceil \frac{R_2^0}{T_1} \right\rceil \cdot C_1^* \quad (19)$$

R_2^0 is easily obtained by using a fixed point algorithm according to:

$$\begin{cases} R_2^{0,l+1} = & C_2^* + \left\lceil \frac{R_2^{0,l}}{T_1} \right\rceil \cdot C_1^* \quad \forall l \geq 0 \\ R_2^{0,0} = & C_2^* \end{cases} \quad (20)$$

The algorithm is stopped as soon as two successive terms of the iteration are equal:

$$R_2^{0,l+1} = R_2^{0,l}, \quad l \geq 0 \quad (21)$$

To simplify the notation, the worst response time will be written as:

$$R_2 = \left\{ R_2^0 = C_2^* + \left\lceil \frac{R_2^0}{T_1} \right\rceil \cdot C_1^* \right\} - s_2^0 \quad (22)$$

Therefore a necessary and sufficient schedulability condition for operation τ_2, and thus for the system $\{\tau_1, \tau_2\}$ taking into account the cost of the preemption is given by:

$$U_2^* \leq 1 \quad \text{i.e.,} \quad U_2 + \frac{N_p(\tau_2) \cdot \alpha}{T_2} \leq 1 \quad (23)$$

Example 1

Let τ_1 and τ_2 be a system with two operations in V_r with the characteristics defined in table 1:

Table 1. Characteristics of example 4.1

	C_i	T_i
τ_1	2	5
τ_2	4	10

We have: $U_2 = \frac{2}{5} + \frac{4}{10} = 0.8$ and $e = 3$.

As operation τ_1 is never preempted, its worst response time R_1 is equal to its worst-case execution time: $R_1 = C_1^* = C_1 = 2$.

Because $\tau_1 \prec \tau_2$, these operations are schedulable if and only if preemption is used (is mandatory).

Although it is not realistic, let $\alpha = 1$ be the cost of one preemption for the processor in order to show clearly the impact of the preemption. Since $C_1 + C_2 = 6 > T_1 = 5$, the computation of $N_p(\tau_2)$ is summarized in the table below:

Therefore, there is only one preemption $N_p(\tau_2) = 1$ (see figure 5) and $C_2^* = 4 + 1 \cdot 1 = 5$

According to (20), $R_2^0 = 9$, and the worst response time R_2 of operation τ_2 is given by:

Table 2. computation of $N_p(\tau_2)$

Steps	q_i	C^i	r_i
1	1	4	1
2	0	2	2

$$R_2 = 9 - 2 = 7 \quad \text{and we have} \quad R_2 \leq T_2 = 10$$

Thus the system is schedulable because:

$$U_2^* = U_2 + \frac{N_p(\tau_2) \cdot \alpha}{T_2} = 0.9 \leq 1.$$

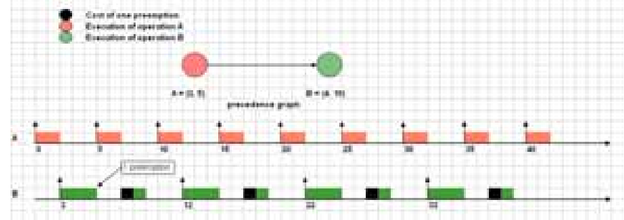


Figure 5. Scheduling of two operations

4.2 System with $n > 2$ operations

The strategy we will adopt in this section of calculating the exact number of preemptions for an operation is different from the one used in the previous section, because we can no longer perform a simple Euclidean division. Although, we can perform the Euclidean division to find the number of preemptions for the second operation, this technique cannot be usable for a third operation, and so on. Actually, the available spaces left after having scheduled the second operation may not be equal, as shown in example 4.2 below, see figure 6.

Example 2

Let $\alpha = 1$ and $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ be a system with four operations in V_r with the characteristics defined in table 3:

Table 3. Characteristics of example 4.2

	C_i	T_i
τ_1	2	5
τ_2	1	10
τ_3	3	20
τ_4	3	40

The schedule is depicted in figure 6.

In figure 6, it can be seen that after the scheduling of the first operation, the available spaces left have equal lengths (3 time units) but it is no longer the case after the scheduling of the second operation, and thus for the third operation after the scheduling of the second operation, and so on.

The intuitive idea of our algorithm consists in two main steps for each operation, according to the precedence relation. First, determine the total number of available time units in each instance, and then the lengths of each available space (consecutive available time units). These data

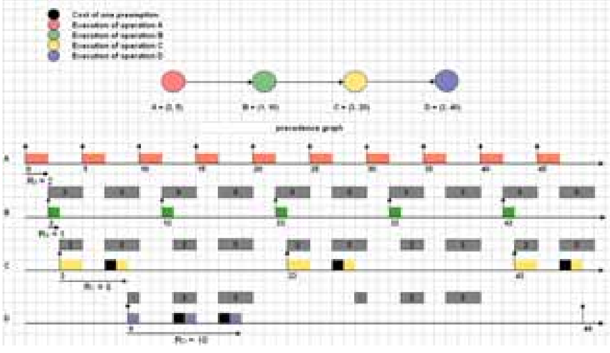


Figure 6. Difficulty of using a simple Euclidean division

allow the computation of the instants when the preemptions occur. A preemption occurence corresponds to the switch from an available time unit to an already executed one. Second, for each potentially schedulable operation, fill available spaces with slices of its WCET up to the value of its WCET, and then add the cost of the preemptions ($p \cdot \alpha$ for p preemptions) to the current inflated WCET, taking into account the increase in the execution time of the operation because of the cost of the preemptions themselves. Finally, the last inflated WCET corresponds to the exact WCET. Thus, it is possible to verify the schedulability condition and then whether this operation is schedulable.

Notice that the number of available spaces is the same for all the instances of an operation, thus it is only necessary to verify the schedulability condition in the first instance which is bounded by the period of the operation. In addition, this verification is performed only once for each operation. Consequently, the complexity of the algorithm even though it has not been yet computed precisely, will actually not explode.

Before going through our proposed algorithm, let us make some assumptions:

1. we will add the cost due to the preemptions to the scheduling analysis of a system if and only if the system is already schedulable without taking it into account, that is $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.

2. we have scheduled the first $j-1$ ($2 \leq j \leq n-1$) operations, and we are about to schedule the jth operation,

3. we have potentially enough available spaces to schedule operation τ_j, that is to say:

$$\sum_{i=1}^{j-1} \left(\left\lceil \frac{s_j^0 + T_j}{T_i} \right\rceil - \left\lceil \frac{s_j^0}{T_i} \right\rceil \right) \cdot C_i^* + C_j \leq T_j$$

Under assumption 2, if F_j denotes the number of available time units left in each instance of the operation τ_j, then we have:

$$F_j = T_j \cdot (1 - U_{j-1}^*) \tag{24}$$

Therefore, the operation $\tau_j = (C_j, T_j)$ is schedulable if and only if:

$$0 < C_j^* \leq F_j \quad \text{i.e.,} \quad C_j^* \in \{1, \cdots, F_j\} \tag{25}$$

Let:

$$L_j = \{1, \cdots, F_j\} \tag{26}$$

L_j denotes the set of all the possible exact WCET C_j^* of operation $\tau_j = (C_j, T_j)$. Thus, it also contains all the possible WCETs for operation τ_j. Once (25) is satisfied, the worst response time of τ_j is given by:

$$R_j = \left\{ R_j^0 = C_j^* + \sum_{i=1}^{j-1} \left\lceil \frac{R_j^0}{T_i} \right\rceil \cdot C_i^* \right\} - s_j^0 \tag{27}$$

and R_j is obtained by using a fixed point algorithm similar to the one given in the previous section, used to obtain R_2.

4.3 Scheduling algorithm

Hereafter is the scheduling algorithm which counts the exact number of preemptions in order to accurately take into account its cost in the schedulability condition. It has the twelve following steps.

1: Determine the start time s_j^0 of the first instance of operation $\tau_j = (C_j, T_j)$ according to whether the exact WCET $C_{j-1}^* = C_{j-1} + N_p(\tau_{j-1}) \cdot \alpha$ of operation $\tau_{j-1} = (C_{j-1}, T_{j-1})$ is critical or not.

2: Calculate the number of available time units F_j left in each instance of τ_j, and build the set L_j thanks to relations (24) and (25).

3: Make a first ordered partition of L_j in $k_{j-1} = \frac{T_j}{T_{j-1}}$ sub-sets of equal cardinals such that:

$$L_j = L_j^1 \cup L_j^2 \cup \cdots \cup L_j^{k_{j-1}} \quad \text{with}$$

$$\begin{array}{|l} L_j^1 = \left\{ 1, \cdots, \frac{F_j}{k_{j-1}} \right\} \\ L_j^2 = \left\{ \frac{F_j}{k_{j-1}} + 1, \cdots, 2\frac{F_j}{k_{j-1}} \right\} \\ \vdots \\ L_j^{k_{j-1}} = \left\{ (k_{j-1} - 1)\frac{F_j}{k_{j-1}} + 1, \cdots, F_j \right\} \end{array}$$

4: For each subset L_j^i obtained in the previous step, make, if possible, a second ordered partition in h_{j-1} subsets such that:

$$L_j^i = L_j^{i,1} \cup L_j^{i,2} \cup \cdots \cup L_j^{i,h_{j-1}}; \quad i = 1, \cdots, k_{j-1}$$

where the cardinal of each $L_j^{i,\sigma}$ with $2 \leq \sigma \leq h_{j-1}$ equals the cardinal of the subset at the same position in the partition of L_{j-1} starting from the subset with the greatest pair (k_{j-2}, h_{j-2}) of indices (the subset the furthest on the right).

To make this step clear, let us give an example with $(k_{j-2}, h_{j-2}) = (2,2)$.
Let the partition of L_{j-1} be such that:

$$\begin{aligned} L_{j-1} &= L_{j-1}^{1,1} \cup L_{j-1}^{1,2} \cup L_{j-1}^{2,1} \cup L_{j-1}^{2,2} \\ &= \{1,2\} \cup \{3,4,5\} \cup \{6,7\} \cup \{8,9,10\} \end{aligned}$$

and let L_j, and k_{j-1} be such that:

$$\begin{cases} L_j = \{1,2,3,4,5,6,7,8,9,10,11,12\} \\ k_{j-1} = 2 \end{cases}$$

Thanks to step 3, we have:

$$L_j = L_j^1 \cup L_j^2$$

where
$L_j^1 = \{1,2,3,4,5,6\}$ and $L_j^2 = \{7,8,9,10,11,12\}$
In step 4, we obtain:

$$\begin{aligned} L_j^1 &= L_j^{1,1} \cup L_j^{1,2} \cup L_j^{1,3} \\ &= \{1\} \cup \{2,3\} \cup \{4,5,6\} \\ L_j^2 &= L_j^{2,1} \cup L_j^{2,2} \cup L_j^{2,3} \\ &= \{7\} \cup \{8,9\} \cup \{10,11,12\} \end{aligned}$$

Thus, at the end of step 4, we can write:

$$L_j = \bigcup_{i=1}^{k_{j-1}} \left\{ \bigcup_{\sigma=1}^{h_{j-1}} L_j^{i,\sigma} \right\} \tag{28}$$

5: Set:

$$\begin{aligned} &\overline{0} = L_j^{1,1} \\ &\overline{1} = L_j^{1,2} \\ &\vdots \\ &\overline{h_{j-1}-1} = L_j^{1,h_{j-1}} \\ &\overline{h_{j-1}} = L_j^{2,1} \\ &\vdots \\ &\overline{2h_{j-1}-1} = L_j^{2,h_{j-1}} \\ &\overline{2h_{j-1}} = L_j^{3,1} \\ &\vdots \\ &\overline{k_{j-1}h_{j-1}-1} = L_j^{k_{j-1},h_{j-1}} \end{aligned}$$

$\overline{\theta}$ denotes the subset of the possible exact WCETs C_j^* of operation τ_j, preempted θ times. Because operation τ_j is potentially schedulable, thus:

$$\exists \theta_1 \in \{0,1,\cdots,k_{j-1}h_{j-1}-1\} \quad \text{and} \quad C_j \in \overline{\theta_1} \tag{29}$$

If $\theta_1 = 0$, then $N_p(\tau_j) = 0$. If it is not the case, i.e. $\theta_1 \neq 0$, thus we obtain for operation τ_j the exact number of preemptions $N_p(\tau_j)$ using the algorithm below:
We initialize

$$\begin{cases} C^1 = C_j \\ q_1 = \theta_1 \\ A^1 = \sum_{k=0}^{\theta_1 - 1} card(\overline{k}) \\ r_1 = C^1 - A^1 \end{cases}$$

For $l \geq 1$, we compute:

$$B^{l+1} = \sum_{k=1}^{l} A^k + (r_l + \theta_l \cdot \alpha) \tag{30}$$

<u>If $B^{l+1} \leq F_j$,</u> thus $\exists \theta_{l+1} \geq 0$ such that $B^{l+1} \in \overline{\theta_1 + \cdots + \theta_{l+1}}$. If $\theta_{l+1} = 0$, then expression (31) holds with $m_2 = l+1$ and $N_p(\tau_j)$ is given by (32), else we set:

$$\begin{cases} C^{l+1} = r_l + \theta_l \cdot \alpha \\ q_{l+1} = \theta_{l+1} \\ A^{l+1} = \sum_{k=\theta_1+\cdots+\theta_l}^{\theta_1+\cdots+\theta_{l+1}-1} card(\overline{k}) \\ r_{l+1} = C^{l+1} - A^{l+1} \end{cases}$$

The algorithm is stopped as soon as: either there exists $m_1 \geq 1$ such that $B^{m_1} > F_j$, and thus operation τ_j is not schedulable in this case, or

$$\exists m_2 \geq 1 \quad \text{such that} \quad \theta_{m_2} = 0 \tag{31}$$

and therefore:

$$N_p(\tau_j) = \sum_{k=1}^{m_2-1} q_k \tag{32}$$

We compute the exact WCET C_j^* of operation τ_j:

$$C_j^* = C_j + N_p(\tau_j) \cdot \alpha \tag{33}$$

6: Determine the set I_j of all the possible critical exact WCETs C_j^* of operation $\tau_j = (C_j, T_j)$. Each element of I_j is the maximum of each subset $L_j^{i,\sigma}$, except F_j, with $(1 \leq i \leq k_{j-1})$ and $(1 \leq \sigma \leq h_{j-1})$ obtained in step 4.
We distinguish between two types of critical exact WCETs: *critical exact WCET of the first order* and *critical exact WCET of the second order*.
Critical exact WCET of the first order consists of the ordered set I_j^1 given by:

$$\begin{aligned} I_j^1 &= \left\{ max(L_j^i) \quad \text{for} \quad 1 \leq i \leq k_{j-1} \right\} \setminus \{F_j\} \\ &= \left\{ \frac{F_j}{k_{j-1}}, 2\frac{F_j}{k_{j-1}}, \cdots, (k_{j-1}-1)\frac{F_j}{k_{j-1}} \right\} \end{aligned}$$

Critical exact WCET of the second order consists of the ordered set I_j^2 given by:

$$\begin{aligned} I_j^2 &= \bigcup_{i=1}^{k_{j-1}} I_j^{2,i} \quad \text{with} \\ I_j^{2,i} &= \left\{ max(L_j^{i,\sigma}) \quad \text{for} \quad 1 \leq \sigma \leq h_{j-1} \right\} \setminus I_j^1 \end{aligned}$$

Hence $I_j = I_j^1 \cup I_j^2$ and can be rewritten as the ordered set defined by:

$$\begin{aligned} I_j = I_j^{2,1} \cup \left\{ \frac{F_j}{k_{j-1}} \right\} \cup I_j^{2,2} \cup \left\{ 2\frac{F_j}{k_{j-1}} \right\} \cup \cdots \\ \cdots \cup \left\{ (k_{j-1}-1)\frac{F_j}{k_{j-1}} \right\} \cup I_j^{2,k_{j-1}} \end{aligned} \tag{34}$$

Again, to make this step clear, let us give an example, using the same one as in step 4. In this step we obtain:

$$I_j^1 = \{max(L_j^1), max(L_j^2)\} \setminus \{12\} = \{6\}$$

$$I_j^{2,1} = \left\{max(L_j^{1,\sigma}), \quad 1 \le \sigma \le 3\right\} \setminus \{6\} = \{1,3\}$$

$$I_j^{2,2} = \left\{max(L_j^{2,\sigma}), \quad 1 \le \sigma \le 3\right\} \setminus \{6\} = \{7,9\}$$

Thus, by writing I_j like in expression (34) we obtain:

$$I_j = \{1,3\} \cup \{6\} \cup \{7,9\}$$

7: Determine whether C_j^* is a critical WCET, i.e. $C_j^* \in I_j$, or not, thanks to step 6.

8: Determine the delay $\Lambda_j(C_j)$ that operation τ_j will cause to the start time of the first instance of operation $\tau_{j+1} = (C_{j+1}, T_{j+1})$. There are three possible cases for C_j^*:

- $C_j^* \in L_j \setminus I_j$, i.e. C_j^* is not a critical exact WCET, then:
$$\Lambda_j(C_j) = 0 \tag{35}$$
- $C_j^* \in I_j^1$, i.e. C_j^* is a critical exact WCET of the first order, then:
$$\Lambda_j(C_j) = s_j^0 \tag{36}$$
- $C_j^* \in I_j^2$, i.e. C_j^* is a critical exact WCET of the second order, then:
$$\Lambda_j(C_j) = \Lambda_{j-1}(C_{j-1}^0) \tag{37}$$
such that for each possible value $C_j^{0,i} \in I_j^{2,i}$ of C_j^* with $(1 \le i \le k_{j-1})$,
$$\Lambda_j(C_j^{0,i}) = \Lambda_{j-1}(C'_{j-1})$$
where $C'_{j-1} \in I_{j-1}$ and C'_{j-1} is at the same position in I_{j-1} written as in (34) as $C_j^{0,i}$ in $I_j^{2,i}$, starting in I_{j-1} from its maximum which belongs to the sub-set with the greatest pair $(2, k_{j-2})$ of indices $I_{j-1}^{2,k_{j-2}}$ (the subset the furthest on the right).

Again, to make this step clear, let us give an example, using that of step 4. Thanks to everything we have presented up to now,

$$I_{j-1} = I_{j-1}^{2,1} \cup \{5\} \cup I_{j-1}^{2,2} = \{2\} \cup \{5\} \cup \{7\}$$

if we assume we had:
$\Lambda_{j-1}(5) = s_{j-1}^0$ and $\Lambda_{j-1}(2) = \Lambda_{j-1}(7) = s_{j-2}^0$, then as

$$I_j = \{1,3\} \cup \{6\} \cup \{7,9\}$$

In this step we obtain:

$$\begin{cases} \Lambda_j(6) = s_j^0 \quad \text{because} \quad 6 \in I_j^1 \\ \Lambda_j(3) = \Lambda_j(9) = \Lambda_{j-1}(7) = s_{j-2}^0 \\ \Lambda_j(1) = \Lambda_j(7) = \Lambda_{j-1}(5) = s_{j-1}^0 \end{cases}$$

9: Calculate the worst response time R_j of operation τ_j thanks to expression (27).

10: Increment j: $j \leftarrow j+1$ and determine the start time s_{j+1}^0 of the first instance of operation $\tau_{j+1} = (C_{j+1}, T_{j+1})$ according to whether operation $\tau_j = (C_j, T_j)$ has a critical exact WCET C_j^*, or not.

$$s_{j+1}^0 = R_j + s_j^0 + \Lambda_j(C_j) \tag{38}$$

11: Go back to step 2 as long as there remain potentially schedulable operations.

12: Give the necessary and sufficient schedulability condition:

$$U_n^* \le 1 \quad \text{i.e.,} \quad U_n + \sum_{i=2}^{n} \frac{N_p(\tau_i) \cdot \alpha}{T_i} \le 1 \tag{39}$$

and the valid schedule S for the system taking into account the global cost due to preemptions:

$$S = \{(s_1^0, s_2^0, \cdots, s_n^0)\} \tag{40}$$

Example 3

Let $\alpha = 1$ and $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ be a system with four operations in V_r with the characteristics defined in table 4.

Table 4. Characteristics of example 4.3

	C_i	T_i
τ_1	2	5
τ_2	1	10
τ_3	3	20
τ_4	3	40

That system is potentially schedulable, indeed:

$$U_4 = \frac{2}{5} + \frac{1}{10} + \frac{3}{20} + \frac{3}{40} = 0.725$$

The scheduling algorithm that we introduced previously gives: $C_1^* = 2, C_2^* = 1, C_3^* = 4, C_4^* = 5$, thus:

$$U_4^* = \frac{2}{5} + \frac{1}{10} + \frac{4}{20} + \frac{5}{40} = 0.825$$

and we obtain (see figure 7):

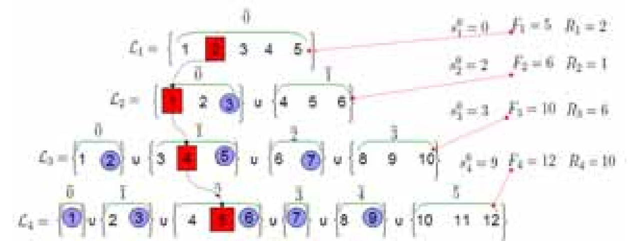


Figure 7. Scheduling algorithm

In figure 7, for each operation, we can see its actual exact WCET (squared), its critical exact WCET (circled), and its exact number of preemptions.

The global cost due to preemption is given by:

$$pr = \frac{1}{10}\cdot 0 + \frac{1}{20}\cdot 1 + \frac{1}{40}\cdot 2 = 0.1$$

and therefore the schedulability condition is:

$$U_4^* = U_4 + pr = 0.825 \leq 1$$

The valid schedule of the system of operations obtained with our scheduling algorithm is given in figure 8, and is such that:

$$\begin{aligned} S &= \{(s_1^0, s_2^0, s_3^0, s_4^0) = (0, R_1, R_2 + s_2^0, R_3 + s_3^0)\} \\ &= \{(0,2,3,9)\} \end{aligned}$$

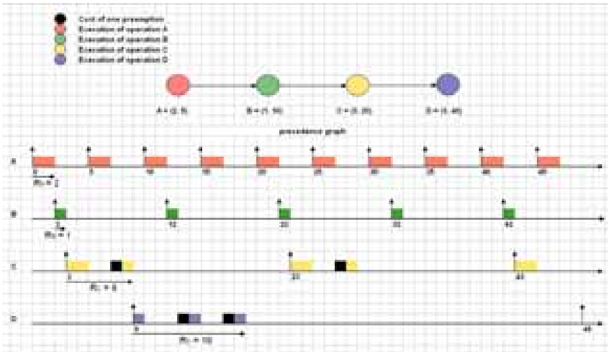


Figure 8. Preemptions taken into account

5 Conclusion and future work

We are interested in hard real-time systems with precedence and strict periodicity constraints where it is mandatory to satisfy these constraints. We are also interested in preemption which offers great advantages when seeking schedules. Since classical approaches are based on an approximation of the cost of the preemption in WCETs, possibly leading to a wrong real-time execution, we proposed a constructive approach so that its cost may be taken into account accurately. We proposed a scheduling algorithm which counts the exact number of preemptions for a system in V_r which is the subset of systems where the periods of all operations constitute an harmonic sequence as presented in section 3.2, and thus gives a stronger schedulability condition than Liu & Layland's condition.

Currently, we are seeking a schedulability condition for systems in V_i which is the subset of systems with irregular operations and we are planning to study the complexity of our approach in both V_r and V_i. Moreover, because idle time may increase the possible schedules we also plan to allow idle time, even though this would increase the complexity of the scheduling algorithm.

References

[1] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 1980.

[2] Ray Obenza and Geoff. Mendal. Guaranteeing real time performance using rma. *The Embedded Systems Conference, San Jose, CA*, 1998.

[3] J.P. Lehoczky, L. Sha, and Y Ding. The rate monotonic sheduling algorithm: exact characterization and average case bahavior. *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[4] N.C. Audsley, Burns A., M.F. Richardson, and A.J. Wellings. Hard real-time scheduling : The deadline-monotonic approach. *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[5] H. Chetto and M. Chetto. Some results on the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 1989.

[6] Patchrawat Uthaisombut. The optimal online algorithms for minimizing maximum lateness. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT)*, 2003.

[7] M. Joseph and P. Pandya. Finding response times in real-time system. *BCS Computer Journal*, 1986.

[8] Burns A. Tindell K. and Wellings A. An extendible approach for analysing fixed priority hard real-time tasks. *J. Real-Time Systems*, 1994.

[9] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[10] Tei-Wei Kuo and Aloysius K. Mok. Load adjustment in adaptive real-time systems. *Proceedings of the 12th IEEE Real-Time Systems Symposium*, 1991.

[11] Tindell K. Burns A. and Wellings A. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Software Eng.*, 1995.

[12] I. Ripol J. Echage and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. *In Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, 1995.

[13] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of 10th Real-Time Systems Conference, RTS'02*, Paris, France, March 2002.

[14] J.H.M. Korst, E.H.L. Aarts, and J.K. Lenstra. Scheduling periodic tasks. *INFORMS Journal on Computing 8*, 1996.

[15] L. Cucu and Y. Sorel. Schedulability condition for systems with precedence and periodicity constraints without preemption. In *Proceedings of 11th Real-Time Systems Conference, RTS'03*, Paris, March 2003.

[16] P. Meumeu and Y. Sorel. Non-schedulability conditions for off-line scheduling of real-time systems subject to precedence and strict periodicity constraints. In *Proceedings of 11th IEEE International Conference on Emerging technologies and Factory Automation, ETFA'06, WIP*, Prague, Czech Republic, September 2006.

[17] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *16th Euromicro Conference on Real-time Systems (ECRTS 04)*, Catania, Sicily, Italy, July 2004.

Algorithm and complexity for the global scheduling of sporadic tasks on multiprocessors with work-limited parallelism

Sébastien Collette* Liliana Cucu† Joël Goossens
Université Libre de Bruxelles, C.P. 212
50 Avenue Franklin D. Roosevelt
1050 Brussels, Belgium
E-mail: {sebastien.collette, liliana.cucu, joel.goossens}@ulb.ac.be

Abstract

We investigate the global scheduling of sporadic, implicit deadline, real-time task systems on identical multiprocessor platforms. We provide a task model which integrates work-limited job parallelism. For work-limited parallelism, we prove that the time-complexity of deciding if a task set is feasible is linear relatively to the number of (sporadic) tasks for a fixed number of processors. Based on this proof, we propose an optimal scheduling algorithm. Moreover, we provide an exact *feasibility utilization bound.*

1 Introduction

The use of computers to control safety-critical real-time functions has increased rapidly over the past few years. As a consequence, real-time systems — computer systems where the correctness of each computation depends on both the logical results of the computation and the time at which these results are produced — have become the focus of much study. Since the concept of "time" is of such importance in real-time application systems, and since these systems typically involve the sharing of one or more resources among various contending processes, the concept of scheduling is integral to real-time system design and analysis. Scheduling theory as it pertains to a finite set of requests for resources is a well-researched topic. However, requests in real-time environment are often of a recurring nature. Such systems are typically modeled as finite collections of simple, highly repetitive tasks, each of which generates *jobs* in a very predictable manner. These jobs have bounds upon their worst-case execution requirements and their periods, and associated deadlines. In this work, we consider *sporadic* task systems, i.e., where there are at least T_i time units between two consecutive instants when a sporadic task τ_i generates jobs and the jobs must be executed for at most C_i time units and completed by their relative deadline D_i. A particular case of sporadic tasks are the *periodic* tasks for which the period is the *exact* temporal separation between the arrival of two successive jobs generated by the task. We shall distinguish between *implicit deadline* systems where $D_i = T_i, \forall i$; *constrained deadline* systems where $D_i \leq T_i, \forall i$; and *arbitrary deadline* systems where there is no constraint between the deadline and the period.

The *scheduling algorithm* determines which job[s] should be executed at each time instant. We distinguish between *off-line* and *on-line* schedulers. On-line schedulers construct the schedule during the execution of the system; while off-line schedulers mimic during the execution of the system a precomputed schedule (off-line). Remark that if a task is not active at a given time instant and the off-line schedule planned to execute that task on a processor, the latter is simply idled (or used for a non-critical task).

When there is at least one schedule satisfying all constraints of the system, the system is said to be *feasible*. More formal definitions of these notions are given in Section 2.

Uniprocessor sporadic (and periodic) real-time systems are well studied since the seminal paper of Liu and Layland [9] which introduces a model of implicit deadline systems. For uniprocessor systems we know that the worst-case arrival pattern for sporadic tasks corresponds to the one of (synchronous and) periodic tasks (see, e.g. [11]). Consequently, the results obtained for periodic tasks apply to sporadic ones as well. Unfortunately, this is not the case upon multiprocessors due to scheduling anomalies (see, e.g. [1]).

The literature considering scheduling algorithms and feasibility tests for uniprocessor scheduling is tremendous. In contrast for *multiprocessor* parallel machines the problem of meeting timing constraints is a relatively new research area.

Related research. Even if the multiprocessor scheduling of sporadic task systems is a new research field, important results have already been obtained. See, e.g., [2] for a good presentation of these results. All these works con-

*Aspirant du F.N.R.S.
†Post-doctorante du F.N.R.S.

sider models of tasks where job parallelism is forbidden (i.e., job correspond to a *sequential* code). This restriction is natural for the uniprocessor scheduling since only one processor is available at any time instant even if we deal with parallel algorithms. Nowadays, the use of parallel computing is growing (see, e.g., [8]); moreover, parallel programs can be easily designed using the Message Passing Interface (MPI [5, 6]) or the Parallel Virtual Machine (PVM [12, 4]) paradigms. Even better, sequential programs can be parallelized using tools like OpenMP (see [3] for details). Therefore for the multiprocessor case we should be able to describe jobs that may be executed on different processors at the same time instant. For instance, we find such requirements in real-time applications such as robot arm dynamics [13], where the computation of dynamics and the solution of a linear systems are both parallelizable and contain real-time constraints.

When a job may be executed on different processors at the very same instant we say that the *job parallelism* is allowed. For a task τ_i and m identical processors we define a m-tuple of real numbers $\Gamma_i \stackrel{\text{def}}{=} (\gamma_{i,1}, \ldots, \gamma_{i,m})$ with the interpretation that a job of τ_i that executes for t time units on j processors completes $\gamma_{i,j} \times t$ units of execution. Full parallelism, which corresponds to the case where $\Gamma_i = (1, 2, \ldots, m)$ is not realistic; moreover, if full parallelism is allowed the multiprocessor scheduling problem is equivalent to the *uni*processor one (by considering, e.g., a processor m times faster).

In this work, we consider *work-limited* job parallelism with the following definition:

Definition 1 (work-limited parallelism) *The job parallelism is said to be work-limited if and only if for all Γ_i we have:*

$$\forall 1 \leq i \leq n, \forall 1 \leq j < j' \leq m, \; \frac{j'}{j} > \frac{\gamma_{i,j'}}{\gamma_{i,j}}.$$

For example, the m-tuple $\Gamma_i = (1.0, 1.1, 1.2, 1.3, 4.9)$ is not a work-limited job parallelism, since $\gamma_{i,5} = 4.9 > 1.3 \times \frac{5}{4} = 1.625$.

Remark that work-limited parallelism requires that for each task (say τ_i), the quantities $\gamma_{i,j}$ are distinct ($\gamma_{i,1} < \gamma_{i,2} < \gamma_{i,3} < \cdots$).

The work-limited parallelism restriction may at first seem strong, but it is in fact intuitive: we require that parallelism cannot be achieved for free, and that even if adding one processor decreases the time to finish a parallel job, a parallel job on j' processors will never run j'/j times as fast as on j processors. Many applications fit in this model, as the increase of parallelism often requires more time to synchronize and to exchange data between the parallel processes.

Few models and results in the literature concern real-time systems taking into account job parallelism. Manimaran et al. in [10] consider the *non-preemptive* EDF scheduling of *periodic* tasks, moreover they consider that the degree of parallelism of each task is *static*. Meanwhile, their task model and parallelism restriction (i.e., the sub-linear speedup) is quite similar to our model and our parallelism restriction (work-limited). Han et al. in [7] considered the scheduling of a (finite) set of real-time jobs allowing job parallelism. Their scheduling problem is quite different than our, moreover they do not provide a real model to take into account the parallelism. This manuscript concerns the scheduling of preemptive real-time sporadic tasks upon multiprocessors which take into account the job parallelism. From the best of our knowledge there is no such result and this manuscript provides a model, a first feasibility test and a first exact utilization bound for such kind of systems. **This research.** In this paper we deal with *global scheduling*[1] of implicit deadline sporadic task systems with work-limited job parallelism upon *identical parallel machines*, i.e., where all the processors are identical in the sense that they have the same computing power. We formally define our model, and consider the feasibility problem of these systems, taking into account work-limited job parallelism. For work-limited job parallelism we prove that the feasibility problem is linear relatively to the number of tasks for a fixed number of processors. We provide a scheduling algorithm.

Organization. This paper is organized as follows. In Section 2, we introduce our model of computation. In Section 3, we present the main result for the feasibility problem of implicit deadline sporadic task systems with work-limited job parallelism upon identical parallel machines when global scheduling is used. We prove that the feasibility problem is linear relatively to the number of tasks when the number of processors is fixed. In Section 4, we provide a linear scheduling algorithm which is proved optimal. We conclude and we give some hints for future work in Section 5.

2 Definitions and assumptions

We consider the scheduling of sporadic task systems on m identical processors $\{p_1, p_2, \ldots, p_m\}$. A task system τ is composed by n sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$, each task is characterized by a period (and implicit deadline) T_i, a worst-case execution time C_i and a m-tuple $\Gamma_i = (\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,m})$ to describe the job parallelism. We assume that $\gamma_{i,0} \stackrel{\text{def}}{=} 0$ $(\forall i)$ in the following. A job of a task can be scheduled at the very same instant on different processors. In order to define the degree of parallelization of each task τ_i we define the execution ratios $\gamma_{i,j}, \forall j \in \{1, 2, \ldots, m\}$ associated to each task-index of processor pair. A job that executes for t time units on j processors completes $\gamma_{i,j} \times t$ units of execution. In this paper we consider work-limited job parallelism as given by Definition 1.

We will use the notation $\tau_i \stackrel{\text{def}}{=} (C_i, T_i, \Gamma_i), \forall i$ with $\Gamma_i = (\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,m})$ with $\gamma_{i,1} < \gamma_{i,2} < \cdots < \gamma_{i,m}$.

[1] Job migration and preemption are allowed.

Such a sporadic task generates an infinite sequence of jobs with T_i being a lower bound on the separation between two consecutive arrivals, having a worst-case execution requirement of C_i units, and an implicit relative hard deadline T_i. We denote the utilization of τ_i by $u_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$. In our model, the period and the worst-case execution time are integers.

A task system τ is said to be *feasible* upon a multiprocessor platform if under all possible scenarios of arrivals there exists at least one schedule in which all tasks meet their deadlines.

Minimal required number of processors. Notice that a task τ_i requires more than k processors simultaneously if $u_i > \gamma_{i,k}$; we denote by k_i the largest such k (meaning that k_i is the smallest number such that the task system $\{\tau_i\}$ is feasible on $k_i + 1$ processors):

$$k_i \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } u_i \le \gamma_{i,1} \\ \max_{k=1}^{m}\{k \mid \gamma_{i,k} < u_i\}, & \text{otherwise.} \end{cases}$$

Notice that if $k_i = m$ for any i, the task system is infeasible as at least one task requires $m + 1$ processors.

For example, let us consider the task system $\tau = \{\tau_1, \tau_2\}$ to be scheduled on three processors. We have $\tau_1 = (6, 4, \Gamma_1)$ with $\Gamma_1 = (1.0, 1.5, 2.0)$ and $\tau_2 = (3, 4, \Gamma_2)$ with $\Gamma_2 = (1.0, 1.2, 1.3)$. Notice that the system is infeasible if the job parallelism is not allowed since τ_1 will never meet its deadline unless it is scheduled on at least two processors. There is a feasible schedule if the task τ_1 is scheduled on two processors and τ_2 on a third one.

Definition 2 (schedule σ) *For any task system $\tau = \{\tau_1, \dots, \tau_n\}$ and any set of m processors $\{p_1, \dots, p_m\}$ we define the* schedule *$\sigma(t)$ of system τ at instant t as $\sigma : \mathbb{R}_+ \to \{0, 1, \dots, n\}^m$ where $\sigma(t) \stackrel{\text{def}}{=} (\sigma_1(t), \sigma_2(t), \dots, \sigma_m(t))$ with*

$$\sigma_j(t) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if there is no task scheduled on } p_j \\ & \text{at instant } t; \\ i, & \text{if } \tau_i \text{ is scheduled on } p_j \text{ at instant } t \end{cases}$$

for all $1 \le j \le m$.

Definition 3 (canonical schedule) *For any task system $\tau = \{\tau_1, \dots, \tau_n\}$ and any set of m processors $\{p_1, \dots, p_m\}$, a schedule σ is* canonical *if and only if the following equations are satisfied:*

$$\forall j \in [1, m], \forall t, t' \in [0, 1), t < t' : \sigma_j(t') \le \sigma_j(t)$$

$$\forall j, j' \in [1, m], j < j', \forall t, t' \in [0, 1) : \sigma_j(t) \le \sigma_{j'}(t')$$

and the schedule σ contains a pattern that is repeated every unit of time, i.e.,

$$\forall t \in \mathbb{R}_+, \forall 1 \le j \le m : \sigma_j(t) = \sigma_j(t + 1).$$

Without loss of generality for the feasibility problem, we consider a feasibility interval of length 1. Notice that the following results can be generalized to consider any interval of length ℓ, as long as ℓ divides entirely the period of every task.

3 Our feasibility problem

In this section we prove that if a task system τ is feasible, then there exists a canonical schedule in which all tasks meet their deadlines. We give an algorithm which, given any task system, constructs a canonical schedule or answers that no schedule exists. The algorithm runs in $\mathcal{O}(n)$ time with n the number of tasks in the system.

We start with a generic necessary condition for schedulability using work-limited parallelism:

Theorem 1 *In the work-limited parallelism model and using an off-line scheduling algorithm, a necessary condition for a sporadic task system τ to be feasible on m processors is given by:*

$$\sum_{i=1}^{n} \left(k_i + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \right) \le m$$

Proof. As τ is feasible on m processors, there exists a schedule σ meeting every deadline. We consider any time interval $[t, t + P)$ with $P \stackrel{\text{def}}{=} \text{lcm}\{T_1, T_2, \dots, T_n\}$.

Let $a_{i,j}$ denote the duration where jobs of a task τ_i are assigned to j processors on the interval $[t, t+P)$ using the schedule σ. $\sum_{j=1}^{m} j \cdot a_{i,j}$ gives the total processor use of the task τ_i on the interval (total number of time units for which a processor has been assigned to τ_i). As we can use at most m processors concurrently, we know that

$$\sum_{i=1}^{n} \sum_{j=1}^{m} j \cdot a_{i,j} \le m \cdot P$$

otherwise the jobs are assigned to more than m processors on the interval. If on some interval of length ℓ, τ_i is assigned to j processors, we can achieve the same quantity of work on $j' > j$ processors on an interval of length $\ell \frac{\gamma_{i,j}}{\gamma_{i,j'}}$. In the first case, the processor use of the task i is $\ell\, j$, while in the second case it is $\ell j' \frac{\gamma_{i,j}}{\gamma_{i,j'}}$. By the restriction that we enforced on the tuple Γ_i (see Definition 1), we have

$$\begin{aligned} \ell j' \frac{\gamma_{i,j}}{\gamma_{i,j'}} &> \ell j' \frac{\gamma_{i,j'} \frac{j}{j'}}{\gamma_{i,j'}} \\ &> \ell j \end{aligned}$$

Let σ' be a slightly modified schedule compared to σ, where $\forall i \neq i', \forall j, a'_{i,j} = a_{i,j}$. For the task $\tau_{i'}$, it is scheduled on j' processors instead of $j < j'$ in σ for some interval of length ℓ, i.e.

$$a'_{i',j} = a_{i',j} - \ell$$

$$a'_{i',j'} = a_{i',j'} + \ell \frac{\gamma_{i,j}}{\gamma_{i,j'}}$$

Then, for that task $\tau_{i'}$,

$$\sum_{j=1}^{m} j \cdot a'_{i',j} > \sum_{j=1}^{m} j \cdot a_{i',j}$$

This proves that increasing the parallelism yields an increased sum; as we want to derive a necessary condition, we schedule the task on the minimal number of processors required. A lower bound on the sum is then given by

$$k_i \cdot P + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \cdot P$$

which corresponds to scheduling the task on $k_i + 1$ processor for a minimal amount of time, and on k_i processors for the rest of the interval. Then

$$\sum_{j=1}^{m} j \cdot a_{i,j} \geq k_i \cdot P + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \cdot P$$

and thus

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{m} j \cdot a_{i,j} &\leq m \cdot P \\ \sum_{i=1}^{n} \left(k_i \cdot P + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \cdot P \right) &\leq m \cdot P \\ \sum_{i=1}^{n} \left(k_i + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \right) &\leq m \end{aligned}$$

which is the claim of our theorem. ■

Theorem 2 *Given any feasible task system τ, there exists a canonical schedule σ in which all tasks meet their deadlines.*

Proof. The proof consists of three parts: we first give an algorithm which constructs a schedule σ for τ, then we prove that σ is canonical, and we finish by showing that the tasks meet their deadline if τ is feasible.

The algorithm works as follows: we consider sequentially every task τ_i, with $i = n, n-1, \ldots, 1$ and define the schedule for these tasks in the time interval $[0, 1)$, which is then repeated.

We calculate the duration (time interval) for which a task τ_i uses $k_i + 1$ processors. If we denote by ℓ_i the duration that the task τ_i spends on $k_i + 1$ processors, then we obtain the following equation:

$$\ell_i \, \gamma_{i,k_i+1} + (1 - \ell_i) \, \gamma_{i,k_i} = u_i.$$

Therefore we assign a task τ_i to $k_i + 1$ processors for a duration of

$$\frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}}$$

and to k_i processors for the remainder of the interval, which ensures that the task satisfies its deadline, since each job generated by the sporadic task τ_i which arrives at time t receives in the time interval $[t, t + T_i)$ exactly $T_i \times u_i = C_i$ time units.

The task τ_n is assigned to the processors $(p_m, \ldots, p_{m-k_n})$ (see Figure 1). If $u_n \neq \gamma_{n,k_n+1}$, another task can be scheduled at the end of the interval on the processor p_{m-k_n}, as τ_n does not require $k_n + 1$ processors on the whole interval.

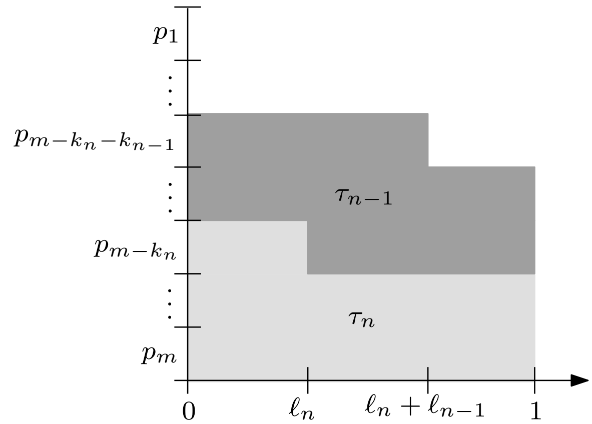


Figure 1. Schedule obtained after scheduling the task τ_n

We continue to assign greedily every task τ_i, by first considering the processors with highest number.

The schedule produced by the above algorithm is canonical as it respects the three constraints of the definition:

- on every processor j we assign tasks by decreasing index, thus $\sigma_j(t)$ is monotone and decreasing;
- for all $i < i'$, if $\tau_{i'}$ is scheduled on a processor $p_{j'}$, then τ_i is assigned to a processor p_j with $j \leq j'$;
- the schedule is repeated every unit of time.

The last step is to prove that if our algorithm fails to construct a schedule, i.e., if at some point we run out of processors while there are still tasks to assign, then the system is infeasible.

In the case of a canonical schedule, λ_i corresponds to:

$$\lambda_i = k_i + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}}.$$

So for instance, if a task τ_i is assigned to $\lambda_i = 2.75$ processors, it means that it is scheduled on two processors

for 0.25 time unit in any time interval of length 1, and on three processors for 0.75 time unit in the same interval.

If our algorithm fails, it means that $\sum_{i=1}^{n} \lambda_i > m$, which by Theorem 1 implies that the system is infeasible. ■

Corollary 3 *In the work-limited parallelism model and using an off-line scheduling algorithm, a necessary* ***and*** *sufficient condition for a sporadic task system τ to be feasible on m processors is given by:*

$$\sum_{i=1}^{n} \left(k_i + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \right) \leq m$$

Please notice that Corollary 3 can be seen as *feasibility utilization bound* and in particular a *generalization* of the bound for uniprocessor (see [9]) where a sporadic and implicit deadline task system is feasible if and only if $\sum_{i=1}^{n} u_i \leq 1$. Like the EDF optimality for sporadic implicit deadline tasks is based on the fact that $\sum_{i=1}^{n} u_i \leq 1$ is a sufficient condition, we prove the optimality of the canonical schedule based on the fact that $\sum_{i=1}^{n} \left(k_i + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}} \right) \leq m$ is a sufficient condition.

Corollary 4 *There exists an algorithm which, given any task system, constructs a canonical schedule or answers that no schedule exists in $\mathcal{O}(n)$ time.*

Proof. We know that the algorithm exists as it was used in the proof of Theorem 2. For every task, we have to compute the number of processors required (in $\mathcal{O}(1)$ time, as the number of processors m is fixed), and for every corresponding processor j, define $\sigma_j(t)$ appropriately. In total, $\mathcal{O}(n)$ time is required. ■

4 Scheduling algorithm

In this section we give a detailed description of the scheduling algorithm provided in the proof of Theorem 2. Based on the results of Section 3 this algorithm is optimal and runs in $\mathcal{O}(n)$ time (see Corollary 4).

For example for the task system $\tau = \{\tau_1, \tau_2\}$ given before we have $k_1 = 1$ and $k_2 = 0$. By using Algorithm 1 we obtain:

$$\begin{aligned}
\sigma_3(t) &= 2, \forall t \in [0, 0.75) \\
\sigma_3(t) &= 1, \forall t \in [0.75, 1) \\
\sigma_2(t) &= 1, \forall t \in [0, 1) \\
\sigma_1(t) &= 1, \forall t \in [0, 0.75) \\
\sigma_1(t) &= 0, \forall t \in [0.75, 1).
\end{aligned}$$

Notice that in Algorithm 1, we do not consider the optimization relatively to the number of preemptions or migrations and the scheduling algorithm does not provide satisfactory schedules for problems for which this is a issue. Nevertheless, the algorithm can decide the feasibility of every task system.

Algorithm 1 Scheduling algorithm of implicit deadline sporadic task system τ of n tasks on m processors with work-limited job parallelism

Require: The task system τ and the number of processors m
Ensure: A canonical schedule of τ or a certificate that the system is infeasible

1: let $j = m$
2: let $t_0 = 0$
3: let $\sigma_p(t) = 0, \forall t \in [0, 1), \forall 1 \leq p \leq m$
4: **for** $i = n$ downto 1 **do**
5:     **if** $u_i \leq \gamma_{i,1}$ **then**
6:         let $k_i = 0$
7:     **else**
8:         let $k_i = \max_{k=1}^{m}\{k \mid \gamma_{i,k} < u_i\}$
9:     **end if**
10:     **for** $r = 1$ upto k_i **do**
11:         let $\sigma_j(t) = i, \forall t \in [t_0, 1)$
12:         let $\sigma_{j-1}(t) = i, \forall t \in [0, t_0)$
13:         let $j = j - 1$
14:     **end for**
15:     let $tmp = t_0 + \frac{u_i - \gamma_{i,k_i}}{\gamma_{i,k_i+1} - \gamma_{i,k_i}}$
16:     **if** $tmp > 1$ **then**
17:         let $\sigma_j(t) = i, \forall t \in [t_0, 1)$
18:         let $j = j - 1$
19:         let $t_0 = 0$
20:         let $tmp = tmp - 1$
21:     **end if**
22:     let $\sigma_j(t) = i, \forall t \in [t_0, tmp)$
23:     let $t_0 = tmp$
24:     **if** $j \leq 0$ **then**
25:         return Infeasible
26:     **end if**
27: **end for**

5 Discussions

Job parallelism vs. task parallelism. In this manuscript we study multiprocessor systems where job parallelism is allowed. We would like to distinguish between two kinds of parallelism, but first the definitions: *task parallelism* allows each task to be executed on several processors at the same time, while *job parallelism* allows each job to be executed on several processors at the same time. If we consider constrained (or implicit) deadline systems task parallelism is not possible. For *arbitrary* deadline systems, where several jobs of the same task can be active at the same time, the distinction makes sense. Task parallelism allows the various active jobs of the same task to be executed on a different (but unique) processor while job parallelism allows each job to be executed on several processors at the same time.

Optimality and future work. In this paper we study the feasibility problem of implicit deadline sporadic task systems with work-limited job parallelism upon identical parallel machines when global scheduling is used. We prove that our problem has a time-complexity that is linear relative to the number of tasks. We provide an optimal scheduling algorithm that runs in $\mathcal{O}(n)$ time and we give an exact feasibility utilization bound.

Our algorithm is optimal in terms of the number of processors used. It is left open whether there exists an optimal algorithm in terms of the number of preemptions and migrations. As a first step, we used an interval of length 1 to study the feasibility problem. If we generalize our algorithm to work on an interval of length equal to the $\gcd$ of the periods of every task, we decrease the preemptions and migrations. We do not know, however, if the result is optimal.

The definitions of work-limited job parallelism was given here for identical processors, one should investigate an extension of this definition to the uniform processor case.

Acknowledgments

The authors would like to thank Sanjoy Baruah for posing the feasibility problem and Jean Cardinal for taking part in interesting discussions. Finally the detailed comments of an anonymous referee greatly helped in improving the presentation of the manuscript.

References

[1] B. Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2003.

[2] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook of Scheduling*, 2005.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[5] S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.

[6] W. Gropp, editor. *Using MPI : portable parallel programming with the message-passing interface*. Cambridge, MIT Press, 2nd edition, 1999.

[7] C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS'89)*, pages 59–67, 1989.

[8] E. Leiss. Parallel and vector computing. *McGraw-Hill, Inc.*, 1995.

[9] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[10] G. Manimaran, C. Siva Ram Murthy, and K. Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15:39–60, 1998.

[11] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[12] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[13] A. Y. Zomaya. Parallel processing for real-time simulation: A case study. *IEEE Parallel and Distributed Technology:System and Technology*, 4(2):49–55, 1996.

Scheduling 2

Schedulability analysis of OSEK/VDX applications

Pierre-Emmanuel Hladik
LINA (FRE CNRS n°2729)
École des Mines de Nantes
pierre-emmanuel.hladik@emn.fr

Anne-Marie Déplanche, Sébastien Faucou and Yvon Trinquet
IRCCyN (UMR CNRS n°6597)
Université de Nantes
firstname.name@irccyn.ec-nantes.fr

Abstract

This paper deals with applying state-of-the art schedulability analysis techniques to OSEK/VDX-based applications for real-time embedded systems. To do so, we explore two complementary problems: (i) extending state-of-the-art results in schedulability analysis in order to take into account OSEK/VDX specific constructs; (ii) defining design rules that must be followed in order to build applications that comply with the hypothesis made during the analysis. The main work reported here deals with a simple periodic task model. An extension to software architectures with precedence constraints, which will be discussed in a future paper, is also briefly introduced.

1. Introduction

This paper deals with the design and schedulability analysis of monoprocessor real-time embedded applications executed on top of an OSEK/VDX-compliant real-time kernel. An OSEK/VDX application consists of a set of concurrent tasks, which compete for the processor. An OSEK/VDX application can be as simple as a set of independent periodic tasks, but it can be much more complicated. Nevertheless, as long as we deal with real-time systems, one of the key requirements to meet at the design step is the verification of the timing constraints. In order to check this, the designer usually performs a schedulability analysis of the task set, based on a technique that makes hypothesis on the application task model. Thus, in order to have the possibility to use schedulability analysis techniques, the designer must master the complexity of the application that he creates, according to the used real-time kernel services, in order to comply with the analysis hypothesis. This is rather difficult because of the vast number of services and the vast number of possible application models.

In this context, our main objective is to provide:

- schedulability analysis techniques that take into account the specificities of the OSEK/VDX real-time kernel;
- design rules that must be followed in order to comply with the hypothesis made by the proposed schedulability analysis techniques.

Notice that our goal is not to develop new schedulability analysis techniques. Hence, to fulfill the first item, we worked on adapting state-of-the art results to OSEK/VDX applications. The second item can be thought of as an attempt to define design guidelines for OSEK/VDX-based applications. Another objective, not presented here, is to build a tool that will complete our OSEK/VDX open source implementation [1, 7].

The paper is organised as follows. First of all, the basic features of OSEK/VDX are presented. Then, we introduce the independent periodic task model together with an analysis technique and some design guidelines. Next, we consider some possible extensions of this work, especially the case of software architectures including precedence constraints, before to conclude.

2. Summary of the OSEK/VDX RTOS specification

OSEK/VDX ("Offene Systeme und deren Schnittstellen fur die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive") [11] aims at being an industry standard for RTOS used in distributed control units in vehicles. The OSEK group aims at standardising the interfaces of the operating system to ease application software portability, interoperability, re-use and the supply of software packages independently of hardware units. Various parts are proposed for the standard: OS (the basic services of the real-time kernel), COM (the communication services), NM (the Network Management

services) and OIL[1] (OSEK Implementation Language). The presentation below concerns only the kernel of the operating system (OS 2.2.3, Feb. 2005). For more information refer to [11].

All the objects of an OSEK/VDX application are static: they are created before the start-up of the application and are never destroyed (no dynamic allocation).

Task management. OSEK/VDX specification provides two different task types. On one side, a basic task is a sequential code without system call being able to block the task. Synchronisation points are only at the beginning and the end of the task. On the other side, an extended task is composed of one or several blocks separated by invocations of OS services, which may result in a *waiting* state. According to the conformance classes of OSEK/VDX OS (see below), it is possible to authorise the record of activation requests of a task while it is already active. Every request is then recorded (no multiple instances) and taken into account when the task ends. The basic services offered for task management are: *ActivateTask* (activates the target task), *TerminateTask* (mandatory auto-termination of the current task) and *ChainTask* (atomic combination of *ActivateTask* and *TerminateTask*).

Conformance classes. The conformance classes define four versions of the real-time kernel, in order to adapt it to different requirements. They are determined by three main attributes and some implementation requirements: the type of task (basic or extended), the possibility of recording multiple activation requests for a basic task and finally the number of tasks per priority level. The four conformance classes are BCC1, BCC2, ECC1 and ECC2. They are summarized in table 1.

	Task type	Activ. queuing	Tasks / priority
BCC1	Basic	No	1
BCC2		Yes (Basic)	Many
ECC1	Basic + Extended	No	1
ECC2		Yes (Basic)	Many

Table 1. OSEK/VDX OS conformance classes summary

Scheduling policy. For scheduling, static priorities are assigned to tasks and the "Highest Priority First" policy is used, with FIFO as a second criterion for BCC2 and ECC2 applications where many tasks share the same priority level. For an application, the scheduling can be: full non preemptive, full preemptive or mixed preemptive. In this last case, every task has its appropriate mode (preemptive or non preemptive, specified in the OIL file). There exists also a notion of group, for tasks that share a common internal resource. An internal resource is automatically taken by a task of the group when it gets the CPU and released when it terminates, waits for an event or invokes the *Schedule* service. Usual preemption rules are used for the tasks which are not in the group, according to their priority level. Inside the group the tasks can't preempt among themselves.

Task synchronisation. Synchronisation (extended tasks only) is based on the private event mechanism: only the owner task can explicitly wait for the occurrence of one or more of its events (logical OR). The setting of occurrences can be made by tasks (basic or extended) or ISRs (Interrupt Service Routine). There is no timeout associated to the *WaitEvent* service, but using the alarm concept (see below) it is possible to build watchdogs to monitor the timing behaviour.

Resources management. OSEK/VDX coordinates the concurrent access to shared resources with the OSEK-PCP protocol (Priority Ceiling Protocol). OSEK-PCP protocol is more simple than original PCP [16, 9]. It is also known as "Immediate Priority Ceiling Protocol" (IPCP). When a task gets a resource, its priority is immediately raised to the resource priority, so that other tasks that share the same resource cannot get the CPU. The resource sharing is allowed between tasks and ISRs or between ISRs. For that purpose, a virtual priority is assigned to every interrupt. Two services allow to control access to resources: *GetResource* and *ReleaseResource*. A predefined system resource (*Res_Scheduler*) allows a task to lock the processor and execute some code in non preemptive scheduling mode.

Alarms and counters. These objects allow mainly the processing of recurring phenomena: timer ticks, signals from mechanical organs of a car engine (camshaft, crankshaft). They constitute complements to the event mechanism. They allow the management of periodic tasks and watchdog timers for the monitoring of various situations (wait for an event occurrence, send/receive a message). A *Counter* is an object intended for the counting of "ticks" from a source. An *Alarm* allows to link a *Counter* and a *Task*. An *Alarm* expires when the value of its associated counter reaches a predefined value. When an *Alarm* expires, an action is taken: either the activation of the associated task, the setting of an event of the task or the execution of a routine. An *Alarm* can be defined to be single-shot or cyclic, the corresponding *Counter* value being absolute or relative to the current *Counter* value.

Communication. The communication services of OSEK/VDX are built around the *Message* object. Two types of messages are offered: those using the blackboard model (*Unqueued*, single place buffer); those using a FIFO (*Queued*). The communication services are the same whatever the communication is local or distant. For more information refer to [11].

In this paper, we focus on the BCC1 and BCC2 conformance classes, thus ignoring the event management services. We also focus on monoprocessor systems, thus ignoring distributed communication services. Moreover, we investigate the case of software architectures where com-

[1] An OIL file describes at a low-level the software architecture of an OSEK/VDX application in order to automatically generate kernel configuration files. More details can be found in [10].

municating tasks follow the blackboard pattern, so that data flows can easily be implemented through shared variables[2] (and, when needed, resources). Hence, we ignore local communication services.

3. The periodic task model

In this section, we consider a subset of OSEK/VDX task configurations. It combines most of the specificities of the BCC1 and BCC2 classes, while giving rise to *accessible* schedulability analyses. We list its features: all tasks are periodic and non-concrete, *i.e.* the date of the first activation of a task is unknown; some tasks can have the same priority; the scheduler is mixed preemptive; there are no precedence between tasks, but some tasks can share resources with the IPCP protocol as well as internal resources through groups.

First , we propose an equivalent model for the schedulability analysis of such tasks. Then, we discuss about the implementation of such tasks so as to ensure consistency between the model and the OSEK/VDX application.

3.1. Notations and definitions

We want to point out that with the OSEK/VDX configurations we are looking at, the scheduling behaviour is not a "strictly fixed priority" one in the sense that the priority of a task may not remain fixed on-line. The priorities that are assigned to tasks when they are declared are used by the scheduler to select the next running task. However, because of shared resources and mixed preemptive scheduling, the priority of a task can increase during its execution. Such a behaviour can be captured with the notion of preemption threshold in a quite generic way. The preemption threshold was introduced in [18], and is the priority level that the task has during its execution, *i.e.* the priority considered by the scheduler for the task after its first start event, and before its terminate event. The notion of preemption threshold that we use hereafter is brought at the code section level in tasks.

3.1.1 Model

Let $\Gamma = \{\tau_i\}$, $1 \leq i \leq n$ be the set of the n tasks of the application under analysis. Each task τ_i is a tuple $< p_i, d_i, \pi_i, e_i >$ where: p_i is its period; d_i is its relative deadline (we allow either $d_i \leq p_i$, or $d_i > p_i$); π_i its user-declared priority level; and $e_i = \{< e_{ij}, \gamma_{ij} >\}$ is a set of execution times with their corresponding preemption threshold. A 2-uplet $< e_{ij}, \gamma_{ij} >$ describes a code section of task τ_i of which e_{ij} is the worst-case execution time (without preemption) and γ_{ij} the priority to be given to τ_i while executing this code (see Fig. 1).

Here, for each task, e_{i0} is its worst-case execution time and γ_{i0} is deduced from its characteristics: for a non-

[2]To our knowledge, all OSEK/VDX implementations use a flat memory model. Hence, there is no need for system services dedicated to shared variable handling

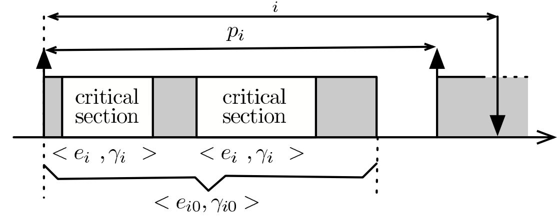


Figure 1. Task model

preemptive task, γ_{i0} is considered as infinite (or the highest priority level); for a task belonging to a task group, γ_{i0} is equal to the priority of the group, *i.e.* a priority higher than all priorities of the tasks in the group; and for other tasks $\gamma_{i0} = \pi_i$.

Each other 2-uplet $< e_{ij}, \gamma_{ij} >$, $1 \leq j$, corresponds to a critical section in τ_i, with e_{ij} the time needed to execute the code between the lock and unlock operations, and γ_{ij} its ceiling priority, *i.e.*, a priority higher than the highest priority of the tasks that share the same resource. If the resource is the processor, the ceiling priority is considered as infinite (or the highest priority level).

3.1.2 Definitions

The schedulability analysis presented in this paper is based on the computation of the worst-case response time of each task. If the worst-case response time of a task is smaller than its deadline, then the task is schedulable. If all the tasks in the system are schedulable, then the system is schedulable too. The response time computation is based on the well-known busy period analysis [2, 5, 8, 18]; it requires to determine the length of the busy period which starts at a critical instant.

A π-busy period is a time interval during which the processor is continuously processing at priority π or higher. Remark that in our model, a non preemptive task is always executed with a priority higher than π (cf. its γ_{i0}). An instance of τ_i is necessarily executed during a π_i-busy period. Furthermore, more than one instance of τ_i can be executed during the same π_i-busy period. For a π_i-busy period, we define $a_i(q)$ the date of the qth activation of τ_i in the busy period, $s_i(q)$ its start time, and $f_i(q)$ its finish time. By definition, the response time of the qth instance of τ_i in a π_i-busy period is:

$$r_i(q) \stackrel{\text{def}}{=} f_i(q) - a_i(q)$$

Then, the worst-case response time of τ_i is defined by:

$$R_i \stackrel{\text{def}}{=} \max_{\forall \pi_i\text{-busy-period}, \forall q \geq 1} \{r_i(q)\}$$

The critical instant for a task τ_i describes a π_i-busy period where τ_i meets its worst-case response time. Thus, to compute the worst-case response time of τ_i, we need to

know its critical instant and to compute each $f_i(q)$ in this π_i-busy period.

We point out that the computation of the worst-case response time developed in the next section is an exact one in the sense that the critical instant that is defined there describes a worst-case scenario for the task that may occur actually; and that the given expressions are not approximate ones.

3.2. Schedulability analysis

In [18], the authors consider the same model as us, but without shared resources and identical priorities. The critical instant is described as an instant (time 0) when τ_i is activated together with an instance of each higher priority task, and the task that contributes to the maximum blocking time has just started executing prior to time 0. They prove that in computing the blocking time for a task τ_i, one needs to consider blocking from only one lower priority task with a preemption threshold higher than or equal to π_i. Such a critical instant specification can easily be extended in order to take into account shared resources and identical priorities.

With regard to shared resources, even if it is slightly different in the implementation than the original priority ceiling protocol (PCP), IPCP exhibits the same worst-case performance (from a scheduling view point) [3]. Thus the property of PCP shown by Sha *et al.* in [16] is still true and the corresponding worst-case blocking time of τ_i is reduced to at most the duration of at most one critical section of a lower priority task that is using a resource whose ceiling priority is higher than or equal to π_i. For the unifying model we consider, it is easy to show that a task τ_i can be blocked by at most one critical section of a lower priority task with a preemption threshold higher than or equal to π_i.

As for tasks with identical priorities, in [2] the authors prove that the critical instant for tasks with the same priority occurs when all tasks are activated simultaneously.

Thus, by combining these various properties, the critical instant can be defined for the specific model studied in this section. It occurs when (1) τ_i is activated (time 0) together with (2) an instance of each higher or equal priority task, and (3) the longest code section with a higher preemption threshold but belonging to a lower priority task starts at time 0.

3.2.1 Computing Blocking Time

A task τ_i can be blocked by a task τ_j with a lower priority, if a code section of τ_j with a higher or equal preemption threshold than π_i exists. Thus, the longest blocking time of τ_i is [16]:

$$B_i = \max_{\forall (j,k), \pi_j < \pi_i \leq \gamma_{jk}} \{e_{jk}\}$$

3.2.2 Computing the qth start time

The qth instance of a task τ_i in a π_i-busy period can start its execution at the latest when: the blocking time is elapsed; and all the previous instances of τ_i in the busy period have completed; and all higher priority tasks activated before $s_i(q)$ have completed; and all the equal priority tasks activated before $a_i(q)$ have completed because of the FIFO scheduling for tasks with the same priority.

Thus, the start time of the qth instance of τ_i is obtained by computing the fix-point of [18, 2]:

$$\begin{aligned} s_i(q) &= B_i + (q-1)e_{i0} \\ &+ \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{s_i(q)}{p_j} \right\rfloor\right) e_{j0} \\ &+ \sum_{\forall j \neq i, \pi_j = \pi_i} \left(1 + \left\lfloor \frac{(q-1)p_i}{p_j} \right\rfloor\right) e_{j0} \end{aligned}$$

3.2.3 Computing the qth finish time

Between the start time of the qth instance of τ_i and its finish time, the scheduler can select for running only a task with a priority higher than γ_{i0}. Thus, the finish time is computed by solving the fix-point of [18, 2]:

$$\begin{aligned} f_i(q) &= s_i(q) + e_{i0} \\ &+ \sum_{\forall j, \pi_j > \gamma_{i0}} \left(\left\lceil \frac{f_i(q)}{p_j} \right\rceil - \left\lfloor \frac{s_i(q)}{p_j} \right\rfloor - 1 \right) e_{j0} \end{aligned}$$

3.2.4 Computing the worst-case response time

To determine the worst-case response time of τ_i it is necessary to check the finish time for each instance of τ_i in the busy period started at the critical instant. The length of this π_i-busy period is:

$$L_i = B_i + \sum_{\forall j, \pi_j \geq \pi_i} \left\lceil \frac{L_i}{p_j} \right\rceil e_{j0}$$

The number of instances of τ_i in the busy period is $N_i = \lceil L_i/p_i \rceil$, and so:

$$R_i = \max_{q \in [1, N_i]} \{f_i(q) - (q-1)p_i\}$$

3.3. Implementation

The schedulability analysis technique presented above extends state-of-the art results in order to integrate some specific constructs of the OSEK/VDX kernel. Nevertheless, it relies on a set of hypothesis that must be verified by the implementation. We expose in this section a set of rules that must be followed in order to achieve this goal.

The model of computation of the periodic task model requires that:

- the first activation of a task is free. Thus it can be triggered by an external event (R1) or at a specified date (R1');

- the following activations are periodic (R2), or sporadic or sporadically periodic (R2');
- the body of a task does not call a service that could cause a preemption except for resource unlocking (note that resource locking is not a preemption point in OSEK/VDX), neither does it contains suspension points (R3);
- the body of a task implements a deterministic algorithm, and a WCET can be computed (R4);

The formalization of the model in the previous section expresses explicitly requirements R1 and R1'. It also states that the tasks of the system are periodic ones (R2). However, as the analysis is based on a worst-case scenario, the analysis is also valid for sporadic or sporadically periodic tasks (these activation patterns produce the same worst-case scenario as the periodic pattern). Hence, we have requirement R2'. Lastly, requirements R3 and R4 are implicit: they translate the model of computation traditionally used in the real-time scheduling theory.

Let us consider requirements R1 and R2. In order to program recurring tasks, OSEK/VDX provides the *Alarm* object. As stated in section 2, an *Alarm* is used to link a *Counter* and a *Task*. When the *Counter* reaches a specific value, the *Alarm* expires and triggers an action on the target *Task*: activation, or event signaling (not possible for BCCx applications). Thus, when the triggering event of the first activation of τ_i is raised, the serving ISR must program the *Alarm* so that it expires every p_i. The ISR also performs the first activation of the task, subsequent activations being made by the kernel, according to the *Alarm* configuration given in the OIL application description file. As shown in fig. 2, the *SetRelAlarm* service must be used. Its parameters are (in this order): reference to the target *Alarm*, relative date of first expiration, and cycle time (both expressed in *Counter* ticks). The figure also shows the OIL code used to link the *Alarm*, the *Counter* (here `SYSTIMER`) and the *Task*.

```
// C code
ISR(TriggerFunc){
    ActivateTask(Ti);
    SetRelAlarm(AwakeTi,Pi,Pi);
}

// OIL code
ALARM AwakeTi {
    COUNTER=SYSTIMER;
    ACTION=ACTIVATETASK{
        TASK=Ti;
    };
    AUTOSTART=FALSE;
};
```

Figure 2. C and OIL codes to ensure requirements R1 and R2

If we consider requirement R1' instead of R1, we can use the `AUTOSTART` attribute of the *Alarm* object to achieve the desired behavior if the date of the first activation is not 0 (see figure 3). Indeed, according to the OSEK/VDX standard, setting parameter `Oi` (offset of task τ_i) to 0 would produce an implementation specific behavior. Hence, in this particular case, we must use the `AUTOSTART` attribute of the *Task* object for the first activation, and the `AUTOSTART` attribute of the *Alarm* object for subsequent ones, where the `ALARMTIME` attribute (first expiry date) is set to `Pi` (where `Pi` is p_i). The corresponding OIL code is given figure 4.

```
ALARM AwakeTi {
    COUNTER=SYSTIMER;
    ACTION=ACTIVATETASK{
        TASK=Ti;
    };
    AUTOSTART=TRUE{
        ALARMTIME=Oi;
        CYCLETIME=Pi;
        APPMODE=DefaultMode;
    };
};
```

Figure 3. OIL code to ensure requirements R1' and R2 when `Oi`$\neq 0$.

```
TASK Ti {
  AUTOSTART=TRUE{
    APPMODE=DefaultMode;
  };
  ...
};
ALARM AwakeTi {
    COUNTER=SYSTIMER;
    ACTION=ACTIVATETASK{
        TASK=Ti;
    };
    AUTOSTART=TRUE{
        ALARMTIME=Pi;
        CYCLETIME=Pi;
        APPMODE=DefaultMode;
    };
};
```

Figure 4. OIL code to ensure requirements R1' and R2 when `Oi`$= 0$.

As OSEK/VDX targets small real-time embedded systems, it does not enforce implementations to provide more than one *Alarm* object. Hence, one can face the situation where the number of *Alarm* objects is lower than the number of periodic tasks. Obviously, our previous approach cannot be used anymore. To solve this problem, one solution consists in using a periodic task, τ_{time}, which will in turn activates the other tasks of the system. Of course, τ_{time} must be included in the analysed task set. In order to achieve all the activations, its period must equal the greatest common divider of the other periodic tasks ($p_{time} = gcd\{p_i\}_{1 \leq i \leq n}$). For some configurations, this can lead to a high cpu utilization. However, this is a re-

current problem of tick-driven schedulers and it is out of the scope of this paper. So as to be able to preempt any running task and activate a task of potentially greater priority, τ_{time} must be granted the highest priority among the tasks: $\forall \tau_i \in \Gamma, \pi_i < \pi_{time}$. These conditions are not sufficient: some tasks can be non preemptive (and all tasks can become non preemptive for some time by taking the *Res_Scheduler* resource). A problem may arise when the duration of such a non preemptive section is greater or equal to p_{time}: one or more of τ_{time} activation requests might be lost. To avoid this situation, m_{time}, the number of activation requests memorized for τ_{time} must fulfill the following constraint: $m_{time} > \left\lceil \frac{R_{time}}{p_{time}} \right\rceil$. If $m_{time} > 1$, we must use the BCC2 conformance class. Let us underline that this selection of the number of memorized activation requests must also be considered when the worst case execution time of a task is greater than its period (and lower than its deadline). Indeed, schedulability analysis techniques always suppose that all activation requests are memorized.

Let us consider now requirement R2': after the first activation, the tasks are sporadic or sporadically periodic. A sporadic task is a real-time event-triggered task (for instance supervision of functional modes of the application, emergency handling, etc.). In order to allow analysis of sporadic behavior, a minimal inter-arrival time, called pseudo-period, must be given. In the worst case, a sporadic task has a periodic behaviour, its period being equals to its pseudo-period. A sporadically periodic task is a real-time event triggered functionality: once triggered, the task that implements the functionality is periodic. Depending on the functional mode, the functionality can be stopped. Hence, the task is not dispatched anymore until the functionality is one more time requested. In order to allow analysis of sporadically periodic behavior, a delay must be respected between a deactivation and the subsequent activation of the functionality. Hence, in the worst case, a sporadically periodic task has a periodic behaviour, its period being equals to the minimum between its period and its deactivation-activation delay. As worst-case response time computation techniques consider worst cases, sporadic tasks and sporadically periodic tasks can be considered as periodic tasks and analysis techniques do not need to be extended.

On the implementation side, OSEK/VDX does not offer any native support to ensure that the effective activation law of a sporadic task (resp. sporadically periodic) will not violate the pseudo-period (resp. deactivation-activation delay) hypothesis. Hence, while there exists no mechanism to solve this problem, it is not safe to use sporadic or sporadically periodic tasks in an OSEK/VDX application. In other words, requirement R2' cannot be fulfilled. However, it is out of the scope of this paper to discuss the implementation of such robustness mechanisms.

Let us consider now requirement R3. To express implementation constraints, we cut the body of tasks in a sequence of a *Computation* part and a *Finalization* part. In its *Computation* part, the task is not allowed to use the following services: *ActivateTask*, *TerminateTask* and *ChainTask*. Moreover, if it performs an I/O that has a non negligible response time (more than just reading/writing a value from/into a dedicated register), it enters a busy-wait, which is taken into account in its WCET. Of course, if the I/O response time is too costly, another solution should be used (for instance delegating the interaction with the device to an ISR, or considering to use an other device). For this model, the *Finalization* part consists in a call to the *TerminateTask* service.

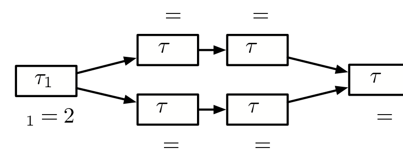
Lastly, requirement R4 is common to all real-time applications and has been largely studied by the scientific community. It is out of the scope of our current work. As an entry point, the interested reader can refer to [13].

4. Adding precedence relations

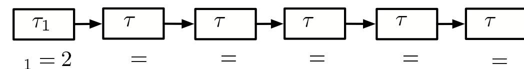
In this section, the software architecture (task model) of section 3 is supplemented with precedence relations modelled by precedence graphs between tasks. Of course, we consider here precedence relations without cycle. A root task of a graph is still strictly periodic and non-concrete, whereas a non-root task is activated upon the completion of its predecessor task(s), ie., a task can have multiple predecessors in the same graph, see Fig. 5(a).

4.1. Transformation of precedence graph into precedence chain

As we will justify in paragraph 4.3, in order to implement software architectures specified as a set of periodic precedence graphs, while meeting the hypothesis of analysis algorithms, we must transform each periodic precedence graph into a periodic precedence chain, i.e., in a chain, a task has at most one successor and at most one predecessor, see Fig. 5(b).



(a) Initial precedence graph



(b) Associated precedence chain

Figure 5. Transformation of a precedence graph into a precedence chain

In the case where tasks cannot share a same priority level, Richard proves in [15] that the transformation from

a set of graphs to a set of chains preserves the execution sequences. We simply adapt his transformation to the OSEK/VDX case. Mathematically, the chain has to be a linear extension of the partial order defined by the precedence graph. Moreover, it must exhibit the same behaviour with regards to task activation and termination instants so as to be a correct implementation of the specification. It means especially that when two tasks of a graph are unrelated, then the task with the higher priority must be inserted in the chain before the task with the lower priority (see Fig. 5). As these two criteria are not sufficient to define a total order (remember that tasks can share a same priority level), at least an other criterion must be considered, for instance the alphabetical order – as the specification is not deterministic, any implementation is correct – (see tasks τ_4 and τ_5 in Fig. 5).

As a consequence, in the following sections, we only consider precedence chains.

4.2. Schedulability Analysis

Due to space limitation, we cannot expand the schedulability analysis of our model. For a detailed presentation of the computation algorithms, we refer the reader to [6]. Contrary to the section 3, the computation for precedence chain gives an upper bound of the worst-case response time.

The computation of the worst-case response time for precedence chains is based on [5, 15]. However, because of some specificities of OSEK/VDX – e.g., mixed scheduling, task groups, same priority level – some major modifications have to be done.

4.3. Implementation

In order to implement software architectures described in the form of a set of periodic precedence graphs and to preserve the meanings of analysis results for the effective system, we have to explain the implicit hypothesis made by the analysis technique. It is supposed that a task (that has at least one predecessor) is activated as soon as all its predecessors are finished. To be more explicit, it also means that a task (that has at least one predecessor) is not activated before all its predecessor are finished. Hence, considering a task, the termination of its last predecessor and its activation must be combined in an atomic action. The only way to obtain such a behaviour in OSEK/VDX is to use the *ChainTask* service. The problem now is that *ChainTask* implements a 1-to-1 precedence relationship. So we have to transform the specification into an implementation, where the only form of precedence authorized is 1-to-1, that is, a precedence chain. We follow the technique described in section 4.1 to perform the transformations.

Our problem now is to implement precedence chains on top of OSEK/VDX. As we did for the periodic task model, we list the requirements that need to be met:

- the root task of a chain must follow requirements R1 (or R1'), R2, R3 and R4 defined in section 3.3;
- the activation of a non-root task of the chain must correspond to the termination of its predecessor (requirement R5);
- non-root tasks of the chain must follow requirements R3 and R4.

We have given in section 3.3 the implementation rules that must be followed in order to achieve requirements R1 to R4. In order to achieve requirement R5 we have only one rule to update: for every task of the chain that has a successor, its *Finalization* part consists in a call to the *ChainTask* service (instead of *TerminateTask*), where the target of the service is the successor of the task. Notice that, as non-root tasks do not need to follow requirements R1 (or R1') and R2, the activation mechanism described in section 3.3 must not be used for them. The only possibility for these tasks to be activated is the execution of the *Finalization* part of their predecessor in the chain.

5. Further extensions

In this section, we discuss possible extensions of the work presented above.

5.1. Offsets

With regard to the periodic task model of the section 3, the assumption made about the first activation dates can be modified and offsets can be given, *i.e.* the date of the first activation is a priori known for each task. For the same reasons as in [4], it can be proved that the computation conducted in section 3.2 still gives an upper bound of the worst-case response time. However in the case where the offsets are not all equal (tasks are said asynchronous), the work of Redell and Törngren [14] can be used to reduce the approximation. In this work, all tasks are assumed: periodic; with some offsets; independent; with different priority levels, or not; with deadlines that can be arbitrary large; and with shared resources managed with PCP. In order to compute the worst-case response time of a task τ_i, Redell and Törngren separate all the task instances that may interfere with τ_i into different sets. These sets are function of the activation date of these instances and the start date of τ_i (as for the computation presented in Section 3, where we consider interference before $s_i(q)$, and after). To extend their method to tasks with preemption threshold, we can simply adapt the interference caused by the instances that occur after the start time by considering only instances with a higher priority than the preemption threshold (Eq(12) in [14]).

5.2. Taking into account system overheads

In [2], Bimbard and George study the overheads of an OSEK/VDX OS implementation in the context of schedulability analysis. In a first time, they identify the source of kernel overheads that influence the response time of tasks; in a second time they show how to take into account these overheads in the computation of the worst-case response

time. They give four sources for system overheads : the execution time of the counter-management-ISR that occurs every tick (see Section 3.3); the computation time required to activate a task; the computation time required to schedule the tasks; the computation time to terminate a task and reschedule. In this paper, because of shared resources, we have to add the computation time needed to lock and unlock a resource (*GetResource* and *ReleaseResource* services).

In [2], the interference of each overhead is included into worst-case response time computation equations. Extending this method in order to take into account shared resources and preemption thresholds seems to be intractable and inefficient. The main reason is that the computation of the response time is not an exact computation when the mode becomes too complex (e.g. when using the model with precedence). Some approximations are done. Hence, worst-case execution times are upper bounds, and critical instants describe unrealistic execution sequences (but still give safe values). Extending the worst-case response time computation equations so as to take into account explicitly the overhead of each system activity during an execution sequence requires a large amount of work. In our opinion, when the software architecture is too complex to be analysed by an exact method, a more reasonable approach to treat system overheads is to include them as contributions to the worst-case execution time of code sections (e_{ij}).

Following this last approach, an immediate improvement could be to derive a worst-case context for each service call, so as to use more accurate overhead values. Indeed, the execution time of a system activity can sometimes be dependant of the number of objects handled by the system. For instance, in Trampoline [1, 7], the execution time for inserting a task in the ready list is composed of a constant part plus a variable one, linear in the number of higher priority tasks already in the ready list. Such an approach is immediate because: (i) OSEK/VDX OS is a static kernel; (ii) we know the software architecture of the application; (iii) we have access to the source code of our OSEK/VDX OS implementation.

5.3. ECCx classes

In this paper, we explore the BCCx conformance classes of the OSEK/VDX OS specifications. Obviously, they support model of computations that are close to the schedulability analysis models. However, they are limited and some application may require more complex services. Thus, we have to explore the ECCx conformance classes. The difference between ECCx and BCCx is the possibility for an extended task to enter a *WAITING* state, by waiting for one or more event.

By restricting the use of the event handling services, it should be simple to adapt the analysis algorithms defined for precedence graphs. To do so, an extended task could be considered as a chain of tasks with the same priority, where the calls to the *WaitEvent* services denote the limit between two tasks.

However, we have to perform further work to verify that this interpretation is correct and to precisely define the accompanying design rules. Moreover, such an approach forbids some constructs of interest, for instance the possibility for a task to wait for different events at the same time and to be awaken as soon as one of this event is signalled. Thus, we also have to extend analysis algorithms in order to be able to relax too restrictive rules.

5.4. Distributed systems

Another natural extension concerns the analysis and implementation of applications distributed among a set of networked ECUs (Electronic Control Unit). It is natural because the OSEK/VDX set of specifications includes an application-level communication protocol: OSEK/VDX COM.

On the one hand, depending on the chosen communication paradigm (COM supports both ”blackboard” and ”mailbox”), on the synchronization between the application layer and the communication subsystem, etc., a wide variety of software architecture can be implemented. On the other hand, there exists some results on the analysis of distributed real-time systems [17, 12]. Following the example of the work described in this paper, we have to adapt these analysis algorithms to the specific constructs of OSEK/VDX OS+COM, and to precisely define design rules that will ensure the predictability of the implementation.

5.5. Tool support

We are currently developing an open-source implementation of the OSEK/VDX OS 2.2.3 specifications [1, 7], together with an OSEK/VDX OIL 2.5 compiler in order to ease system generation.

In order to instantiate the work presented in this paper, we will define new OIL properties, so as to be able to directly extract schedulability analysis model from the OIL description of the application. Moreover, we plan to include specific rules in the OIL compiler to check that an application complies to the design rules associated to the selected analysis algorithms (these rules must also be able to parse and analyse the source code for tasks).

6. Conclusion

We have explored the problem of analysing the schedulability of OSEK/VDX-based applications. We focused on mono-ECU systems, and considered only the BCCx conformance classes of the OSEK/VDX OS specifications. In this context, we did study two cases:

- software architectures composed of a set of independent periodic and sporadic tasks;
- software architectures composed of a set of independent periodic and sporadic graphs of tasks;

In both cases, we have defined the software architecture model. For the first model, we have shown how to adapt state-of-the-art schedulability analysis algorithms so as to take into account the specific constructs of OSEK/VDX OS (mixed preemptive / non-preemptive scheduling, internal resources and task groups, mutliple tasks per priority level, etc.). Due to space limitation, we could not give the details of the schedulability analysis of the second model. The intersted reader may refer to [6]. In both case, we have also defined a set of design rules in order to ensure the consistency between the analysis model and the implementation.

Our approach is a pragmatic one. It aims at helping the designer of an application to fill the semantic gap between the analysis level and the implementation level. Although real-time operating systems are designed in order to achieve predictability of the applications, they also offer services the use of which may violates the hypothesis made by the analysis technique. The most simple answer to this problem consists in completely forbidding the usage of such services. However, this answer is unbearable in practice, as it hugely increases the complexity of the implementation. Hence, we try to give more reasonable answers, by restricting the usage of the problematic services (through design rules), while adapting the analysis algorithms. As we quickly discovered, such an approach requires a deep knowledge of both domains (analysis and services). In other words, solving the "simple" problem of "implementing an analysable application" is not straightforward.

The work presented in this paper is a first attempt to give reasonable answers. However, the design rules are still restrictive, so we should explore if they can be relaxed. This includes especially the extension of our proposal to the ECCx OSEK/VDX OS conformance classes. Then, it will be necessary to take into account the case of distributed systems. Lastly, by extending OIL, we will be able to provide a tool integrating system generation and schedulability analysis.

References

[1] J. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline an open source implementation of the OSEK/VDX RTOS. In *Proceeding of the Eleventh IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'06)*, 2006.

[2] F. Bimbard and L. George. FP/FIFO feasibility conditions with kernel overheads for periodic tasks on an event driven osek system. In *Proceeding of the Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 566–574, 2006.

[3] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley Longmain, 2001.

[4] J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2), 2003.

[5] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *Proceedings of the Twelfth Real-Time Systems Symposium (RTSS 1991)*, pages 116–128, 1991.

[6] P. Hladik, A. Déplanche, S. Faucou, and Y. Trinquet. Computation of worst-case response time of OSEK/VDX applications with precedence relations. `http://www.irccyn.ec-nantes.fr/~faucou/rtns07prec.pdf`, 2007.

[7] IRCCyN. Trampoline, real-time systems group. http://trampoline.rts-software.org, 2005.

[8] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the Eleventh IEEE Real-Time Systems Symposium (RTSS 1990)*, pages 201–209, 1990.

[9] J. Liu. *Real-Time Systems*. Prentice Hall Inc, 2000.

[10] OSEK. *OSEK/VDX System Generation – OIL : OSEK Implementation Language version 2.5*, 2004. `http://www.osek-vdx.org/`.

[11] OSEK group. *OSEK/VDX Operating System version 2.2.3*, 2005. `http://www.osek-vdx.org/`.

[12] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *proceedings of the Twentieth IEEE Real-time Systems Symposium (RTSS 1999)*, 1999.

[13] P. Puschner and A. Burns. Guest Editorial: A Review of Worst Case Execution Time Analysis. *Real-Time Systems*, 18(2-3):115–128, 2000.

[14] O. Redell and M. Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proceedings of the Eighth IEEE Real Time Technology and Applications Symposium (RTAS 2002)*, 2002.

[15] M. Richard. *Contribution à la Validation des Systèmes Temps Réel Distribués : Ordonnancement à Priorités Fixes & Placement*. PhD thesis, Université de Poitiers, 2002.

[16] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[17] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[18] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 328–337, 1999.

Improvements in the configuration and analysis of Posix 1003.1b scheduling

Mathieu Grenier Nicolas Navet
LORIA-INRIA
Campus Scientifique, BP 239
54506 Vandoeuvre-lès-Nancy- France
{grenier, nnavet}@loria.fr

Abstract

Posix 1003.1b compliant systems provide two well-specified scheduling policies, namely sched_rr (Round-Robin like) and sched_fifo (FPP like). Recently, an optimal priority and policy assignment algorithm for Posix 1003.1b has been proposed in the case where the quantum value is a system-wide constant. Here we extend this analysis to the case where quanta can be chosen on a task-per-task basis. The algorithm is shown to be optimal with regards to the power of the feasibility test (i.e. its ability to distinguish feasible and non feasible configurations). Though much less complex than an exhaustive exploration, the exponential complexity of the algorithm limits its applicability to small or medium-size problems. In this context, as shown in the experiments, our proposal allows achieving a significant gain in feasibility over FPP and Posix with system-wide quanta, and therefore using the computational resources at their fullest potential.

1. Introduction

Context of the paper. This study deals with the scheduling of real-time systems implemented on Posix 1003.1b compliant Operating System (OS). Posix 1003.1b [7], previously known as Posix4, defines real-time extension to Posix mainly concerning signals, inter-process communications, memory mapped files, synchronous and asynchronous IO, timers and scheduling (a recap of Posix's features related to scheduling is given in §2.1). This standard has become very popular and most of today's OS conform, at least partially, to it.

Problem definition. Posix 1003.1b compliant OSs provide two scheduling policies *sched_fifo* and *sched_rr*, which under some restrictions discussed in §2.1, are respectively equivalent to Fixed Preemptive Priority (FPP) and Round-Robin (RR for short). Thus, under Posix 1003.1b, each process is assigned both a priority, a scheduling policy and, in the case of Round-Robin, a quantum. At each point in time, one of the ready processes with the highest priority is executed, according to the rules of its scheduling policy (e.g. yielding the CPU after a quantum under RR).

The problem addressed here is to assign priorities, policies and quanta to tasks in such a way as to respect deadline constraints. For FPP alone, the well-known Audsley algorithm [2] is optimal. A similar algorithm exists for both RR and FPP in the case of a system-wide quantum [6]. Here we consider the case where quanta can be chosen on a task-per-task basis. As it will be seen in §3.2, the complexity of the problem is such that an exhaustive search is usually not feasible even on small size problems. For instance, a task set of cardinality 10 with quanta chosen among 5 different values requires to analyze the feasibility of more 10^{11} different configurations (see §3.2).

Contributions. Traditionally, the RR policy is only considered useful for low priority processes performing some background computation tasks "when nothing more important is running". In this paper, as we did in [10, 6], we argue that the combined use of RR and FPP allows to successfully schedule a large number of systems that are unschedulable with FPP alone.

The contribution of the paper is twofold, first we propose an algorithm for assigning priorities, policies and quanta that is optimal in the sense that if there exists at least a feasible solution[1], then the algorithm will return a feasible solution. The algorithm being an extension of the classical Audsley algorithm [2] and the *Audsley-RR-FPP* from [6], we name it the *Audsley-RR-FPP** algorithm. The worst-case complexity of the algorithm is assessed and a set of optimizations are proposed to reduce the search space. The second contribution of the paper is that we give further evidences that the combined use of both FPP and RR is effective - especially when quanta can be chosen for each individual task - for finding feasible schedules even when the workload of the system is high.

Related work. We identify two closely related lines of research: schedulability analyses and priority assignment.

[1] We call here a *feasible* solution, a solution that successfully passes a schedulability test verifying property 2 (see §2.5). In the following, we make use of the response time bound analysis derived in [9].

Audsley in [2, 3] proposes an optimal priority assignment algorithm for FPP, that is now well-known in the literature as the Audsley algorithm. Later on in [5], this algorithm has been shown to be also optimal for the non-preemptive scheduling with fixed priorities. The problem of best assigning priorities and policies under Posix 1003.1b was first tackled in [9] but the solution relies on heuristics and is not optimal in the general case. Then, in [6], an optimal solution is proposed for the case where the quantum value is a system-wide constant.

As in [6], the problem addressed here is different than in the plain FPP case because the use of RR leads to the occurrence of scheduling "anomalies", which are sometimes counter-intuitive. For instance, as it will be seen in §2.5, increasing the quantum value for a task can leads sometimes to a greater worst-case response time for this task. Similarly, decreasing the set of higher priority tasks, can increase the response time (see [6]). This prevents us from using the proposed priority and assignment algorithm with the schedulability assessed by simulation, or with a feasibility test that would not possess some specific properties discussed in §2.5. Indeed there would be cases where the algorithm would discard schedulable assignments and thus not be optimal. In this study, feasibility is assessed by the analysis published in [9], which ensures that the computed response time bounds decrease when the set of higher priority tasks is reduced. This property enables us to use an Audsley-like algorithm for the assignment that will be shown to be optimal with regard to the power of the test, that is its ability to distinguish feasible or non feasible configurations.

Organisation. Section *2* summarizes the main features of the scheduling under Posix 1003.1b and introduces the model and notations. In section 3, we present the optimal priority, policy and quantum assignment *Audsley-RR-FPP** algorithm. Efficiency of the proposal is then assessed in section 4.

2. Scheduling under Posix 1003.1b: model and basic properties

In this section we present the system model and summarize the main features related to scheduling of Posix 1003.1b. We then present the assumptions made in this study and derive some basic properties of the scheduling under Posix 1003.1b that will be used in the subsequent sections.

2.1. Overview of Posix 1003.1b scheduling

In the context of OS, we define a task as a recurrent activity which is either performed by repetitively launching a process or by a unique process that runs in cycle. Posix 1003.1b specifies 3 scheduling policies: *sched_rr*, *sched_fifo* and *sched_other*. These policies apply on a process-by-process basis: each process run with a particular scheduling policy and a given priority. Each process inherits its scheduling parameters from its father but may also change them at run-time.

- *sched_fifo* : fixed preemptive priority with First-In First-Out ordering among same-priority processes. In the rest of the paper, it will be assumed that all *sched_fifo* tasks of an application have different priorities. With this assumption and without change during run-time *sched_fifo* is equivalent to FPP.
- *sched_rr* : Round-Robin policy (RR) which allows processes of the same priority to share the CPU. Note that a process will not get the CPU until a higher priority ready-to-run processes are executed. The quantum value may be a system-wide constant (e.g. QNX OS), process specific (e.g. VxWorks OS) or fixed for a given priority interval.
- *sched_other* is an implementation-defined scheduler. It could map onto *sched_fifo* or *sched_rr*, or also implement a classical Unix time-sharing policy. The standard merely mandates its presence and its documentation. Because we cannot rely on the same behaviour of *sched_other* under all Posix compliant OSs, it is strongly suggested not to use it if a portability is a matter of concern. We will not consider it in our analysis.

Associated with each policy is a priority range. Depending on the implementation, these priority ranges may or may not overlap but most implementations allow overlapping. Note that these previously explained scheduling mechanisms similarly apply to Posix threads with the system contention scope as standardised by Posix 1003.1c standard [7].

2.2. System model

The activities of the system are modeled by a set $\mathcal{T}$ of n periodic and independent tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is characterized by a tuple (C_i, T_i, D_i) where each request of τ_i, called an instance, has an execution time of C_i, a relative deadline D_i and a period equal to T_i time units. One denotes by $\tau_{i,j}$ the jth release of τ_i. As usual, the response time of an instance is the time elapsed between its arrival and its end of execution.

Under Posix 1003.1b, see §2.1, each task τ_i possesses both a priority p_i and a scheduling policy $sched_i$. In this study, we choose the convention "the smaller the numerical value, the higher the priority". In addition to the priority, under RR, each task τ_i is assigned a quantum value ψ_i. The priority and scheduling policy assignment $\mathcal{P}$ is fully defined by a set of n tuples $(\tau_i, p_i, sched_i^{\mathcal{P}})$ (i.e. one for each task). A quantum assignment under $\mathcal{P}$, denoted by $\Psi_{\mathcal{P}}$, defines the set of quantum values $\psi_i^{\Psi_{\mathcal{P}}}$ where $\psi_i^{\Psi_{\mathcal{P}}}$ is the quantum of τ_i. The whole scheduling is fully defined by the tuple $(\mathcal{P}, \Psi_{\mathcal{P}})$ which is called a *configuration* of the system.

Under assignment $\mathcal{P}$, the set of tasks $\mathcal{T}$ is partitioned into separate layers, one layer for each priority level j

where the layer $\mathcal{T}_j^{\mathcal{P}}$ is the subset of tasks assigned to priority level j. Under $\mathcal{P}$, $\mathcal{T}_{hp(j)}^{\mathcal{P}}$ (resp. $\mathcal{T}_{lp(j)}^{\mathcal{P}}$) denotes the set of all tasks possessing a higher (resp. lower) priority than j. A layer in which all tasks are scheduled with RR (resp. FPP) is called an RR layer (resp. FPP layer). In the following, $\mathcal{P}$ or $\Psi_{\mathcal{P}}$ will be omitted when no confusions are possible. A list of the notations is provided in appendix at the end of the paper.

In the following, a task τ_i is said *schedulable* under assignment $(\mathcal{P}, \Psi_{\mathcal{P}})$ if its response time bound, as computed by the existing Posix 1003.1b schedulability analysis [9], is no greater than its relative deadline (i.e. maximum duration allowed between the arrival of an instance and its end of execution). The whole system is said schedulable if all tasks are schedulable. Note that the test presented in [9] is sufficient but not necessary, there are thus task sets which won't be classified as schedulable while there exist configurations under which no deadlines are missed.

2.3. Assumptions

In this study, as explained in *§2.1*, only *sched_fifo* and *sched_rr* are considered for portability concern. Due to the complexity of assigning priorities and scheduling policies, the following restrictions are made:

1. context switch latencies are neglected, but they could be included in the schedulability analysis of [9] as classically done (see, for instance, [11]),
2. since a priority level without any tasks has no effect on the scheduling, we impose the priority range to be contiguous,
3. two tasks having different scheduling policies have different priorities, i.e., $\forall i \neq j, sched_i \neq sched_j \implies p_i \neq p_j$,
4. all *sched_fifo* tasks must possess distinct priorities ($sched_i = sched_j = sched_fifo \implies p_i \neq p_j$). With this assumption and without priority change at run-time, *sched_fifo* is equivalent to fixed-preemptive priority (FPP). Thus, several tasks having the same priority are necessarily scheduled under *sched_rr* policy,
5. the quantum value can be chosen on a task-per-task basis in the interval $[\Psi_{\min}, \Psi_{\max}]$, where $\Psi_{\min}$ and $\Psi_{\max}$ are natural numbers whose values are OS-specific constraints or chosen by the application designer.

2.4. Schedulability analysis under Posix: a recap [9]

In this paragraph, we summarize the schedulability analysis [9] of a configuration $(\mathcal{P}, \Psi_{\mathcal{P}})$ under Posix. Tasks scheduled under Posix can be described as a superposition of priority layers [9]. At each point in time, one of the ready instances with the highest priority (let's say p_i) is executed as soon as and as long as no instances in the higher priority layers (instances of tasks in $\mathcal{T}_{hp(p_i)}$) are pending. Inside each priority layer, instances are scheduled either according to FPP or RR with the restrictions that all instances belonging to the same layer have the same policy.

FPP policy is achieved when a ready instance $\tau_{i,j}$ is executed when no higher priority instances is pending. Under RR, a task τ_i has repeatedly the opportunity to execute during a time slot of maximal length $\psi_i^{\Psi_{\mathcal{P}}}$. If the task has no pending instance or less pending work than the slot is long, then the rest of the slot is lost and the task has to wait for the next cycle to resume. The time between two consecutive opportunities to execute may vary, depending on the actual demand of the others tasks, but it is bounded by $\overline{\psi}_i^{\Psi_{\mathcal{P}}} = \sum_{\tau_k \in \mathcal{T}_{p_i}^{\mathcal{P}}} \psi_k^{\Psi_{\mathcal{P}}}$ in any interval where the considered task has pending instances at any moment. In [9], worst-case response time bounds for priority layers have been derived in a way that is independent from the scheduling policies used for each layer. This analysis is based on the concept of majorizing work arrival functions, which measure a bound on the processor demand, for each task, over an interval starting at a "generalized critical instant". The majorizing work arrival function on an interval of length t for a periodic task τ_i is:

$$s_i(t) = C_i \cdot \left\lceil \frac{t}{T_i} \right\rceil . \quad (1)$$

The worst-case response time bound can be expressed as

$$\max_{j<j^*} (e_{i,j} - a_{i,j}) , \quad (2)$$

where $j^* = \min\{j \mid e_{i,j} \leq a_{i,j+1}\}$, where $a_{i,j}$ is the release of the jth instance of τ_i after the critical instant and $e_{i,j}$ is a bound on the execution end of this instance. Since τ_i is a periodic task, $a_{i,j} = (j-1) \cdot T_i$ $(j = 1, 2, ...)$. If τ_i is in an FPP layer, then

$$e_{i,j} = \min\{t > 0 \mid \widetilde{s}_i(t) + s_{i,j} = t\} , \quad (3)$$

where $\widetilde{s}_i(t) = \sum_{\tau_k \in \mathcal{T}_{hp(p_i)}^{\mathcal{P}}} s_k(t)$ is the demand from higher priority tasks (i.e. task in $\mathcal{T}_{hp(p_i)}^{\mathcal{P}}$) and $s_{i,j} = \sum_{i=1}^{j} C_i$ is the demand from previous instances and the current instance of τ_i. If τ_i is in an RR layer, then

$$e_{i,j} = \min\{t > 0 \mid \overline{\Psi}_i(t) + s_{i,j} = t\} , \quad (4)$$

where the demand from higher priority tasks and of all other tasks of the RR layer is:

$$\overline{\Psi}_i(t) = \min \left(\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) + \widetilde{s}_i(t), s_i^*(t) \right) , \quad (5)$$

where $\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}$ is the sum of the quanta of all other tasks of the RR layer and

$$s_i^*(x) = \max_{u \geq 0} \left(s_i(u) + \widetilde{s}_i(u+x) + \overline{s}_i(u+x) - u \right) , \quad (6)$$

where $\overline{s}_i(u+x) = \sum_{\tau_k \in \mathcal{T}_{p_i}^{\mathcal{P}} \setminus \{\tau_i\}} s_k(u+x)$ is the demand from other tasks than τ_i in $\mathcal{T}_{p_i}^{\mathcal{P}}$. The algorithm for computing the worst-case response time bounds can be found in [9]. It is to stress that this schedulability analysis is sufficient but not necessary; some task sets may fail the test while they are perfectly schedulable. This will certainly induce conservative results but the approach developed here remains valid with another - better - schedulability test as long as it is sufficient and possesses the properties described in §2.5.

2.5. Scheduling under Posix 1003.1b: basic properties

Under FPP, as well as under RR, any higher priority task will preempt a lower priority task thus the following properties hold for any task τ_i :

1. all ready instances, with higher priorities than p_i, will delay the end-of-execution of the instances of τ_i. It is worth noting that this delay is not dependent on the relative priority ordering among these higher priority instances and their quantum values,
2. lower priority instances, whatever their policy, will not interfer with the execution of instances of τ_i and thus won't delay their end-of-execution.

These two properties ensure that the following lemma, which is well-known in the FPP case, holds.

Lemma 1 *[3] The worst-case response time of an instance of τ_i only depends on the set of same priority tasks, the values of their quantum and the set of higher priority tasks. The relative priority order among higher priority tasks and the values of their quantum has no influence.*

However, despite lemma 1 holding, scheduling under RR leads to scheduling anomalies. Indeed, scheduling under Posix is often counter-intuitive. For instance, it has been shown in [4], that early end-of-executions can lead to missed deadlines in configurations that would be feasible with WCETs. Similarly, removing a task with a higher priority than τ_j may lead to increased response times for τ_i (see figures 1 and 2 in [6]).

Here, we highlight that increasing the quantum size of a task can increase its response time. Figures 1 and 2 present the scheduling of task set $\mathcal{T} = \{\tau_1, \tau_2\}$ where $\tau_1 = (C_1 = 2, T_1 = 5)$ and $\tau_2 = (4, 10)$. All the tasks belong to the same layer and the chosen quantum assignments are $\Psi^{'} = \{\psi_1 = 2, \psi_2 = 2\}$ (figure 1) and $\Psi = \{\psi_1 = 2, \psi_2 = 3\}$ (figure 2) .

As it can be seen on figures 1 and 2, surprisingly the response time of τ_2 is 6 with a quantum of 2 and 8 with 3. However, with the schedulability analysis used in this study, property 1 holds and will be used to restrain the search space in section 3. A proof is given in appendix A.

Property 1 *Let τ_i be a task in a RR layer, increasing (resp. reducing) its quantum value, while reducing (resp. increasing) the quantum value of the other tasks of its RR layer, diminishes (resp. increases) the response time bound of τ_i computed with the chosen schedulability analysis.*

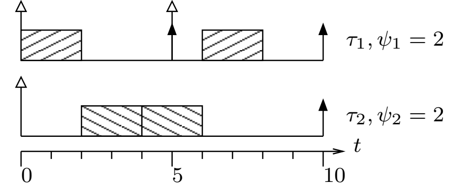


Figure 1. Scheduling of task set $\mathcal{T} = \{\tau_1, \tau_2\}$ with Round-Robin and quantum assignment $\Psi^{'} = \{\psi_1 = 2, \psi_2 = 2\}$.

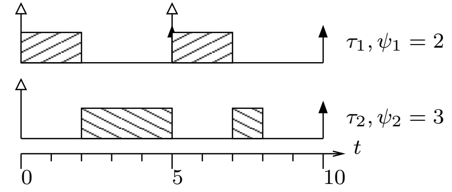


Figure 2. Scheduling of task set $\mathcal{T} = \{\tau_1, \tau_2\}$ with Round-Robin and quantum assignment $\Psi_1 = \{\psi_1 = 2, \psi_2 = 3\}$.

To be optimal, the Audsley algorithm requires that the schedulability test fulfills some properties (see §3.3). In particular, removing a task with a higher priority must not lead to increased response times. In the case of Posix 1003.1b, this imposes constraints on the schedulability test which must fulfill property 2.

Property 2 *Let τ_i be a task in RR or FPP layer, reducing its set of higher and same priority tasks, while keeping the quantum allocation unchanged within its Round-Robin layer (if τ_i is scheduled under RR), diminishes or leaves unchanged the response time bound of τ_i computed with the chosen schedulability analysis.*

It has been shown in [6] that the conservative response time bound computed with [9] ensures that property 2 holds. The proof, given in [6] in the context of a unique sysrem-wide quantum value, is still valid when different values for the quanta are possible. As it will be shown in section 3, a schedulability test which ensures that property 2 is verified, allows to use an extension of the Audsley algorithm and preserves its optimality with regards to the ability of the test to distinguish between feasible and non-feasible solutions (i.e., what is called the power of the test in the following).

3. Optimal assignment algorithm with task-specific quanta

We present here an optimal priority, scheduling policy and quanta assignment for Posix 1003.1b systems when the feasibility is assessed with schedulability analysis which verifies property 2 described in §2.5. This algorithm heavily relies on both the Audsley algorithm and the algorithm previously proposed for system-wide quantum values (called *Audsley-RR-FPP* in [6]). Here we extend previous works to the case where quanta can be chosen on a task-per-task basis, the corresponding algorithm is named the *Audsley-RR-FPP**. With the assumption made in section 2, the policy is implied by the number of tasks having the same priority level: should only one task be assigned priority level i then its policy is FPP (i.e. a RR layer of cardinality 1 is strictly equivalent to an FPP layer, see §2.1), otherwise the policy is necessarily RR. The problem is thus reduced to assigning priorities and quanta to tasks in a RR layer.

3.1. *Audsley-RR-FPP** algorithm

In the same way as the original Audsley algorithm (abridged by AA in the following), the idea is to start assigning the priorities from the lowest priority n to the highest priority 1 (line 3 in algorithm 1). The difference with AA, is that, at each priority level, the algorithm is not looking for a single task but for a set of tasks (line 5). For each such set of tasks, our algorithm examines all possible quantum assignments until it finds one suitable one.

Underlying idea. The underlying idea of the algorithm is to move , when needed, the maximum amount of workload to the lower priority levels and to schedule the tasks under RR. When an instance $\tau_{i,j}$ is assigned the same priority as $\tau_{k,h}$ and both are scheduled under RR, $\tau_{i,j}$ can delay $\tau_{k,h}$ less than if $\tau_{i,j}$ would be scheduled with a higher priority. The same argument holds for the delay induced by $\tau_{k,h}$ to $\tau_{i,j}$. Thus, as illustrated with an example in [10], where a task set that is not feasible under FPP alone, becomes feasible with RR. Of course, in the general case, combining the use of both policies is the most efficient and, as it will be shown, leads to an optimal priority and policy assignment.

Step of the algorithm. For each priority level i (line 3), the *Audsley-RR-FPP** algorithm attempts to find a schedulable subset $\mathcal{T}_i$ in subset $\mathcal{R}$ (line 5) where $\mathcal{R}$ is made of all the tasks which have not been yet assigned a priority, a policy and a quantum. The algorithm tries all possible subsets of $\mathcal{R}$, one by one, and all possible quantum assignments for each subset until a schedulable configuration is obtained or all configurations have been considered. In the latter case, the system is not schedulable (lines 7-8). Otherwise, we have found a schedulable subset, denoted by $\mathcal{T}_i$, which, in the RR case, possesses quantum assignment $\{\psi_k\}_{\tau_k \in \mathcal{T}_i}$ (lines 7 and 8). Precisely,

Input: task set $\mathcal{T} = \{\tau_1, \tau_2 ..., \tau_n\}$
Result: schedulable priority, scheduling policy and quantum assignment $\mathcal{P}_k = (\mathcal{P}, \Psi_k)$
Data: i: priority level to assign
$\mathcal{R}$: task-set with no assigned priority
$\mathcal{P}$: partial priority and policy assignment
$\Psi_{\mathcal{P}}$: partial quantum allocation

```
1  R = T;
2  P = ∅;
3  for i = n to 1 do
4      try to assign priority i:
5      search a schedulable subset of tasks T_i under
       quantum allocation {ψ_k}_{τ_k ∈ T_i} in R
6      if no subset T_i is schedulable at priority i then
7          failure, return partial
           assignement:
8          return (P, Ψ_P);
9      else
10         let T_i a schedulable subset at priority i with
           quantum allocation {ψ_k}_{τ_k ∈ T_i};
11         assign priority, policy and
           quantum:
12         if #T_i = 1 then
13             P = P ∪ {(τ_k, i, sched_fifo)}_{τ_k ∈ T_i};
14         else
15             P = P ∪ {(τ_k, i, sched_rr)}_{τ_k ∈ T_i};
16             Ψ_P = Ψ_P ∪ {ψ_k}_{τ_k ∈ T_i};
17         end
18         remove T_i from R:
19         R = R\T_i;
20     end
21     if R = ∅ then return (P, Ψ_P);
22 end
```

Algorithm 1: *Audsley-RR-FPP** algorithm with task-specific quantum.

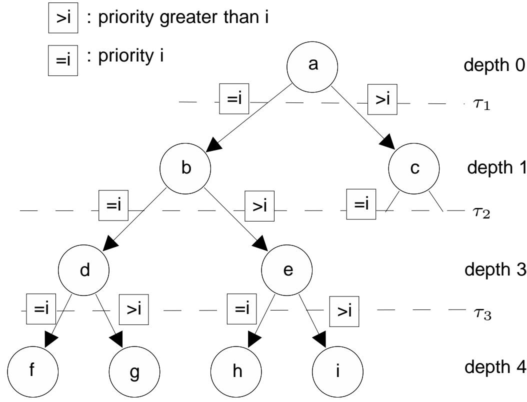


Figure 3. Search tree constructed in the search of a feasible subset of $\mathcal{R} = \{\tau_1, \tau_2, \tau_3\}$ at priority i. For instance, node b models the partial priority assignment where τ_1 is assigned priority i while node c means that τ_1 is assigned a greater priority.

$\mathcal{T}_i$ is schedulable when all tasks of $\mathcal{T}_i$ are feasible at priority i while all tasks without assignment (i.e., tasks in $\mathcal{R}\backslash\mathcal{T}_i$) have a priority greater than i. At each step, at least one task is assigned a priority and a policy (lines 11 to 17). Note that, when RR is used at least once, less than n priority levels are needed (early exit on line 21).

Looking for the set of schedulable tasks $\mathcal{T}_i$. There are $2^{\#\mathcal{R}}$ possible subsets $\mathcal{T}_i$ of $\mathcal{R}$ that can be assigned priority level i (line 5). Since the quantum can take $\|\psi\| = \psi_{\max} - \psi_{\min} + 1$ different values, there are $\|\psi\|^{\#\mathcal{T}_i}$ different quantum assignments for each subset $\mathcal{T}_i$. First, we explain the basic exhaustive tree-search used to set priorities. Then, we explain how we use a similar search to choose the quantum assignment for each possible set $\mathcal{T}_i$. A method that speeds-up the search by pruning away subtrees that cannot contain a solution is provided in §3.2.

A binary tree structure reflects the priority choices and the search for the schedulable subset is performed by exploring the tree. In the following, we call *priority-search-tree* the search tree modeling the priority choices. As an illustration, figure 3 shows the priority-search-tree corresponding to the set $\mathcal{R} = \{\tau_1, \tau_2, \tau_3\}$. Each edge is labeled either with "$= i$" (i.e., priority equal to i) or "$> i$" (i.e., priority greater than i). A label "$= i$" (resp. "$> i$") on the edge between vertices of depth k and $k+1$ means that the $(k+1)^{th}$ task of $\mathcal{R}$ belongs to the layer of priority i (resp. belongs to a layer of priority greater than i). Thus, a vertex of depth k models the choices performed for the k first tasks of $\mathcal{R}$. For instance, on figure 3, the vertex e implies that tasks τ_1 belongs to layer of priority i while task τ_2 does not. Each leaf is a complete assignment for priority level i, for instance leaf g corresponds to set $\mathcal{T}_i = \{\tau_1, \tau_2\}$.

The search is performed according to a depth-first strategy. The algorithm considers the first child of a vertex that appears and goes deeper and deeper until a leaf is reached, i.e., until the set $\mathcal{T}_i$ is fully defined. When a leaf is reached, the schedulability of $\mathcal{T}_i$ is assessed. If $\mathcal{T}_i$ is feasible, the algorithm returns, otherwise, it backtracks till the first vertex such that not all its child vertices have been explored.

To assess the schedulability of $\mathcal{T}_i$, all possible quantum assignments are successively considered. In the same manner as for the priority allocation, a tree -called *quantum-search-tree*- reflects the choices for quantum values. A depth-first strategy is used as well to explore the search space. In this case, a node has $\|\psi\|$ children where each child models a different quantum value. Here, we label the edge between vertices of depth k and $k+1$ with the quantum value of the $(k+1)^{th}$ task of $\mathcal{T}_i$. Thus, a vertex of depth k models the choices performed for the k first tasks of $\mathcal{T}_i$.

3.2. Complexity and improvements

Size of the search space. Assigning n tasks to different non-empty layers is like subdividing a set of n elements into non-empty subsets. Let k be the number of layers. The number of possible assignments is equal, by definition, to the the Stirling number of the second kind (see [1], page 824):

$$\frac{1}{k!}\sum_{i=0}^{k}(-1)^{(k-i)}\left(\begin{array}{c}k\\i\end{array}\right)i^n,$$

where $\left(\begin{array}{c}k\\i\end{array}\right)$ is the binomial coefficient, i.e., the number of ways of picking an unordered subset of i elements in a set of k elements.

The complexity depends on the number of tasks scheduled under RR since their quantum values have to be chosen. When there are k layers, at least $n-k+1$ tasks are in an RR layer (i.e., $n-k+1$ tasks in a single RR layer and one task in each of the remaining $k-1$ FPP layers) and up to $\max(n, 2(n-k))$ (i.e., tasks are "evenly" distributed among RR layers). Since the quantum can take $\|\psi\| = \psi_{\max} - \psi_{\min} + 1$ different values, there are between $\|\psi\|^{n-k+1}$ and $\|\psi\|^{\max(n,2(n-k))}$ different quantum assignments for a configuration of k layers.

In addition, n tasks can be subdivided into $k = 1, 2, ..., n$ many layers and there are $k!$ different possible priority orderings among the k priority layers. Thus, a lower bound for the search space of the problem of assigning priority, policy and quantum for a set of n tasks is

$$\sum_{k=1}^{n}\sum_{i=0}^{k}(-1)^{(k-i)}\cdot\left(\begin{array}{c}k\\i\end{array}\right)\cdot i^n\cdot\|\psi\|^{n-k+1}.$$

In a similar way, we derive an upper bound by replacing $\|\psi\|^{n-k+1}$ with $\|\psi\|^{\max(n,2(n-k))}$.

For instance, as can be seen on figure 4, the size of the search space comprises about $4\cdot 10^{10}$ scheduling configurations for a set of 10 tasks. The search space grows

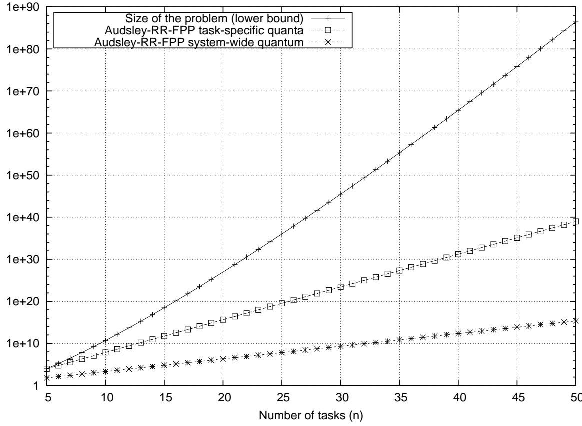


Figure 4. Complexity of the problem for a number of tasks varying from 5 to 50 when the quantum value can be chosen in the interval $[1, 5]$.

more than exponentially, thus an exhaustive search is not possible in practice in a wide range of real-time problems.

Audsley-RR-FPP*.* Our algorithm looks at each priority level i for a subset $\mathcal{T}_i$ in $\mathcal{R}$ which is schedulable at priority i (line 5). Since at least one task is assigned to each priority level, the number of tasks belonging to $\mathcal{R}$ when dealing with priority level i is lower than or equal to i. In addition, we know that there are $\|\psi\|^k$ different quantum assignments for a subset of k tasks. Thus, at each priority level i, the algorithm examines $\sum_{j=1}^{i} \left(\begin{array}{c} i \\ j \end{array} \right) \cdot \|\psi\|^j = (\|\psi\| + 1)^i - 1$ assignments in the worst-case. Thus, for priority level from 1 to n, the algorithm considers in the worst-case a number of assignments given by:

$$\sum_{i=1}^{n} (\|\psi\| + 1)^i - 1 = \frac{1 - (\|\psi\| + 1)^{n+1}}{1 - (\|\psi\| + 1)} - (n + 1)$$

This complexity for a varying number of tasks is shown on figure 4, for instance, for a set of 10 tasks with $\psi_{min} = 1$ and $\psi_{max} = 5$ it is approximately equal to $72 \cdot 10^6$. Figure 4 shows also the size of the search space and, for comparison, the worst-case complexity of the solution proposed in [6] in the case where the quantum size is a system-wide constant. Although we achieve a great complexity reduction with regards to an exhaustive search, the complexity remains exponential in the number of tasks. Thus, in practice, our proposal is not suited for large-size task sets that would, for instance, be better handled by heuristics guiding the search towards promising parts of the search space. This is left as future work.

Complexity reduction. As seen before, the *Audsley-RR-FPP** performs an exhaustive search for each priority level. To a certain extent, it is possible to reduce the number of sets that are to be considered. Indeed, the property 3 given in this paragraph shows that it is possible to identify priority and policy assignments that are not schedulable whatever the quantum allocation. Thanks to property 2 and property 3, one can identify and prune away branches of the priority-search-tree which necessarily lead to subsets $\mathcal{T}_i$ that are not schedulable whatever the quantum assignments. Furthermore, with property 3, one can reduce in a similar manner the number of quantum assignments to consider for a particular subset $\mathcal{T}_i$ in a quantum-search-tree.

With the basic algorithm explains in §3.1, feasibility of a priority allocation is assessed at the leafs when all tasks have been given a priority by testing all quantum assignments. The idea is here to evaluate feasibility at intermediate vertices as well, by assigning a priority lower than i to the tasks for which no priority choice has been made yet. Under that configuration, if a task τ_i which is assigned the priority i is not schedulable whatever the quantum assignment, there is no need to consider the children of this vertex. Indeed, from property 2, since the priority assignment of the children of this node will increase the set of same or higher priority tasks, the response time of τ_i cannot decrease. Thus, all child vertices corresponds to priority assignments that are not schedulable. Now, it remains to identify priority and policy assignments that are not schedulable whatever the quantum allocation. The following property, proven in appendix A.3, can be stated.

Property 3 *Let $\mathcal{S}$ be a schedulability test for which property 2 holds. Let $\mathcal{T}$ be a task set and $\mathcal{P}$ be a global priority and policy assignment. Let τ_i be a task with the maximum quantum value ψ_{max} in an RR layer. Let the quantum values of all other tasks in the RR layer be set to the minimum ψ_{min}. If the response time bound of τ_i, computed with $\mathcal{S}$, is greater than its relative deadline, then, whatever the quantum assignment under $\mathcal{P}$, τ_i will remain unschedulable with $\mathcal{S}$.*

Thus, at each vertex of the priority search tree, a priority assignment $\mathcal{P}$ is not feasible whatever the quantum assignment, if a task τ_k which has a priority i is not feasible with the quantum allocation given in property 3.

Similarly, we can cut branches when exploring the quantum-search-tree of a set $\mathcal{T}_i$. The idea is again to evaluate feasibility at intermediate vertices. Since an intermediate vertex models a partial quantum assignment for a set $\mathcal{T}_i$, we assign the lowest quantum value to each task in $\mathcal{T}_i$ which has no quantum assigned yet. In that case, if a task τ_k for which the quantum has already been set at this vertex is not schedulable, then there is no need to consider the children of this vertex. Indeed, given property 1, the response time of τ_k can only increase when the the children of this vertex are considered.

The finding of this paragraph allows a very significant decrease in the average number of configurations tested by the *Audsley-RR-FPP** algorithm. For instance, for task sets constituted of 10 tasks, the algorithm examines on average only about 4000 configurations before coming up

with a feasible solution or concluding that the task set is unfeasible while it would require about $7 \cdot 10^7$ tests otherwise.

3.3. Proof of optimality

Here we show that the *Audsley-RR-FPP** algorithm is optimal in the sense that if there is a priority, policy and quantum assignment that can be identified as feasible by the schedulability analysis, it will be found by the algorithm. Let us first remind the following theorem which has been proven in [2, 3, 5, 9] for various contexts of fixed priority scheduling.

Theorem 1 *[3] Let $(\mathcal{P}, \Psi_{\mathcal{P}})$ be a schedulable configuration up to priority i, i.e. tasks that have been assigned the priorities from n to i are schedulable. If there exists a schedulable configuration $(\mathcal{A}, \Psi_{\mathcal{A}})$, then there is at least one schedulable configuration $(\mathcal{Q}, \Psi_{\mathcal{Q}})$ having an identical configuration as $(\mathcal{P}, \Psi_{\mathcal{P}})$ for priorities n to i.*

From theorem 1, we can prove the optimality of *Audsley-RR-FPP**. Indeed, if *Audsley-RR-FPP** happens to fail at level i, the priority, scheduling policy and quantum assignment $(\mathcal{P}, \Psi_{\mathcal{P}})$ provided by *Audsley-RR-FPP** leads to a schedulable solution up to level $i + 1$. Since *Audsley-RR-FPP** performs an exhaustive search to assign level i, there cannot be any *schedulable* assignment $(\mathcal{Q}, \Psi_{\mathcal{Q}})$ possessing the same assignment as $(\mathcal{P}, \Psi_{\mathcal{P}})$ for priority $i + 1$ to n. Thus, from theorem 1, there is no schedulable assignment.

We give here an intuitive proof of theorem 1, which basically is valid under Posix thanks to lemma 1 and property 2. It should be pointed out that theorem 1, and thus the optimality result of *Audsley-RR-FPP**, does not hold where property 2 is not verified by the schedulability test.

Theorem 1 holds if a schedulable configuration $(\mathcal{A}, \Psi_{\mathcal{A}})$ can be transformed into a schedulable configuration $(\mathcal{Q}, \Psi_{\mathcal{Q}})$ for which the configuration is the same as $(\mathcal{P}, \Psi_{\mathcal{P}})$ for priority i to n. This transformation can be done iteratively by changing the configuration of certain tasks in $(\mathcal{A}, \Psi_{\mathcal{A}})$ to the configuration they have in $(\mathcal{P}, \Psi_{\mathcal{P}})$. The procedure is the following: for priority level k from n to i, assign in $(\mathcal{A}, \Psi_{\mathcal{A}})$ the priority $k+n-i$ to the tasks of priority k in $(\mathcal{P}, \Psi_{\mathcal{P}})$ (i.e., the set $\mathcal{T}_k^{\mathcal{P}}$) and set their quantum value to their values $\psi_i^{\mathcal{P}}$ in $\Psi_{\mathcal{P}}$ ($\forall \tau_j \in \mathcal{T}_k^{\mathcal{P}}, p_j^{\mathcal{A}} = p_j^{\mathcal{P}} + n - i$, $sched_j^{\mathcal{A}} = sched_j^{\mathcal{P}}$ and $\psi_j^{\Psi_{\mathcal{A}}} = \psi_j^{\Psi_{\mathcal{P}}}$). Since at each step, tasks in $\mathcal{T}_k^{\mathcal{P}}$ have the same quantum assignment, the same set of higher and equal priority tasks under the current configuration $(\mathcal{A}, \Psi_{\mathcal{A}})$ as under $(\mathcal{P}, \Psi_{\mathcal{P}})$, they remain schedulable under $(\mathcal{A}, \Psi_{\mathcal{A}})$ by lemma 1. From property 2, the other tasks $(\mathcal{T} \setminus \mathcal{T}_k^{\mathcal{P}})$ meet their deadline too since the quantum assignment and the set of higher and same priority task is reduced or stay unchanged under current configuration compared to the initial configuration $(\mathcal{A}, \Psi_{\mathcal{A}})$. Note that in the proof the priority range has been artificially extended by adding $n - i$ lower priority levels in order to avoid the case where a higher priority tasks is moved to a non-empty layer since property 2 does not cover this situation.

4. Experimental results

Here our aim is to quantify the extent to which using task-specific quanta enables us to improve the schedulability of the system by comparison 1) with FPP and 2) with system-wide quanta.

4.1. Experimental setup.

In the following experiments, we only consider task sets that are unschedulable with FPP alone. Since we choose to consider periodic tasks with deadlines equal to periods ($D_i = T_i$), we use the Rate Monotonic priority assignment, which is optimal in that context. The global load U (i.e., $\sum_{i=1}^{n} \frac{C_i}{T_i}$) has to be necessarily greater than $n \cdot (2^{1/n} - 1)$ (from [8]) in order to be able to exhibit non-feasible task sets. In the following, we choose a quantum value of 1 for the system-wide quantum or, when task-specific quanta is considered, a quantum value which can be chosen in the interval $[1, 5]$. The actual parameters of an experiment are defined by the tuple (n, U). The utilization rate ($\frac{C_i}{T_i}$) of each task τ_i is uniformly distributed in the interval $\left[\frac{U}{n} \cdot 0.9, \frac{U}{n} \cdot 1.1\right]$ where n is the number of tasks. The computation time C_i is randomly chosen with an uniform law in the interval $[1, 30]$ and the period T_i is upper bounded by 500. The results shown on figure 5 have been obtained with 200 task sets randomly generated with the aforementioned parameters.

4.2. Schedulability improvement over FPP and system-wide quanta

Figure 5 shows the percentage of task sets that are not schedulable with FPP alone and become schedulable when using the *Audsley-RR-FPP**(task-specific quanta) and *Audsley-RR-FPP* (system-wide quanta - see [6]) algorithms that are both optimal in their context. One observes that the improvement with task-specific quanta is very important, at least 3 times better than with a system-wide quantum. When the load is lower than 84%, a solution is found in almost all cases, the percentage of successes remaining greater than 50% up to a load equal to 88%. As it was to be expected, when the load gets higher, feasible scheduling solution tends to rarefy.

Our experiments show that the combined used of RR and FPP with process-specific quanta allows to schedule a large number of task sets which are neither schedulable with FPP nor with a system-wide quantum. It is worth noting that context switch latencies were neglected while RR induces more context switches than FPP. This fact weakens to a certain extent our conclusions. A future work is to find the feasible quantum allocation that minimizes the global number of context switches.

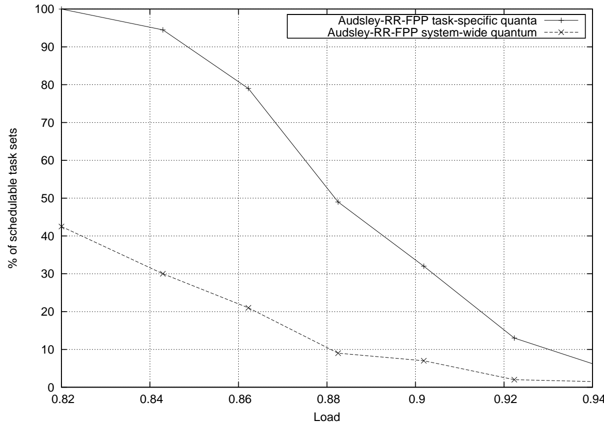


Figure 5. Percentage of task sets unschedulable with DM which become schedulable under Posix using the *Audsley-RR-FPP**(task-specific quanta) and *Audsley-RR-FPP* (system-wide quanta - see [6]) algorithms. The CPU load ranges from 0.82 to 0.94. The number of tasks is equal to 10.

5. Conclusion

In this paper, we propose a priority, policy and quantum assignment algorithm for Posix 1003.1b compliant OS that we named the *Audsley-RR-FPP**. We have shown this algorithm to be optimal in the sense that if there is a feasible schedule using FPP and RR that can be identified as such by the schedulability test, it will be found by the algorithm. A result yields by the experiments is that the combined used of FPP and RR with process-specific quanta enables to significantly improve schedulability by comparison with FPP alone and with system-wide quanta. This is particularly interesting in the context of embedded systems where the cost pressure is high, which lead us to exploit the computational resources at their fullest.

In terms of worst-case complexity, the algorithm greatly improves upon an exhaustive exploration of the search space but is still exponential in the number of tasks in the worst-case. Therefore, it is not suited to large task sets and future work is needed to develop techniques able to handle such systems. A future work is to take into account context switches and come up with a way of assigning quantum values in such a manner as to minimize the context-switch overhead.

References

[1] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions*. Dover Publications (ISBN 0-486-61272-4), 1970.

[2] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Report YO1 5DD, Dept. of Computer Science, University of York, England, 1991.

[3] N.C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001.

[4] R. Brito and N. Navet. Low-power round-robin scheduling. In *Proc. of the 12th international conference on real-time systems (RTS 2004)*, 2004.

[5] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report RR-2966, INRIA, 1996. Available at http://www.inria.fr/rrrt/rr-2966.html .

[6] M. Grenier and N. Navet. Scheduling configuration on Posix 1003.1b systems. Technical report, INRIA, to appear, 2007.

[7] (ISO/IEC) 9945-1:2004 and IEEE Std 1003.1, 2004 Edition. Information technology—portable operating system interface (POSIX®)—part 1: Base definitions. IEEE Standards Press, 2004.

[8] C.L. Liu and J.W Layland. Scheduling algorithms for multiprogramming in hard-real time environnement. *Journal of the ACM*, 20(1):40–61, 1973.

[9] J. Migge, A. Jean-Marie, and N. Navet. Timing analysis of compound scheduling policies : Application to Posix1003.1b. *Journal of Scheduling*, 6(5):457–482, 2003.

[10] N. Navet and J. Migge. Fine tuning the scheduling of tasks through a genetic algorithm: Application to Posix1003.1b compliant OS. *Proc. of IEEE Proceedings Software*, 150(1):13–24, 2003.

[11] K. Tindell. An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS-92-189, Department of Computer Science, University of York, 1992.

A. Proof of properties 1 and 2

In this appendix, we prove that the schedulability analysis [9] ensures that properties 1 and 3 hold. The first paragraph is devoted to the study of the execution end $e_{i,j}$ of $\tau_{i,j}$ computed with [9] under two configurations $(\mathcal{P}, \Psi_\mathcal{P})$ and $(\mathcal{P}, \Psi'_\mathcal{P})$ that only differ by their quantum assignment. This result is used in subsequent proofs.

A.1. Execution end bound: basic properties

We compare bounds on the execution end of τ_i under the same priority and policy assignment $\mathcal{P}$ with two different quantum allocations. Let $e_{i,j}$ and $e'_{i,j}$ be respectively the execution end bound of τ_i under $(\mathcal{P}, \Psi_\mathcal{P})$ and under $(\mathcal{P}, \Psi'_\mathcal{P})$. Since τ_i is in an RR layer, $e_{i,j}$ is computed with equation 4 of §2.4:

$$e_{i,j} = \min\{t > 0 \mid \overline{\Psi}_i(t) + s_{i,j} = t\} ,$$

where (equation 5 of §2.4)

$$\overline{\Psi}_i^{\Psi_{\mathcal{P}}}(t) = \min\left(\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) + \widetilde{s}_i(t), s_i^*(x)\right) ,$$

where $\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}$ is the sum of the quanta of all other tasks of the RR layer. Since $s_{i,j}$, $\widetilde{s}_i(t)$ and $s_i^*(x)$ are independent of the quantum assignment (see §2.4), it is enough to compare the first term of the $\min()$ to decide which task will have the smallest response time bound. Two cases arise:

1. $\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) > \left\lceil \frac{s_{i,j}}{\psi_i^{\Psi'_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi'_{\mathcal{P}}} - \psi_i^{\Psi'_{\mathcal{P}}})$ then we conclude $e_{i,j} \geq e'_{i,j}$,

2. otherwise:

$$\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) \leq \left\lceil \frac{s_{i,j}}{\psi_i^{\Psi'_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi'_{\mathcal{P}}} - \psi_i^{\Psi'_{\mathcal{P}}}) ,$$

   and $e_{i,j} \leq e'_{i,j}$.

When $s_i^*(x)$ is the minimum, we have $e_{i,j} = e'_{i,j}$.

From this finding we can deduce that for any other assignment $\Psi'_{\mathcal{P}}$, if the two following requirements are met:

requirement 1: the quantum $\psi_i^{\Psi'_{\mathcal{P}}}$ of τ_i in $\Psi'_{\mathcal{P}}$ is lower than or equal to its quantum $\psi_i^{\Psi_{\mathcal{P}}}$ under $\Psi_{\mathcal{P}}$,

requirement 2: the sum of the quanta of all other tasks of the RR layer $\mathcal{T}_i^{\mathcal{P}}$ under $\Psi'_{\mathcal{P}}$ is greater than or equal to the one under $\Psi_{\mathcal{P}}$, i.e., $\overline{\psi}_i^{\Psi'_{\mathcal{P}}} - \psi_i^{\Psi'_{\mathcal{P}}} \geq \overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}$ where $\overline{\psi}_i^{\Psi_{\mathcal{P}}} = \sum_{\tau_k \in \mathcal{T}_i} \psi_{\tau_k}^{\Psi_{\mathcal{P}}}$ is the sum of the quantum of all tasks of the RR layer $\mathcal{T}_i^{\mathcal{P}}$ under quantum allocation $\Psi_{\mathcal{P}}$,

then we have:

$$\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi'_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi'_{\mathcal{P}}} - \psi_i^{\Psi'_{\mathcal{P}}}) \geq \left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) , \quad (7)$$

and thus $\forall \tau_{i,j}$, $e_{i,j} \leq e'_{i,j}$ which implies that the response time bound of τ_i under $(\mathcal{P}, \Psi'_{\mathcal{P}})$ is greater than or equal to the response time bound under $(\mathcal{P}, \Psi_{\mathcal{P}})$.

A.2. Proof of property 1

Since the prerequisites of property 3 are exactly requirements 1 and 2 of §A.1, the response time bound of τ_i in property 3, is no less under $(\mathcal{P}, \Psi'_{\mathcal{P}})$ than under $(\mathcal{P}, \Psi_{\mathcal{P}})$. Since τ_i is not schedulable under $(\mathcal{P}, \Psi_{\mathcal{P}})$, it cannot be schedulable under $(\mathcal{P}, \Psi'_{\mathcal{P}})$.

A.3. Proof of property 3

We show that the bound on the execution end $e_{i,j}$ for a task in an RR layer under $\mathcal{P}$, is minimum under $\mathcal{P}$ when the quantum of τ_i is equal to ψ_{max} while the quanta of the other tasks in the layer are set to ψ_{min}. Let $\Psi_{\mathcal{P}}$ be the corresponding quantum assignment where

$$\left\lceil \frac{s_{i,j}}{\psi_i^{\Psi_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi_{\mathcal{P}}} - \psi_i^{\Psi_{\mathcal{P}}}) = \left\lceil \frac{s_{i,j}}{\psi_{max}} \right\rceil \cdot (\sum_{\tau_k \in \mathcal{T}_{p_i}^{\mathcal{P}} \setminus \{\tau_i\}} \psi_{min})$$

and one notes that whatever a different quantum assignment $\Psi'_{\mathcal{P}}$:

$$\left\lceil \frac{s_{i,j}}{\psi_{max}} \right\rceil \cdot \sum_{\tau_k \in \mathcal{T}_{p_i}^{\mathcal{P}} \setminus \{\tau_i\}} \psi_{min} \leq \left\lceil \frac{s_{i,j}}{\psi_i^{\Psi'_{\mathcal{P}}}} \right\rceil \cdot (\overline{\psi}_i^{\Psi'_{\mathcal{P}}} - \psi_i^{\Psi'_{\mathcal{P}}})$$

since, by definition, $\psi_{max} \geq \psi_i^{\Psi'_{\mathcal{P}}}$ and $\psi_{min} \leq \psi_k^{\Psi'_{\mathcal{P}}}$. From equation 7, the execution end bound $e_{i,j}$ of $\tau_{i,j}$ is thus minimum with $\Psi_{\mathcal{P}}$ among the set of all possible quantum assignments.

Notations

- $\mathcal{T} = \{\tau_1, ..., \tau_n\}$: a set of n periodic tasks
- $\mathcal{P}$: priority and policy assignment
- $\Psi_{\mathcal{P}}$: a specific quantum allocation under assignment $\mathcal{P}$
- $(\mathcal{P}, \Psi_{\mathcal{P}})$: a priority, policy and a quantum assignment
- $\mathcal{T}_i^{\mathcal{P}}$: subset of tasks assigned to priority level i under $\mathcal{P}$
- $\mathcal{T}_{hp(i)}^{\mathcal{P}}$: subset of tasks assigned to a higher priority than i under $\mathcal{P}$
- $\mathcal{T}_{lp(i)}^{\mathcal{P}}$: subset of tasks assigned to a lower priority than i under $\mathcal{P}$
- $\psi_i^{\Psi_{\mathcal{P}}}$: Round-Robin quantum for task τ_i under $\Psi_{\mathcal{P}}$
- $\overline{\psi}_i^{\Psi_{\mathcal{P}}}$: sum of the quanta of all tasks in layer $\mathcal{T}_i$ under $\Psi_{\mathcal{P}}$
- $s_i(t) = C_i \cdot \left\lceil \frac{t}{T_i} \right\rceil$: majorizing work arrival function on an interval of length t for a periodic task τ_i
- $\widetilde{s}_i(t) = \sum_{\tau_k \in \mathcal{T}_{hp(p_i)}^{\mathcal{P}}} s_k(t)$: the demand from higher priority tasks under $\mathcal{P}$
- $s_{i,j} = \sum_{i=1}^{j} C_i$: the demand from previous instances plus demand of current instance $\tau_{i,j}$ of τ_i
- $\overline{s}_i(x) = \sum_{\tau_k \in \mathcal{T}_{p_i}^{\mathcal{P}} \setminus \{\tau_i\}} s_k(x)$ is the demand from all other tasks than τ_i at priority level i under assignment $\mathcal{P}$.

An Extended Scheduling Protocol for the Enhancement of RTDBSs Performances *

S. Semghouni*, B. Sadeg*, L. Amanton* and A. Berred**
Laboratoire L.I.T.I.S antenne du Havre*,
Laboratoire de Mathématiques Appliquées du Havre**
Université du Havre 25 rue Philippe Lebon BP 540
76058 Le Havre Cedex, FRANCE
{samy-rostom.semghouni, bruno.sadeg, laurent.amanton, alexandre.berred}@univ-lehavre.fr

Abstract

The performance criteria generally used in real-time database systems (RTDBSs) are the transactions success ratio and the system quality of service. The main scheduling policy used for real-time transactions is earliest deadline first (EDF) where the shortest is the transaction deadline, the highest is its priority. With EDF, the successful transactions are not necessarily the most important transactions in the system. Moreover, it is well-known that EDF is not efficient in overload conditions. In this paper, we introduce the notion of transaction importance and present a new priority assignment technique based on both transactions importance and deadlines. This assignment policy leads to a new scheduling policy, called generalized earliest deadline first (GEDF). In order to show the benefits of using GEDF for managing real-time transactions, we have designed an RTDBS simulator and carried out Monte Carlo simulations.

1 Introduction

Real-time database systems (RTDBSs) must guarantee the transactions *ACID* (Atomicity, Consistency, Isolation, Durability) properties on one hand, and they must schedule the transactions in order to meet their individual deadlines, on the other hand [14]. An RTDBS can be considered as a combination of a traditional database system (DBS) and a real-time system (RTS).

Most performance studies in RTDBSs use EDF scheduling policy which is based on a priority assignment according to the deadlines, i.e. the earliest the transaction deadline is, the highest the priority is. However, such successful transactions are not necessarily the most important transactions in the system. Moreover, it is well-known that EDF is not efficient to schedule transactions (or tasks) in overload conditions, leading to the degradation of the system performances. This results from the assignment of high priorities to transactions that finally miss their deadlines. These high-priority transactions also waste system resources and delay other transactions [21]. To overcome these disadvantages, the study dealt with in [7] introduced an extension of EDF called adapted earliest deadline (AED). AED is a priority assignment policy which stabilizes the overload performance of EDF through an adaptive admission control mechanism in an RTDBS environment. In this method, the incoming transactions are assigned to either *hit* or *miss* group. Using a feedback mechanism, the capacity of the hit group is adjusted dynamically to improve the performances. Transactions in miss group only receive processing if the hit group is empty. In [13], Pang et al. proposed an extension of AED, called adaptive earliest virtual deadline (AEVD), to address the fairness issue in an overloaded system. In AEVD, virtual deadlines are computed based on both arrival times and deadlines. Since transactions with longer execution times will arrive earlier relative to their deadlines, AEVD can raise their priorities in a more rapid pace as their durations in the system increase. Consequently, longer transactions can exceed the priorities of shorter transactions that have earlier deadlines but later arrival times. The results of comparative performance study of AED and AEVD reported in [13] have established that AEVD provides better performances than AED. To resolve some weaknesses of AEVD, Datta et al. [4] have introduced priority based scheduling policy, called adaptive access parameter (AAP) method where they use explicit admission control. An other study done in [5] deals with the problem of repeatedly transactions processing in an RTDBS. In this work, Dogdu gave a number of priority assignment techniques based on the execution histories of real-time transactions that overcome the biased scheduling in favor of the short transactions when using EDF policy.

In this paper, we introduce a new approach based on a weight technique: a weight is assigned to a transaction according to the importance of its tasks. We also propose a new priority assignment technique which uses both the deadline and the transaction importance. This assignment

*This work is supported by grant $ACI\text{-}JC$ #1055

policy leads to a new scheduling policy, called *Generalized Earliest Deadline First* (GEDF) developed to overcome the weakness of EDF scheduling policy. GEDF may be considered as a generalization of EDF due to its flexibility and its adaptability to the system workload conditions (in normal conditions, GEDF behaves like EDF). To show the effectiveness of GEDF scheduling policy on RTDBS performances, we analyze the system performances according to the transactions success ratio and quality of service (QoS). To this purpose, Monte Carlo simulations are conducted on the RTDBS simulator we have developed. This simulator is based on components generally encountered in RTDBSs [10, 15, 14]. The results are compared to EDF scheduling technique under various execution constraints and conditions, such as the transactions arrival process, system load, conflicts level, concurrency control policy and database size.

The remainder of this paper is organized as follows. In Section 2, we describe the system model and some simulator components. Then, we present our weighted approach of transactions and the GEDF scheduling policy we propose. Section 3 is devoted to the Monte Carlo simulation experiments and the results we obtained. We then present the performance evaluation results of the GEDF scheduling policy. Finally, in Section 4, we conclude the paper and give some aspects of our future work.

2 Simulator and system model

We base our work on a system model dealt with in our previous work [16, 17], where some other real-time characteristics are added, e.g. temporal data, update transactions and the implementation of the freshness manager. We have also developed a new priority assignment policy where the importance criterion is added and on which is based the new scheduling approach, called GEDF. Note that we do not use an admission control mechanism (ACM) to reduce or to manage the submitted transactions according to the system workload. In our application, all transactions are accepted in the system. We think that it is more profitable to study the behavior of GEDF vs EDF without influence the system workload by using an ACM. The general mechanism of the simulator components is discussed briefly in subsection 2.6.

Due to decreasing of main memory cost and its relatively high performance [2, 18], main memory databases have been increasingly applied to real-time data management such as stock trading, e-commerce, and voice/data networking. In this work, we consider a main memory database model. The RTDBS is materialized by the simulator we developed and available on line [1]. In the following, we will focus on the components related to the data and transactions model and the new scheduling technique. Other components of the simulator are detailed in earlier papers [16, 17].

[1]The simulator is available on-line at the following URL : `http://litis.univ-lehavre.fr/~semghouni/`

2.1 Data and transactions

The database is composed of independent data objects classified into two classes: temporal data (TD) and non-temporal data (NTD). The state of a temporal data object may become invalid with the passage of time. Associated with its state, there is an absolute validity interval, denoted avi [20]. A Temporal data d_i is considered temporally inconsistent or stale if the current time is later than the timestamp of d_i (time of its last update) followed by the length of the absolute validity interval of d_i, i.e. $currenttime > timestamp_i + avi_i$. Here, we do not restrict the notion of temporal data to data provided by physical sensors. Instead, we consider a broad meaning of sensor data. Any item whose value reflects the time-varying real world status is a temporal data item [14], for example, information on the state of a deposit stock. A data whose state does not become invalid with the passage of time is a non-temporal data object [20].

We consider only firm real-time transactions and we classify them into update and user transactions. Update transactions are periodic and only write temporal data which capture the continuously state changing environment. We assume that an update transaction is responsible for updating a single temporal data item in the system. Each temporal data item is updated following a *more-less* approach where the period of an update transaction is assigned to be more than half of the validity interval of the temporal data [19].

We assume that user transactions can read or write non-temporal data and can only read temporal data [9]. The user transactions arrive in the system according to a Poisson process with an average rate λ. The number of operations generated for each user transaction is uniformly distributed in the user transaction size interval, denoted $User_{SInerval}$. Data accessed by the operations of the transaction are randomly generated and built according to the level of data conflicts (see subsection 2.4). Transactions execute "read" operations on data with a probability φ and "write" operations with a probability *(1-φ)*. For more details, see transaction characteristics in Table 1 page 5.

To distinguish the important transactions from the others, transactions are weighted according to their importance. The importance criterion is called *transaction system priority*, and is denoted by $SPriority$. In the following, we will describe fully how this importance criterion is assigned to each transaction.

2.2 Transactions system priorities (SPriority)

The transaction system priority (SPriority) is a parameter related to each transaction. It expresses the degree of importance of the task(s) executed by a transaction and defines its rank among all the transactions in the system. This parameter is assigned to each transaction when it is generated. The proposed GEDF scheduling policy (see section 2.3) uses this parameter in addition to the deadline to schedule transactions. We also consider the SPriority

as one of the criteria used to evaluate the system quality of service (see subsection 2.5).

In order to maintain temporal data consistency, i.e. to ensure that data in the database reflect the state of the environment, we consider that the rank of update transactions is higher than that of user transactions. The update transactions class is more important than that of the user transactions because one of the main design goals of RTDBSs is to both guarantee the temporal data freshness [14] and maintain the database consistency. Consequently, we divide the interval of SPriority values, i.e. $[0, MaxValue]$, into two intervals: the first interval $[0, N]$ is devoted to the SPriority values of update transactions and the second, $]N, MaxValue]$, is devoted to the SPriority values of user transactions. The value of N is application-dependent and is fixed by the system manager. In our model, we consider that the zero value of SPriority, i.e. SPriority=0, corresponds to the highest rank that a transaction can have in the system. To assign the SPriority value to each transaction, we use the following technique:

Transaction weight technique (WTec) We use two weight functions according to the transaction class to assign the SPriority value:

Update transactions class: We consider that temporal data items which are updated frequently, i.e. which have short update period, are data items that contain important information (for example, position of an aircraft). We relate the importance of a temporal data item to its update frequency because its absolute validity interval is also short (*more-less* [19] approach), which makes its update more critical.

Let *MaxPeriod* be the longest period among the periods of update transactions. The SPriority of an update transaction T is computed according to the following formula:

$$SPriority_{update} = N \times \frac{Period_T}{MaxPeriod} \quad (1)$$

User transactions class: The user transaction importance SPriority uses criteria based on both the transaction "write" set operations and the transaction "read" set operations. A user transaction T is assigned a SPriority value by the following formula:

$$SPriority_{user} = MaxValue - \gamma \times Weight_T - (1-\gamma) \times DBA_{Value} \quad (2)$$

where

- $Weight_T$ denotes the weight of the current user transaction and is given by

  $$Weight_T = (\sum_{i=1}^{n} Wread_{TD} + \sum_{j=1}^{m} Wwrite_{NTD} - \sum_{k=1}^{l} Wread_{NTD}) \quad (3)$$

  where

  – $Wread_{TD}$, $Wwrite_{NTD}$ and $Wread_{NTD}$ denote respectively the weight assigned to a *read operation* of a temporal data, the weight assigned to a *write operation* of a non-temporal data and the weight assigned to a *read operation* of a non-temporal data (see the transaction characteristics in Table 1, in page 5).

  – n, m, and l are the numbers of operations ("Read" TD, "Write" NTD and "Read" NTD) in each user transaction.

- $\gamma \in]0, 1]$ is a rational value assigned to the transaction weight in the SPriority formula (see Table 1, in page 5).

- DBA_{Value} is an uniform random variable whose values are between 0 and $(MaxValue - N)$. We recall that N is the value that divides the SPriority interval $[0, MaxValue]$ according to transactions class, i.e. $SPriority_{update} \in [0, N]$ and $SPriority_{user} \in]N, MaxValue]$.

- $Maximum(\gamma \times Weight_T - (1-\gamma) \times DBA_{Value}) \leq MaxValue - N$, because the user transactions $SPriority$ belongs to $]N, MaxValue]$.

Motivations: the choice of Formula 2 to assign the SPriority value to a user transaction is motivated by the following arguments:

- To favor the results obtained by transactions reading temporal data, we consider that their results are generally more important than those obtained by transactions which read non-temporal data. Thus, a transaction which will read many temporal data is assigned a higher rank than others (see Formula 3).

- To favor database freshness, we consider that write operations are more important than read operations, because their function is to refresh the database regularly. Thus, a transaction which will write many data items on the database is assigned a higher rank than others (see Formula 3).

- To reduce data access conflicts in database, we consider that transactions that execute many read operations on the database are transactions that can induce many data access conflicts when the database is in update state. Optimistic conflict resolution approach (OCC-Wait-50) creates long wait durations for conflicts resolution, whereas pessimistic conflict resolution approach (2PL-HP) induces many restarts and aborts. In both situations, the system is overloaded and its performances are degraded. In order to avoid those weaknesses, the rank of a transaction which reads many non-temporal data is decreased in the system (see Formula 3).

- The database administrator can influence a transaction priority by modifying its importance in the system. This interaction is modelized in Formula 2 by

DBA_{Value}. For example, it is used to give high priority to an urgent transaction or to give high priorities to transactions that need high services.

2.3 Transaction scheduler (TS)

In RTDBSs, transactions can be periodic with synchronous release times, periodic with asynchronous release times or aperiodic. EDF protocol is considered as the best policy among RTS scheduling policies that are adapted to RTDBSs to schedule transactions. EDF can fit any workload of periodic or non-periodic real-time transaction [11]. EDF effectiveness to schedule synchronous and asynchronous tasks was dealt with in [12, 3].

With EDF scheduling policy, transactions are scheduled according to their deadlines. However, the deadline criterion is not sufficient to express task(s) importance of a transaction in the system. Although EDF has been shown to improve the average success ratio of the transactions, it discriminates against longer transactions under overload conditions [7, 13].

In order to optimize the system quality of service and to schedule transactions according to both the importance criterion, i.e. SPriority, and the deadlines, we propose an adapted scheduling policy, called generalized earliest deadline first (GEDF), which is described in the following subsection.

2.3.1 GEDF Scheduling policy

GEDF is a dynamic scheduling policy where transactions are processed in an order determined by their priorities, i.e. the next transaction to run is the transaction with the highest priority in the active queue. The priority is assigned according to both the deadline which expresses the criticality of time and the SPriority (see subsection 2.2) which expresses the importance of the transaction. We consider that the zero value of the $Priority$, i.e. $Priority = 0$, corresponds to the highest priority in the system. Transaction T is assigned a priority by the formula:

$$Priority(T) = (1-a) \times Deadline(T) + a \times SPriority(T) \quad (4)$$

where $0 \leq a \leq 1$, (see Table 1) is the weight given to the $SPriority$ in the priority formula and is application-dependent. Note that if two transactions have the same priority, we use timestamp of the transaction for data conflicts resolution.

2.3.2 GEDF contributions

- Update transactions are assigned high priorities with GEDF, which guarantee both the temporal data freshness and the database consistency (see subsection 3.2).

- Important transactions are assigned high priorities. This gives them more chances to be scheduled and executed before their deadlines.

- Each transaction executes a group of operations (Read/Write). The operation group of a transaction can be seen as more or less important than operations groups of other transactions. EDF policy can not express this importance. Whereas, with GEDF policy, we can express both the criticality of time and the transactions importance in the priority assignment policy.

- By varying the parameter a, GEDF can be adapted to the system load in order to optimize its performances. This assertion will be explained deeply in subsection 3.4.

- GEDF can be seen as an extension of EDF scheduling policy. In fact, it is sufficient to initialize the SPriority of transactions with the same value or to initialize the weight parameter in priority formula to zero value, i.e. $a = 0$, then GEDF becomes an EDF scheduling policy. This property is used to preserve the EDF qualities, while avoiding its weakness (see subsection 3.4).

2.4 Conflicts level

Data conflicts result from the behavior of the transactions in the database. We assume that some data are more important than others and they are frequently requested by user transactions. In order to reproduce this behavior in the transaction action (read or write), we assign each data item a drawing probability in the following manner.

Let $r_1 < r_2 < \ldots < r_k < \ldots < r_n$, denote the ranking of the data items D_1, D_2, ..., D_k, ..., D_n respectively. We use a ranking function defined as follows: $r_i = i + 1$, where i is the index of data item D_i.

The probability of drawing the data item D_i is given by

$$Prob_{D_i} = \frac{r_i}{R},$$

where $R = \sum_{i=1}^{n} r_i$, is the sum of all ranks. Thus, data with high probabilities will be more drawn than those with low probabilities.

We select the data item D_k according to the above probabilities, i.e. we generate a uniform random variable $\mathcal{U}$ in $]0, 1[$ and select D_k if $\mathcal{U} \in]\sum_{i=1}^{k-1} Prob_{D_i}, \sum_{i=1}^{k} Prob_{D_i}]$, by convention $k = 1$ if $\mathcal{U} \in]0, Prob_{D_1}]$.

2.5 System performance metrics

2.5.1 Transaction success ratio (SRatio)

To assess the system performances, we consider transaction success ratio as the main metric. The success ratio is given by:

$$SRatio_{Type} = \frac{CommitT_{Type}}{SubmittedT_{Type}},$$

where $CommitT$ indicates the number of transactions committed by their deadlines, $SubmittedT$ indicates all

Database characteristics		
Notation	Definition	Values
λ	User transaction arrival rate.	0.6 to 2.4.
Time	Duration of one experiment.	1000 clock cycles.
Dsize	Number of data in the DB.	1000.
TD-size	Number of temporal data in the DB.	15% $\times$ Dsize, i.e. 150 data.
Min_avi, Max_avi	Minimal and maximal avi.	Min_avi=5 clock cycles, Max_avi=100 clock cycles.
Transaction characteristics		
Notation	Definition	Values
φ	Probability to execute a "Read" or a "Write" operation.	$\varphi(Read) = 2/3, \varphi(Write) = 1 - \varphi(Read)$
$User_{SInerval}$	User transaction size interval.	[5, 20] combined operations.
$Update_{size}$	Number of operations in an update transaction.	1 write operation.
D_{-UpT}	Deadline of update transaction (*more-less* approach).	$D_{-UpT} = \frac{1}{3} \times Avi.$
P_{-UpT}	Period of update transaction (*more-less* approach).	$P_{-UpT} = \frac{2}{3} \times Avi.$
Slack Function	Initialization of α and β.	$\alpha = 4$ and $\beta = 0.9$.
SPriority	Intervals of SPriority.	$SPriority_{Update} \in [0, 16]$ and $SPriority_{User} \in]16, 80]$.
γ	Initialization of γ.	$\gamma = 4/5$.
$Wread_{TD}$	Reading weight of one temporal data.	$Wread_{TD} = 1$.
$Wwrite_{NTD}$	Writing weight of one non-temporal data.	$Wwrite_{NTD} = 2$.
$Wread_{NTD}$	Reading weight of one non-temporal data.	$Wread_{NTD} = 1$.
System characteristics		
Notation	Definition	Values
Quantum	Execution capacity in one clock cycle.	20 Tasks/clock cycle
Task	Indivisible action.	one Read or Write operation.
ReadTime	Consumption of a read operation.	1 quantum unit.
WriteTime	Consumption of a write operation.	2 quantum units.
a	Initialization of the parameter a in (4) when using GEDF.	$a = 0, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}$ or $\frac{1}{8}$.
$\mathcal{R}$	Initialization of $\mathcal{R}$ in QoS function.	$\mathcal{R} = \frac{1}{20}$
SP	Scheduling policy.	"EDF", "GEDF".
CC	Concurrency Control protocol.	"2PL-HP" or "OCC-Wait-50".

Table 1. Simulation parameters.

submitted transactions to the system in the sampling period and $Type$ indicates the type of transactions class, i.e. $user$ or $update$.

2.5.2 System Quality of Service (QoS)

The QoS can be seen as a global metric which measures the amount of service provided by the system to the users. In this part, we define two parameters of the system QoS: (a) the success ratios of committed user and update transactions; (b) and the satisfaction degree (denoted: $SatDegree$) of the system on the important transactions, i.e. maximization of the commit of important transactions among the committed transactions.

As in the case of SRatio, the SatDegree is also specialized according to the class of transactions and is measured as follows:

$$SatDegree_{Type} = \frac{\sum_{i=1}^{CommitT_{Type}} Exp(-\mathcal{R} \times SPriority_i)}{\sum_{l=1}^{SubmittedT_{Type}} Exp(-\mathcal{R} \times SPriority_l)},$$

where $\mathcal{R}$ is a scale parameter and $Type$ indicates the type of transactions class, i.e. either $user$ or $update$. The construction of $SatDegree$ coefficient satisfies two facts:

- it takes values between 0 and 1, and
- it increases from 0 to 1 according to the number of the committed transactions with smaller $SPriority$.

It follows that in order to maximize the system QoS, we have to maximize the quadruplet $(SRatio_{user}, SatDegree_{user}, SRatio_{update}, SatDegree_{update})$.

2.6 General mechanism of the simulator

User transactions are submitted to the system following a Poisson process with an average rate λ into the active queue. The *deadline controller (DC)* supervises the transactions deadlines, and informs the *transaction scheduler (TS)* if a transaction misses its deadline in order to abort it. The *freshness manager (FM)* exploits the absolute validity interval (*avi*) to check the freshness of a data item before a user transaction accesses it and blocks all user transactions reading stale temporal data. Transactions data conflicts are resolved by the *Concurrency controller (CC)* according to transactions priorities. *CC* informs *TS* in the following cases: (a) when a transaction is finished (committed) and its results are validated in the database, (b) when a transaction is blocked waiting for a conflict resolution, (c) when a transaction is restarted, following the commit of other transactions, (d) when a transaction is rejected because its restart is impossible, i.e. its best execution time is higher than its deadline minus the current time ($BET_T > DT - currenttime$), (e) or when a transaction is transfered from the blocked queue to the active queue, i.e. its data conflicts are resolved.

3 Simulations and results

3.1 Simuations parameters

To assess the performances of GEDF scheduling policy in comparison to EDF scheduling policy, we carried out Monte Carlo simulations. This allows us to study the transactions success ratio behavior and the system quality of service. Given the system parameters of Table 1, we repeat the experiment 1000 times in each simulation in order to obtain a sample of 1000 values for the performances,

i.e. SRatio and QoS. Each point shown in Figures 1, 2 and 5 (success ratio) and Figures 3 and 6 (quality of service) represents the computed average of performance results deduced from each simulation sample.

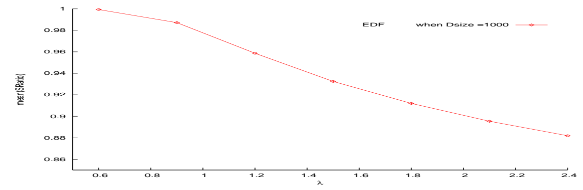
The workload of the system is varied according to both the database size and the arrival rate λ of user transactions. When $\lambda = 0.6$, the number of user transactions arriving to the system during one experiment is about 600. When $\lambda = 2.4$, this number is about 2400. The database workload is related to the number of temporal data (TD) in the database and increases substantially when TD increases. In our simulations, the number of temporal data represents 15% of the database size, which leads to 7000 to 12000 update transactions in one experiment.

The absolute validity interval (avi) of each temporal data is randomly generated from the interval [Min_avi, Max_avi], i.e. [5,100]. Min_avi is fixed to 5 clock cycles and Max_avi is fixed to 100 clock cycles in order to have enough workload of temporal data (minimum 15 updates and maximum 335 updates in the experiment duration). Each update transaction is assigned a period and a deadline according to the avi of the temporal data it accesses (for more details, see *more-less* approach [19]).

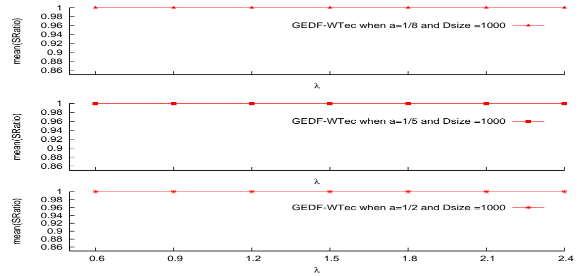
The parameters α and β of the slack function SF (see [16, 17]) are assigned the values 4 and 0.9 in order to obtain an average behavior of transactions load in the system. The probability to execute a read operation is assigned a value of $\frac{2}{3}$, and a write operation probability is $\frac{1}{3}$. We assume that a write operation requires two quantum units for execution and a read operation requires one quantum unit. The parameter γ is assigned the value $\frac{4}{5} = 0.8$ in order to minimize the effect of the DBA_{Value}, i.e. database administrator interaction, in the SPriority (Formula 2). In order to show the influence of the $SPriority$ weight on the GEDF behavior and on the system performances, we varied the value of the parameter a in Formula 4 page 4. The assigned values used in simulations are $a = 0, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}$ or $\frac{1}{8}$. This variation allows to deduce the appropriate assigned value to the parameter a according to the system workload (see subsection 3.4).

For the simulations we have implemented two main concurrency controllers: 2PL-HP, a pessimistic protocol, where a low priority transaction is aborted and restarted upon a conflict to avoid priority inversion and deadlock problems [1], and OCC-Wait-50, an optimistic protocol [6, 8], which incorporates a wait control mechanism in the classical OCC. This mechanism monitors transaction conflicts states and dynamically decides when, and for how long, a low priority transaction should wait for its conflicting higher priority transactions to complete.

In the following subsection, we introduce the discussions of simulation results while comparing EDF and GEDF using the defined *weight technique* to assign SPriority of transactions. We also introduce a discussion on GEDF flexibility and how it is possible to exploit this flexibility in order to improve the system performances. Due



(a) Success ratio of update transactions when using EDF.



(b) Success ratio of update transactions when using GEDF with $a = \frac{1}{8}$, $a = \frac{1}{5}$ and $a = \frac{1}{2}$.

Figure 1. Influence of scheduling policy on update transactions with 2PL-HP protocol.

to the lack of space, we discuss in the following only the influence of scheduling policy when using 2PL-HP protocol, and the system QoS when using OCC-Wait-50 protocol. We can argue that the simulations results revealed that the two protocols have a similar behavior on the system performances when using either EDF or GEDF scheduling policy.

3.2 Influence of the scheduling policy

In order to analyze the influence of the scheduling policy on the success ratio performances, we compare the results obtained under EDF and GEDF when using 2PL-HP protocol and when varying the system workload. Figures 1 and 2 illustrate graphically this comparison.

The best performances for update transactions are obtained with GEDF scheduling policy (Figure 1(b)): the success ratio is maximal, i.e. 100%. We can see also in Figure 1(b) that for all variations of $a > 0$, i.e. SPriority weight, we obtain the same performances on the update success ratio results for all system workload conditions. We can conclude that when increasing the user transactions number, there is no effect on the update transactions performances. This result may be explained by the higher priority given to update transactions which ensures their processing before user transactions. When we look at the performances with EDF scheduling policy (Figure 1(a)), we notice a progressive decreasing of the success ratio when the workload progressively becomes heavy.

With EDF, there is no difference between the two classes of transactions, since only the deadline is taken into account. Thus, user transactions can be scheduled prior to update transactions if their deadlines are imminent, which affects and decreases the success ratio of update transactions and degrades the temporal data con-

sistency in the database. This affects considerably the success ratio of user transactions, especially when the system workload is heavy. In the following, we comment the user transactions performances on three intervals: $\lambda \in [0.6, 1.2[$ (light workload to average workload), $\lambda \in [1.2, 1.5[$ (average workload) and $\lambda \in]1.5, 2.4]$ (high workload).

When we consider the values of λ in the interval $[0.6, 1.2[$ (see Figure 2(a)), we notice that when the system is not overloaded, EDF gives better performances on user transactions SRatio than GEDF with all variations of the parameter a. Indeed, when using GEDF scheduling policy, the lower priority transactions must wait for the commit of the higher priority transactions to be executed even if their deadlines are imminent. This has a negative effect when the system workload is light, which reduces the chances of lower priority transactions to commit, i.e. the user transaction success ratio decreases.

When the value of λ is in the interval $[1.2, 1.5[$, i.e. average workload (see Figure 2(b)), we can see that GEDF provides better results than EDF according to the variations of the SPriority weight parameter. The reversible points corresponding to those situations can be seen respectively in Figure 2(b): $\lambda \simeq 1.2$ for GEDF with $a = \frac{1}{8}$, $\lambda \simeq 1.25$ for GEDF with $a = \frac{1}{7}$, $\lambda \simeq 1.28$ for GEDF with $a = \frac{1}{6}$, $\lambda \simeq 1.32$ for GEDF with $a = \frac{1}{5}$, $\lambda \simeq 1.38$ for GEDF with $a = \frac{1}{4}$, $\lambda \simeq 1.45$ for GEDF with $a = \frac{1}{3}$, and $\lambda \simeq 1.48$ for GEDF with $a = \frac{1}{2}$. We will see in subsection 3.4 how the reversible points can be used to enhance the system performances in this interval.

When the system workload is heavy, i.e. $\lambda \in]1.5, 2.4]$, the situation is reversed completely in favor of GEDF that provides better performances than EDF (for example $SRatio(GEDF(a = \frac{1}{2})) \simeq SRatio(EDF) + 10\%$ when $\lambda = 2.4$) with all values assigned to the parameter a. We also deduce that when the workload increases, the improvement of the system performances is correlated with the increasing of the value assigned to the parameter a. The results obtained by GEDF can be explained by the fact that the temporal data consistency is more safeguarded with GEDF than with EDF policy (see the success ratio of update transactions in Figure 1(b)). With GEDF, the waiting time of fresh data is reduced thanks to the success ratio of update transactions, which is maximal, i.e. 100%. This gives to the user transactions reading temporal data the maximum chances to meet their deadlines, decreasing then the system load. Moreover, only the important transactions are scheduled in the system. When the system workload is heavy, GEDF scheduling policy reduces the useless aborts and restarts, that are inherent to EDF scheduling policy, i.e. transactions that are aborted and restarted by other transactions which finally miss their deadlines.

3.3 System Quality of Service (QoS)

In the following, we discuss and compare the system quality of service (QoS) registered under EDF and GEDF when using OCC-Wait-50 protocol. Figures 3(a) and 3(b) illustrate graphically this comparison. When we look at the QoS given on update transactions (Figure 3(a)), we deduce that all variants of GEDF give the optimal performances[2] on $SRatio$ and $SatDegree$, i.e $(SRatio_{update}, SatDegree_{update}) = (1, 1)$. We can conclude that GEDF scheduling policy is better than EDF and gives high QoS on update transactions in all system workload conditions. Thus GEDF scheduling policy maintains the temporal data consistency in all workload conditions.

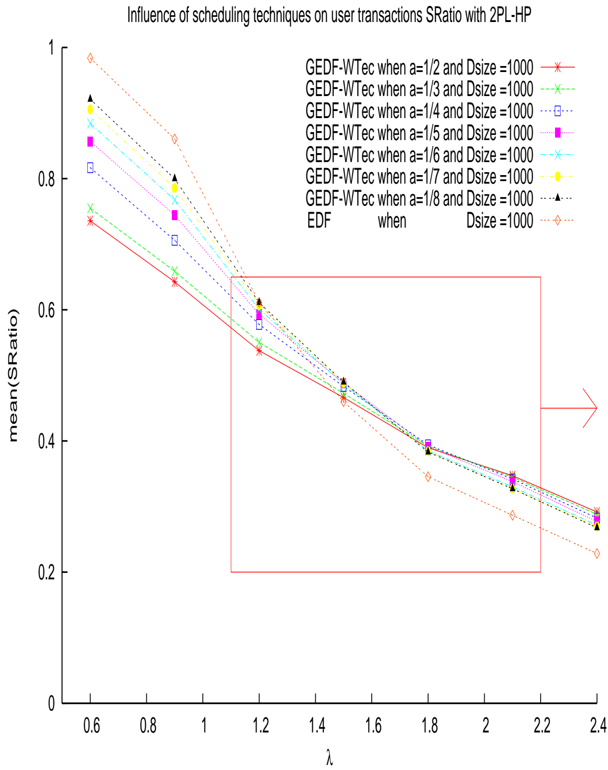
When we consider QoS on user transactions (Figure 3(b)), the values of $(SRatio_{user}, SatDegree_{user})$ given by GEDF scheduling policy are related to the variation of the parameter a, i.e. the SPriority weight. In the following, we comment the QoS of user transactions in three intervals $\lambda \in [0.6, 1.1[$, $\lambda \in [1.1, 1.5[$ and $\lambda \in [1.5, 2.4]$ where we illustrate respectively the three system situations: light workload, average workload, and high workload.

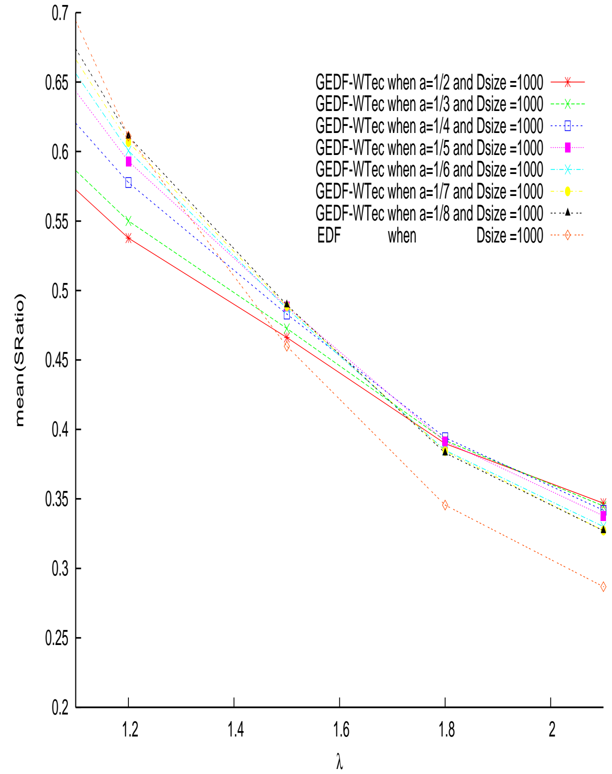
When λ is in the interval $[0.6, 1.1[$, EDF gives the best QoS on user transactions, i.e. $QoS_{EDF}\ (SRatio_{user}, SatDegree_{user}) >$ $QoS_{GEDF}(SRatio_{user}, SatDegree_{user})$ with all variations of the parameter a. This can be explained by the high SRatio registered with EDF when the system workload is light (as detailed in subsection 3.2) which gives high SatDegree on the committed user transactions.

When λ is in the interval $[1.1, 1.5[$, we note that EDF *SRatio* is higher than GEDF *SRatio* before the reversible points deduced in subsection 3.2. The situation is reversed in favor of GEDF with *SatDegree* according to the value assigned to the parameter a. The reversible points can be seen respectively in Figure 3(b): $\lambda = 1.1$ for GEDF with $a = \frac{1}{8}$, $\lambda = 1.15$ for GEDF with $a = \frac{1}{5}$, and $\lambda = 1.3$ for GEDF with $a = \frac{1}{2}$. These cases indicate that the number of committing important transactions with GEDF is higher than that of EDF. When we combine the results with the reversible points deduced in subsection 3.2 we can argue that GEDF gives a better QoS than EDF. We can see these cases, for example, in Figure 3(b), in the intervals $\lambda \in [1.25, 1.5[$ when $a = \frac{1}{8}$, $\lambda \in [1.38, 1.5[$ when $a = \frac{1}{5}$, and $\lambda \in [1.48, 1.5[$ when $a = \frac{1}{2}$. With GEDF, the importance criterion (SPriority) of a transaction influences its scheduling order. This gives the important transactions the best chances to commit before their deadlines.

When $\lambda \geq 1.5$, GEDF variants give the best QoS on user transactions, i.e. $QoS_{GEDF}\ (SRatio_{user}, SatDegree_{user}) > QoS_{EDF}$ $(SRatio_{user}, SatDegree_{user})$ (see Figure 3(b)). This can be explained by the best capacity of GEDF

[2] In Figure 3(a), the SRatio and SatDegree of update transactions when using GEDF is maximal, i.e. 100% in all conditions and for all $a > 0$.

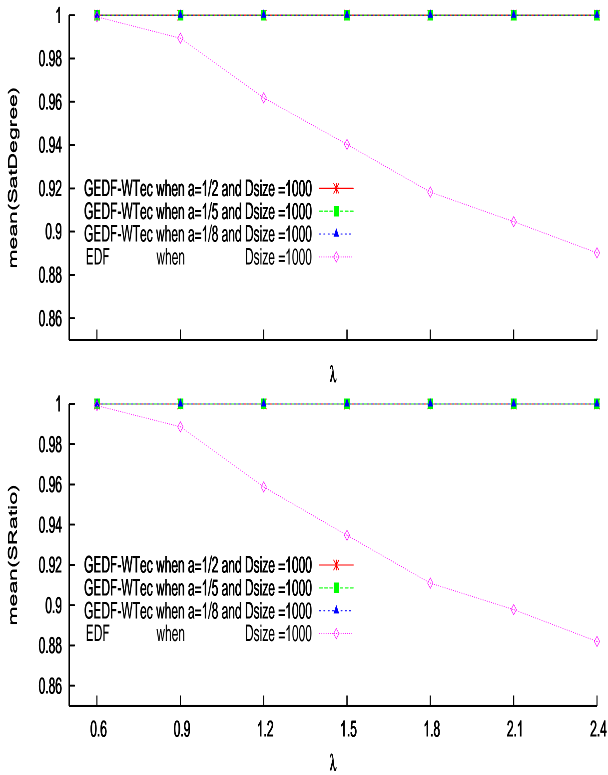


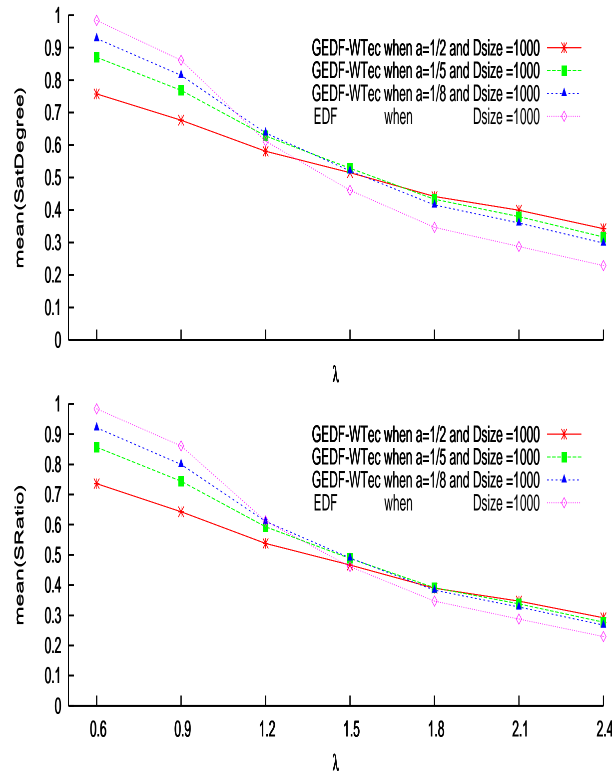


(a) Comparison of success ratio of user transactions between GEDF and EDF scheduler.

(b) Zoom in the square zone of Figure 2(a).

Figure 2. Scheduling policy influence on user transactions with 2PL-HP protocol.





(a) System QoS on update transactions with OCC-Wait-50

(b) System QoS on user transactions with OCC-Wait-50

Figure 3. Comparison of system QoS between EDF and GEDF scheduling policies.

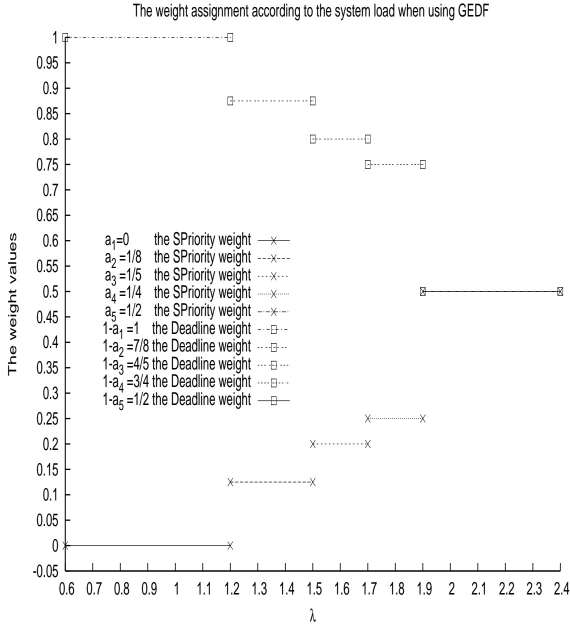
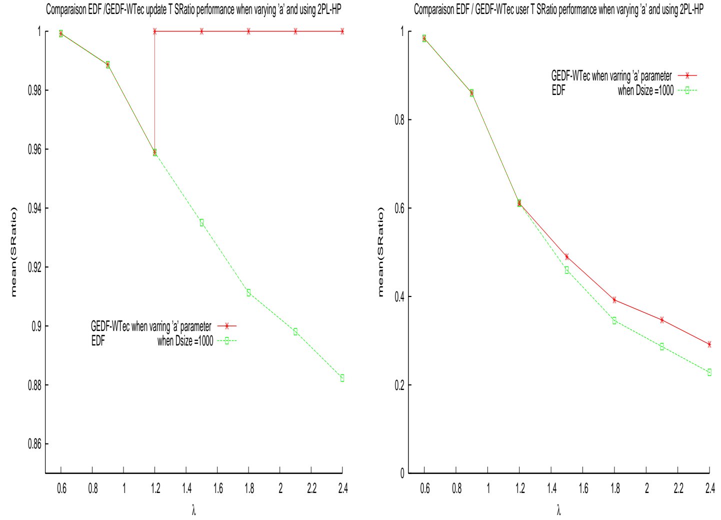


Figure 4. The weight assignment to the SPriority and Deadline according to the system load when using GEDF.



(a) The influence of the flexibility of GEDF on update transactions with 2PL-HP protocol.

(b) The influence of the flexibility of GEDF on user transactions with 2PL-HP protocol.

Figure 5. Comparison between EDF and flexible GEDF transactions SRatio performances.

to schedule transactions when the workload is heavy, which ensures a better SRatio (see subsection 3.2). We can also explain this result by the GEDF capacity to succeed in the commit of important transactions than EDF.

3.4 GEDF flexibility according to the system workload

In this subsection, we show the GEDF flexibility and its capacity to allow the system manager to interact and to adapt to different system workload situations. To this purpose, we exploit the GEDF variants to enhance the system performances.

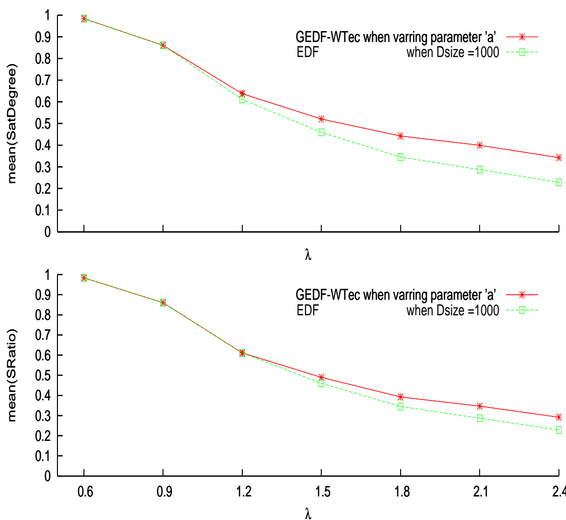


Figure 6. Comparison between the registered QoS when using EDF/flexible GEDF with 2PL-HP.

Our objective is to obtain the optimal output of the system on user transaction SRatio and QoS. To this purpose, we have deduced by simulations the adequate values of the parameter a according to the system workload intervals. Figure 4 shows the weight values assigned to the SPriority and deadline which we use in GEDF priority assignment formula according to the system workload intervals. In Figure 4, we can see for example when λ is between 0.6 and 1.2, the assigned value to a is 0 and when λ is between 1.5 and 1.7 the assigned value to a is $\frac{1}{5}$. The intervals of the parameter a values are deduced by simulation in order to have the best results with GEDF.

A comparison of success ratio obtained by GEDF scheduling policy when varying the parameter a according to system load (see Figure 4) and EDF scheduling policy are summarized in Figure 5. The curves in Figure 5(b) show the effectiveness of the GEDF scheduling policy to provide a good output of the system, i.e. we obtain the best success ratio of user transactions for all experimental conditions, unlike with EDF. When $\lambda < 1.2$, the parameter a is assigned zero value, i.e. GEDF becomes an EDF scheduling policy and allows to exploit the EDF characteristics when the system is not overloaded. When $\lambda \geq 1.2$ and according to the workload intervals, we use the adequate values for the parameter a. This allows GEDF to give the best performances on user transactions success ratio. In addition, we can see the influence of the parameter a on update transactions (SRatio). When $a = 0$, the SRatio decreases up to $\lambda = 1.2$ and when $a > 0$, the SRatio becomes optimal, i.e. 100% (see Figure 5(a)).

The related QoS results obtained on the user transactions when exploiting the GEDF flexibility in compari-

son to the EDF scheduling policy are summarized in Figure 6. We can argue that GEDF provides better QoS on user transactions in all system conditions workload, and it optimizes not only the $SRatio$ but also the satisfaction degree, i.e. $SatDegree$, on the user transactions in all workload conditions (note that $SatDegree(GEDF) \simeq SatDegree(EDF) + 15\%$, when $\lambda = 2.4$).

4 Conclusion

In this paper, we have proposed a weighted approach which expresses the transactions tasks importance and a new scheduling policy (GEDF) to improve the system performances in firm RTDBS. The GEDF scheduling policy uses the deadline and the importance criterion to schedule transactions. The impact of the GEDF scheduling policy on RTDBS performances, i.e. transaction success ratio and system QoS, is studied under different system workload situations and when using different concurrency control protocols. The study is done in comparison with EDF scheduling policy performances. We have also discussed the GEDF scheduling policy results and have shown its flexibility according to the system workload and its effectiveness to improve the transactions success ratio and system QoS in firm RTDBS.

In our future work, we plan to deduce the function that expresses the value of the parameter a in Formula 4 according to the system workload in order to give GEDF scheduling policy a better flexibility. We also project to extend our study to other scheduling policies and concurrency control protocols to compare their performances with the results obtained in this paper.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, September 1992.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in soft real-time database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 245–256, 1995.

[3] E. Coffman. *Introduction to deterministic scheduling theory.* Computer and Job-Shop Scheduling Theory, Wiley and Sons, New York, 1976.

[4] A. Datta, S. Mukherjee, P. Konana, I. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Technical report TR-11, Time Center.*, March 1996.

[5] E. Dogdu. Utilization of execution histories in scheduling real-time database transactions. *Data and Knowledge Engineering*, 57(2):148–178, May 2006.

[6] R. Haritsa, M. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *11th IEEE Real-Time Systems Symposium (RTSS)*, pages 94–103, 1990.

[7] R. Haritsa, M. Carey, and M. Livny. Earliest deadline scheduling for real-time database systems. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 232–243, 1991.

[8] R. Haritsa, M. Carey, and M. Livny. Data access scheduling in firm real-time database systems. *Real-Time Systems journal*, 4(3):203–241, 1992.

[9] K. Kang, S. Son, and J. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transaction on Knowledge and Data Engineering*, 16(10):1200–1216, 2004.

[10] Y. Kim and H. Son. Supporting predictability in real-time database systems. In *Proceeding of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 38–48, 1996.

[11] T. Kuo and K. Lam. Real-time database systems: An overview of system characteristics and issues. In *Real-Time Database Systems*, pages 3–8, 2001.

[12] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, (20):46–61, 1973.

[13] H. Pang, M. Livny, and M. Carey. Transaction scheduling in multiclass real-time database systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 23–34, 1992.

[14] K. Ramamritham, H. Son, and L. Dipippo. Real-time databases and data services. *Real-Time Systems Journal*, 28:179–215, 2004.

[15] K. Ramamritham, M. Xiong, R. Sivasankaran, J. Stankovic, and D. Towsley. Integrating temporal, real-time and active databases. *ACM SIGMOD Record. Special Issue on RTDBS*, 25(1):8–12, March 1996.

[16] S. Semghouni, B. Sadeg, A. Berred, and L. Amanton. Statistical model for real-time DBMS performances. In *Proceedings of International Mediterranean Modeling Multiconference I3M'05, Conceptual Modeling and Simulation Conference (CMS)*, Marseilles, France, October 20-22, 2005.

[17] S. Semghouni, B. Sadeg, A. Berred, and L. Amanton. Probability density function of firm real-time transactions success ratio. In *Proceedings of the 8th Brazilian Workshop on Real-Time Systems (WTR2006)*, Curitiba, Brazil, June 2, 2006.

[18] T.-T. Team. In-memory data management for consumer transactions The Times-Ten approach. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 528–529, USA, June 1-3 1999.

[19] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. *IEEE Transactions on Computers*, 53(1):567–583, May 2004.

[20] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. *Scheduling Access to Temporal Data in Real-Time Databases.* Real-Time Database Systems: Issues and Applications, Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed, Kluwer Academic Publishers, 1997.

[21] P. Yu, K. Wu, K. Lin, and S. Son. On real-time databases: Concurrency control and scheduling. In *Proceedings of the IEEE, Special Issue on Real-Time Systems.*, pages 82(1):140–157, 1994.

Scheduling and control

Comparative Assessment and Evaluation of Jitter Control Methods

Giorgio Buttazzo
Scuola Superiore S. Anna
Pisa, Italy
giorgio@sssup.it

Anton Cervin
Lund University
Lund, Sweden
anton@control.lth.se

Abstract

Most control systems involve the execution of periodic activities, which are automatically activated by the operating system at the specified rates. When the application consists of many concurrent tasks, each control activity may experience delay and jitter, which depend on several factors, including the scheduling algorithm running in the kernel, the overall workload, the task parameters, and the task interactions. If not properly taken into account, delays and jitter may degrade the performance of the system and even jeopardize its stability.

In this paper, we evaluate three methodologies for reducing the jitter in control tasks: the first one consists of forcing the execution of inputs and outputs at the period boundaries, so trading jitter with delay; the second method reduces jitter and delay by assigning tasks shorter deadlines; whereas, the third method relies on non preemptive execution. We compare these techniques by illustrating examples, pointing out advantages and disadvantages, and evaluating their effects in control applications by simulation. It is found that the deadline advancement method gives the better control performance for most configurations.

1 Introduction

Real-time control applications typically involve the execution of periodic activities to perform data sampling, sensory processing, control, action planning, and actuation. Although not strictly necessary, periodic execution simplifies the design of control algorithms and allows using standard control theory to guarantee system stability and performance requirements. In a computer controlled system, periodic activities are enforced by the operating system, which automatically activates each control task at the specified rate.

Nevertheless, when the system involves the execution of many concurrent tasks, each activity may experience delay and jitter, which depend on several factors, including the scheduling algorithm running in the kernel, the overall workload, the task parameters, and the task interactions. If not properly taken into account, delay and jitter may degrade the performance of the system and even jeopardize its stability [18, 20, 12].

The problem of jitter in real-time control applications has received increased attention during the last decade and several techniques have been proposed to cope with it. Nilsson [24] analyzed the stability and performance of real-time control systems with random delays and derived an optimal, jitter-compensating controller. Martí *et al.* [23] proposed a compensation technique for controllers based on the pole placement design method. Di Natale and Stankovic [13] proposed the use of simulated annealing to find the optimal configuration of task offsets that minimizes jitter, according to some user defined cost function. Cervin *et al.* [10] presented a method for finding an upper bound of the input-output jitter of each task by estimating the worst-case and the best-case response time under EDF scheduling [22], but no method is provided to reduce the jitter by shortening task deadlines. Rather, the concept of *jitter margin* is introduced to simplify the analysis of control systems and guarantee their stability when certain conditions on jitter are satisfied.

Another way of reducing jitter and delay is to limit the execution interval of each task by setting a suitable relative deadline. Working on this line, Baruah at al. [5] proposed two methods for assigning shorter relative deadlines to tasks and guaranteeing the schedulability of the task set. Shin *et al.* [25] presented a method for computing the minimum deadline of a newly arrived task, assuming the existing task set is feasibly schedulable by EDF. Buttazzo and Sensini [8] also presented an on-line algorithm to compute the minimum deadline to be assigned to a new incoming task in order to guarantee feasibility under EDF.

Another common practice to reduce jitter in control applications is to separate each control task into three distinct subtasks performing data input, processing, and control output [11]. The input-output jitter is reduced by postponing the input-output subtasks to some later point in time, so trading jitter with delay. While it has been shown that task splitting in general may improve the schedulability of a task set [16], the method also introduces a number of problems that have not been deeply investigated in the real-

time literature. Finally, another possible technique to reduce scheduling-induced jitter is to execute the application in a non preemptive fashion.

In general, none of the techniques above can reduce the jitter all the way down to zero. There will always be some variability in the execution time of the (sub)tasks themselves, and there might be additional jitter caused by poor timer resolution, tick scheduling, non-preemptable kernel calls, etc. In this paper, however, we will focus on scheduling-induced jitter, and we will consider standard control algorithms that can be assumed to have near-constant computation times.

Although the techniques above have often been used in control applications, a comprehensive assessment and a comparative evaluation of their effect on control performance is still missing. In this paper, we provide a systematic description of these techniques, illustrating examples and pointing out possible problems introduced by each method. Then, we discuss a number of simulation experiments aimed at evaluating the effect of these approaches in control applications.

The rest of the paper is organized as follows. Section 2 presents the system model and the basic assumptions. Section 3 provides a definition of jitter and identifies the possible causes. Section 4 introduces the three approaches considered in this work for jitter reduction and discusses pros and cons of the methods. Section 5 describes some experimental results carried out to evaluate the three approaches. Section 6 states our conclusions and future work.

2 Terminology and Assumptions

We consider a set $\mathscr{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$ of periodic tasks that have to be executed on a uniprocessor system. Each periodic task τ_i consists of an infinite sequence of task instances, or jobs, having the same worst-case execution time (WCET), the same relative deadline, and the same interarrival period. The following notation is used throughout the paper:

$\tau_{i,k}$ denotes the k-th job of task τ_i, with $k \in \mathscr{N}$.

C_i denotes the worst-case execution time (WCET) of task τ_i, that is, the WCET of each job of τ_i.

T_i denotes the period of task τ_i, or worst-case minimum inter-arrival time.

D_i denotes the relative deadline of task τ_i, that is, the maximum finishing time allowed for any job, relative to its activation time.

$r_{i,k}$ denotes the release time of job $\tau_{i,k}$. If the first job is released at time $r_{i,1} = \Phi_i$, also referred to as the task phase or the offset, the generic k-th job is released at time $r_{i,k} = \Phi_i + (k-1)T_i$.

$s_{i,k}$ denotes the start time of job $\tau_{i,k}$.

$f_{i,k}$ denotes the finishing time of job $\tau_{i,k}$.

$R_{i,k}$ denotes the response time of job $\tau_{i,k}$, that is, the difference of its finishing time and its release time ($R_{i,k} = f_{i,k} - r_{i,k}$).

$INL_{i,k}$ denotes the input latency of a control job $\tau_{i,k}$, that is, the interval between the release of the task and the reading of the input signal. If the input is performed at the beginning of the job, then $INL_{i,k} = s_{i,k} - r_{i,k}$.

$IOL_{i,k}$ denotes the input-output latency of a control job $\tau_{i,k}$, that is, the interval between the reading of the input and the writing of the output. If the input is performed at the beginning of the job and the output at the end, then $IOL_{i,k} = f_{i,k} - s_{i,k}$.

U_i denotes the utilization of task τ_i, that is, the fraction of CPU time used by τ_i ($U_i = C_i/T_i$).

U denotes the total utilization of the task set, that is, the sum of all tasks utilizations ($U_i = \sum_{i=1}^{n} U_i$).

We assume all tasks are fully preemptive, although some of them can be executed in a non preemptive fashion. Moreover, we allow relative deadlines to be less than or equal to periods.

3 Jitter characterization

Due to the presence of other concurrent tasks that compete for the processor, a task may evolve in different ways from instance to instance; that is, the instructions that compose a job can be executed at different times, relative to the release time, within different jobs. The maximum time variation (relative to the release time) in the occurrence of a particular event in different instances of a task defines the jitter for that event. The jitter of an event of a task τ_i is said to be *relative* if the variation refers to two consecutive instances of τ_i, and *absolute* if it is computed as the maximum variation with respect to all the instances.

For example, the response time jitter (RTJ) of a task is the maximum time variation between the response times of the various jobs. If $R_{i,k}$ denotes the response time of the kth job of task τ_i, then the relative response time jitter of task τ_i is defined as

$$\mathrm{RTJ}_i^{rel} = \max_k |R_{i,k+1} - R_{i,k}| \tag{1}$$

whereas the absolute response time jitter of task τ_i is defined as

$$\mathrm{RTJ}_i^{abs} = \max_k R_{i,k} - \min_k R_{i,k}. \tag{2}$$

Of particular interest for control applications are the time instants at which inputs are read and outputs are written. An overview of control task timing is given in Figure 1. The time interval between the release of the task and the reading of the input signal $y_i(t)$ is called the input latency and is denoted by $\mathrm{INL}_{i,k}$. The interval between the reading of the

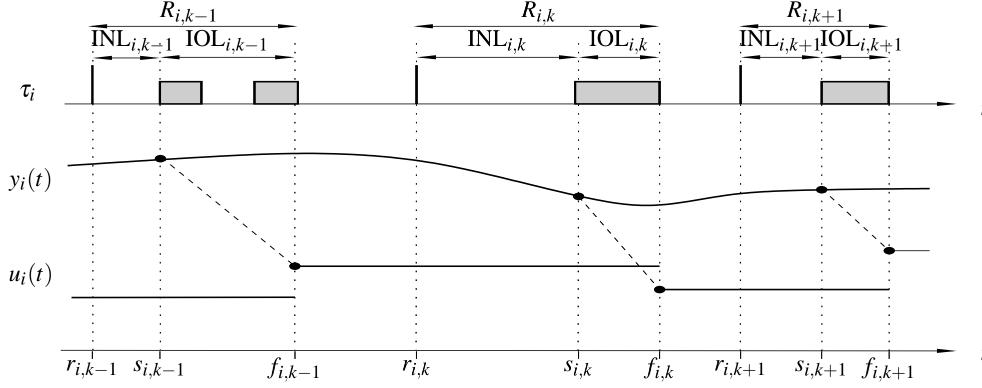


Figure 1. Control task timing.

input and the writing of the output is called the input-output latency and is denoted by $\mathrm{IOL}_{i,k}$.

In the figure, it is assumed that the input signal $y_i(t)$ is sampled at the start time $s_{i,k}$, and that the control signal $u_i(t)$ is updated at the finishing time $f_{i,k}$. Under this assumption, the response-time jitter is equivalent to the output jitter.

Similarly, the input jitter (INJ) of a task is the maximum time variation of the instants at which the input is performed in the various jobs. Thus, the relative input jitter of task τ_i can be defined as

$$\mathrm{INJ}_i^{rel} = \max_k |\mathrm{INL}_{i,k+1} - \mathrm{INL}_{i,k}| \tag{3}$$

whereas the absolute input jitter of task τ_i is defined as

$$\mathrm{INJ}_i^{abs} = \max_k \mathrm{INL}_{i,k} - \min_k \mathrm{INL}_{i,k}. \tag{4}$$

Another type of jitter of interest in control applications is the input-output jitter (IOJ), that is, the maximum time variation of the interval between the reading of the input and the writing of the output. The relative input-output jitter of task τ_i is defined as

$$\mathrm{IOJ}_i^{rel} = \max_k |\mathrm{IOL}_{i,k+1} - \mathrm{IOL}_{i,k}| \tag{5}$$

whereas the absolute input-output jitter of task τ_i is defined as

$$\mathrm{IOJ}_i^{abs} = \max_k \mathrm{IOL}_{i,k} - \min_k \mathrm{IOL}_{i,k}. \tag{6}$$

The jitter experienced by a task depends on several factors, including the scheduling algorithm running in the kernel, the overall workload, the task parameters, and the task interactions through shared resources.

The example shown in Figure 2 illustrates how the jitter is affected by the scheduling algorithm. The task set consists of three periodic tasks with computation times $C_1 = 2$, $C_2 = 3$, $C_3 = 2$, and periods $T_1 = 6$, $T_2 = 8$, $T_3 = 12$. Notice that, if the task set is scheduled by the Rate Monotonic (RM) algorithm (Figure 2a), the three tasks experience a response time jitter (both relative and absolute) equal to 0, 2, and 8, respectively. Under the Earliest Deadline First (EDF) algorithm (Figure 2b), the same tasks experience a response time jitter (both relative and absolute) equal to 1, 2, and 3, respectively. Also the input-output jitter changes with the scheduling algorithm; in fact, under RM, the three tasks have an input-output jitter (both relative and absolute) equal to 0, 2, 5, respectively, whereas under EDF the input-output jitter is zero for all the tasks. In general, EDF significantly reduces the jitter experienced by tasks with long period by slightly increasing the one of tasks with shorter period. A more detailed evaluation of these two scheduling algorithms for different scenarios can be found in [9].

Using the same example shown in Figure 2, it is easy to see that, if task τ_3 has a shorter computation time (e.g., $C_3 = 1$), the response time jitter experienced by τ_3 under RM decreases from 8 to 3, while the input-output jitter becomes 0. Hence, the jitter is heavily dependent on the workload, especially for fixed priority assignments. Also note the dependency on the task set parameters. In fact, by slightly increasing the period T_3, under EDF, the first job of τ_3 would be preempted by the second job of τ_1, so the jitter of τ_3 would increase significantly. It is also clear that the task offsets have an influence on the jitter. Finally, more complex interferences may occur in the presence of shared resources, which can introduce additional blocking times that may increase the jitter.

4 Jitter control methods

We now introduce three common techniques typically adopted to reduce jitter in real-time control systems. They are described in the following sections.

4.1 Reducing jitter by task splitting

The first approach exploits the fact that most control activities have a common structure, including an input phase, where sensory data acquisition is performed, a processing phase, where the control law is computed, and an output

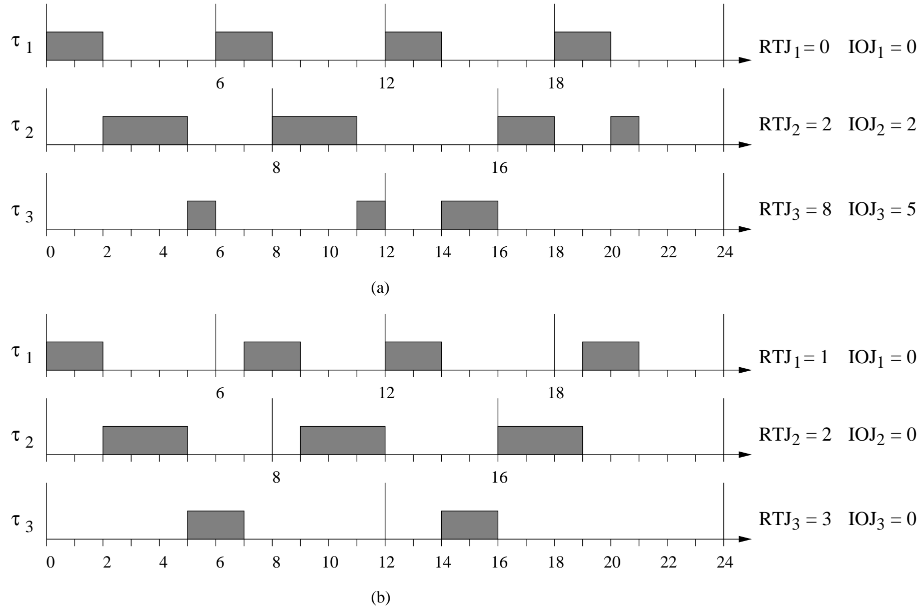


Figure 2. Jitter under RM (a) and EDF (b).

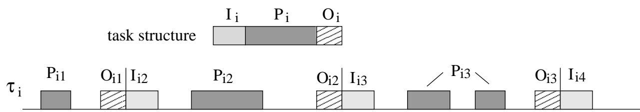


Figure 3. Task input and output can be forced to occur at period boundaries.

phase, where the proper control actions are sent to the controlled plant. Hence, each control task τ_i is divided into three subtasks: input, processing, and output. The computation times of these three subtasks will be denoted as I_i, P_i, and O_i, respectively, and the total computation time is given by $C_i = I_i + P_i + O_i$.

The key idea behind this method is to force the input subtask to be executed at the beginning of the period and the output subtask to be executed at the end, as illustrated in Figure 3.

The input and output subtasks can be forced to execute at desired time instants by treating them as interrupt routines activated by dedicated timers. The processing subtask, instead, is handled as a normal periodic task, subject to preemption according to the scheduling algorithm. This method will be referred to as *Reducing Jitter by Task Splitting* (RJTS). To avoid using two distinct timers, the output part of the k-th job can be executed at the beginning of the next period, that is just before the execution of the input part of the $(k+1)$-th job.

Figure 4 shows how the task set illustrated in Figure 2 would be handled by using the RJTS technique. Note that the output part of each job is always executed at the beginning of the next period, just before the input part of the next

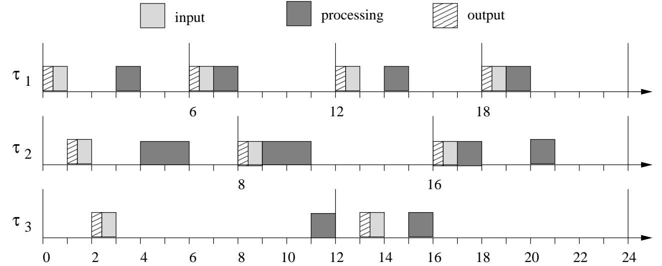


Figure 4. A task set executed according to the RJTS method.

job. As it can be seen from the example, treating the input-output parts as interrupt handling routines can significantly reduce the jitter for all tasks. The advantages of this method are the following:

- The fixed input-output delay greatly simplifies both the controller design and the assertion of system stability. The case of a one-sample delay is especially simple to handle in the control analysis [1]. In contrast, analyzing stability under jitter is much more difficult and requires either knowledge of the statistical distributions of the delays [24] or leads to potentially conservative results [19].
- In theory, the task splitting method can be applied to all the tasks and under any workload, whenever the task splitting overhead can be considered negligible.

However, there are other concerns that must be taken into account when applying this technique:

- The jitter reduction is obtained by inserting extra delay in the task execution. In fact, when applying this method, input and output parts are always separated by exactly a period, while normally the average delay could be smaller. The effect of having a higher delay in the control loop has to be carefully analyzed, since it could be more negative than the effect of jitter.

- Executing the input/output parts as high priority interrupt routines causes extra interference on the processing parts of the control tasks. Such an interference needs to be taken into account in the guarantee test. Feasibility analysis becomes more complex, but can still be performed using appropriate methods, like the one proposed by Jeffay and Stone [17], or by Facchinetti *et al.* [14].

- The extra interference caused by the input/output parts running as high priority interrupt routines decreases the schedulability of the system. As a consequence, tasks must run at lower rates with respect to the normal case.

- The input and output parts of different tasks may compete for the processor among themselves, hence they need to be scheduled with a policy that, in general, could be different than that used for the control tasks (e.g., it could be preemptive or non preemptive). As a consequence, a two-level scheduler is required in the real-time kernel to support such a mechanism. Figure 5 shows an example in which the input and output parts of different tasks overlap in time and require a scheduling policy. In the example, input and output parts always preempt processing parts, but among themselves they are scheduled with a priority equal to the task priority they belong. More specifically, if $P_1, \dots, P_n$ are the priorities assigned to the n control tasks (where $P_1 \geq P_2 \geq \dots \geq P_n$), each corresponding input and output part of task τ_i can be assigned a priority $P_i^* = P_0 + P_i$, where P_0 is a priority level higher than P_1.

- Finally, the implementation of the RJTS method requires an extra effort to the user, who has to program a timer for each task to trigger the input and output parts at the period boundaries.

4.2 Reducing jitter by advancing deadlines

Another common approach that can be applied to reduce jitter is to shorten the task relative deadlines. In fact, if a task τ_i is guaranteed to be executed with a relative deadline D_i, clearly its input-output jitter, as well as its response time jitter, cannot be greater than $D_i - C_i$. This method will

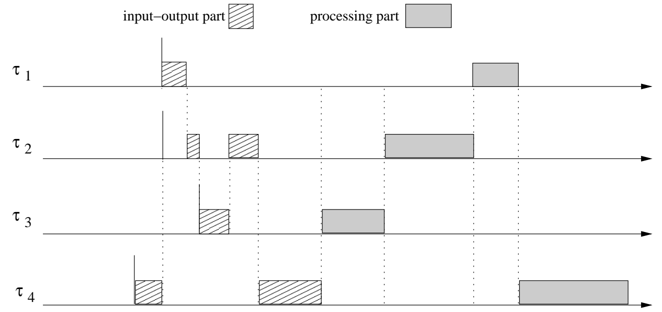


Figure 5. Input and output subtasks need to be scheduled when they overlap in time.

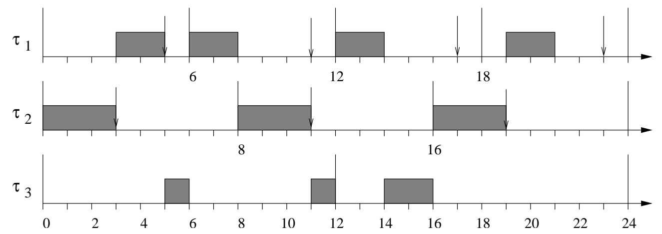


Figure 6. A task set executed according to the RJAD method.

be referred to as *Reducing Jitter by Advancing Deadlines* (RJAD). Following this approach, Baruah *et al.* [5] proposed two methods for assigning shorter relative deadlines to tasks and guaranteeing the schedulability of the task set.

Figure 6 illustrates an example in which the three tasks shown in Figure 2 are required to execute with a jitter no higher that 3, 0, and 10, so their deadlines are set to $D_i = C_i + Jitter$, that is to 5, 3, and 12, respectively.

The RJAD method has the following advantages with respect to the RJTS approach:

- The method does not require any special support from the operating system, since jitter constraints are simply mapped into deadline constraints.

- No extra effort is required to the user to program dedicated timers. Once jitter constraints are mapped into deadlines, the kernel scheduler can do the job.

- There are no interrupt routines creating extra interference in the schedule, so the guarantee test can be performed with the classical response time analysis [21, 2] or more efficient techniques [6], under fixed priorities, or with the processor demand criterion [4], under EDF.

- Advancing deadlines, jitter and delay are both reduced, implying better achievable control performance.

However, there are also the following disadvantages with respect to the RJTS approach:

- The major problem of this method is that it cannot reduce the jitter of all the tasks, but only a few tasks can be selected to run with zero (or very low) jitter. A better result could be achieved by exploiting task release offsets, but the analysis becomes of exponential complexity.

- Advancing task deadlines, the system schedulability could be reduced. As a consequence, as in the RJTS method, tasks could be required to run at lower rates with respect to the normal case.

4.3 Reducing jitter by non preemption

A third method for reducing the input-output jitter of a task is simply to execute it in a non preemptive fashion. This method will be referred to as *Reducing Jitter by Non Preemption* (RJNP). For example, if the task set illustrated in Figure 2 is executed using non preemptive rate-monotonic scheduling, the resulting schedule would be, for this particular case, the same as that one generated by EDF, depicted in Figure 2b.

The RJNP method has the following advantages:

- Using a non preemptive scheduling discipline, the input-output jitter becomes very small for all the tasks (assuming that the task execution times are constant), since the interval between the input and output parts is always equal to the task computation time. This makes it easy to compensate for the delay in the control design.

- Another advantage of this method is that the input-output delay is also reduced to the minimum, which is also equal to the task computation time. This probably gives the largest performance improvement, since control loops are typically more sensitive towards delay than jitter.

- Non preemptive execution allows using stack sharing techniques [3] to save memory space in small embedded systems with stringent memory constraints [15].

On the other hand, the RJNP approach introduces the following problems:

- A general disadvantage of the non preemptive discipline is that it reduces schedulability. In fact, a non preemptive section of code introduces an additional blocking factor in higher priority tasks that can be taken into account with the same guarantee methods used for resource sharing protocols.

- There is no least upper bound on the processor utilization below which the schedulability of any task set can be guaranteed. This can easily be shown by considering a set of two periodic tasks, τ_1 and τ_2, with priorities $P_1 > P_2$ and utilization $U_i = \varepsilon$, arbitrarily small. If $C_2 > T_1$, $C_1 = \varepsilon T_1$, and $T_2 = C_2/\varepsilon$, the task set is unschedulable, although having an arbitrarily small utilization.

- Achieving non preemptive execution for all user tasks is easy in standard operating systems (a single semaphore can be used), but making only one or a few tasks non preemptible requires a larger programming effort.

In order to evaluate the impact of the different approaches on control performance, the three methods have been compared by simulation under different scenarios. The results of the simulations are illustrated and discussed in the next section.

5 Experimental Results

5.1 Simulation Set-Up

We consider a system with $n = 7$ periodic tasks that are scheduled under EDF[1] [22]. The tasks are distinguished in two categories: a subset of control tasks, whose jitter and delay must be bounded, and a subset of hard real-time tasks, with no jitter and delay requirements, except for schedulability. The number of control tasks is changed in the experiments as a simulation parameter.

The performance of the various jitter reduction methods is evaluated by monitoring the execution of a control task, τ_1, with period $T_1 = 50$ ms and constant execution time $C_1 = 5$ ms. The input and output operations are assumed to take $I_1 = O_1 = 0.5$ ms, leaving $P_1 = 4$ ms for the processing part. Notice that τ_1 is not necessarily the task with the shortest deadline. In fact, the other six tasks, τ_2–τ_7, are generated with random attributes, with fixed execution times C_i uniformly distributed in [1, 10] ms, and utilizations U_i chosen according to a 6-dimensional uniform distribution to reach the desired total utilization (algorithm UUniFast in [7]). Task periods are then given by $T_i = C_i/U_i$. Relative deadlines are set equal to periods ($D_i = T_i$) for all hard real-time tasks, whereas they can be reduced by the RJAD method for the control tasks. The offset Φ_i of each task is uniformly distributed in [0, T_i].

The following parameters are varied in the simulation experiments:

- The total utilization U is varied between 0.2 to 0.9 in steps of 0.1. We also include the case $U = 0.99$.

[1] For lack of space we decided to perform the experiments only under EDF, which guarantees full processor utilization in the fully preemptive case, but similar simulations can be carried out under any fixed priority assignment.

- The number of control tasks is varied between 1 and 7. For each control task, $I_1 = O_1 = 0.5$ ms is assumed for the input and output phases.
- The implementation of the control tasks is varied depending on the specific technique used to reduce the jitter.

(We have assumed fixed execution times in an attempt to keep down the number of simulation parameters. In future work, we would like to study the effects of stochastic execution times as well.)

To better evaluate the enhancements achievable with the three methods discussed in Section 4, we also monitor the results when no special treatment is applied on control tasks. Thus, we consider the following four methods:

Standard Task Model (STM). Each task, including the control tasks, is assigned a relative deadline equal to the period.

Reducing Jitter by Task Splitting (RJTS) The output and input operations of the control tasks are implemented in non-preemptible timer interrupt routines. An extra overhead of 0.5 ms was assumed for this implementation, making the non-preemptible section 1.5 ms. This can potentially cause hard real-time tasks to miss their deadlines.

Reducing Jitter by Advancing Deadlines (RJAD) The deadlines of the control tasks are advanced according to Method 2 of Baruah *et al.* [5].

Reducing Jitter by Non Preemptive Execution (RJNP). Control tasks are implemented as non-preemptible tasks, whereas hard tasks can be normally preempted. This can potentially cause hard real-time tasks to miss their deadlines.

For each parameter configuration, $N = 500$ random task sets are generated, and the system is simulated for 50 s (corresponding to 1000 invocations of τ_1). The absolute input jitter (INJ), the absolute input-output jitter (IOJ), and the average input-output latency (IOL) for τ_1 are recorded for each experiment. To evaluate the effect of each method on task schedulability, we also recorded the percentage of unfeasible task sets, i.e., the number of task sets where one or more hard real-time tasks miss a deadline. When a deadline is missed, the simulation for the current task set is aborted and the performance is not counted.

Two sets of experiments were carried out: one to evaluate the effects of the methods on the timing behavior, and one to see their impact on the control performance. For the INJ, IOJ, and IOL results, standard deviations for the average values were computed and they were never greater than 0.35 ms for any configuration. For the control cost evaluation, the standard deviation was less than 0.2%, not counting cases where the control loop went unstable.

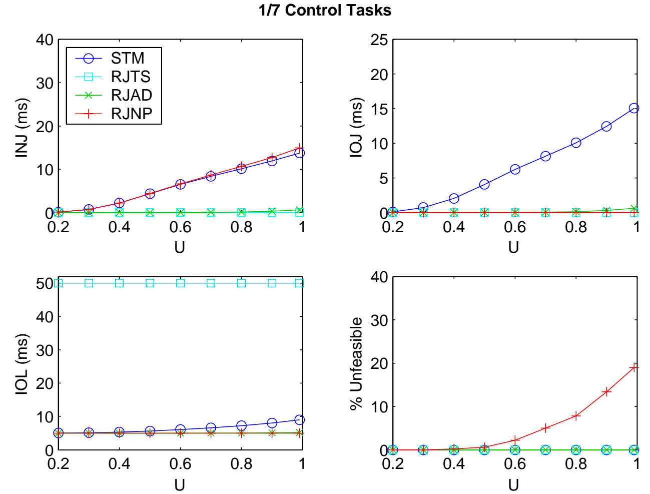


Figure 7. Jitter results with one control task and six hard real-time tasks.

5.2 Jitter Results

We first tested the jitter reduction methods with one control task and six hard real-time tasks. The results are reported in Figure 7. Note that RJAD is very successful in reducing both the input jitter (INJ) and the input-output jitter (IOJ) while minimizing the input-output latency (IOL) for all load cases. RJTS also performs very well in reducing both INJ and IOJ for any utilization, but gives a very long IOL, as expected. Finally, RJNP is able to reduce the IOJ and the IOL to a minimum, but actually increases the INJ slightly compared to the standard task model. RJNP is also the most invasive method in terms of schedulability (bottom-right graph), causing hard real-time tasks to miss deadlines for utilizations higher than 0.5.

Figure 8 illustrates the results achieved with four control tasks and three hard real-time tasks. In this case, RJAD does not work quite as good anymore, because four control tasks to advance their deadlines. RJNP keeps the IOJ and the IOL and a minimum, while the INJ increases. Moreover, it causes even more deadlines to be missed with respect to the previous case. Note that some deadlines are also missed with RJTS at high utilizations, due to the non-preemptible interrupt routines needed for executing the input-output parts at the end of the periods. For the same reason, some jitter is now also experienced by RJTS (upper graphs).

A final simulation experiment has been performed with seven control tasks and the results are shown in Figure 9. In this case, a very high sampling jitter occurs under RJNP (upper-left graph), whereas RJAD shows virtually no improvement at all with respect to the standard task model (STM, upper graphs). Also RJTS experiences more jitter than before, due to the higher number of interrupts. Notice that no deadlines are missed (bottom-right graph) since there are no hard tasks anymore.

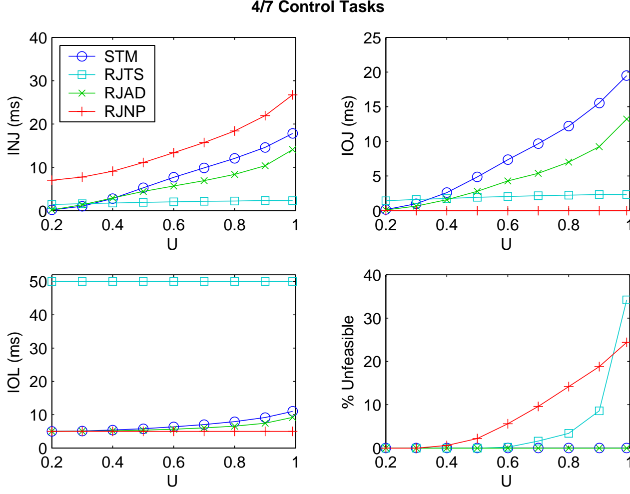


Figure 8. Jitter results with four control tasks and three hard real-time tasks.

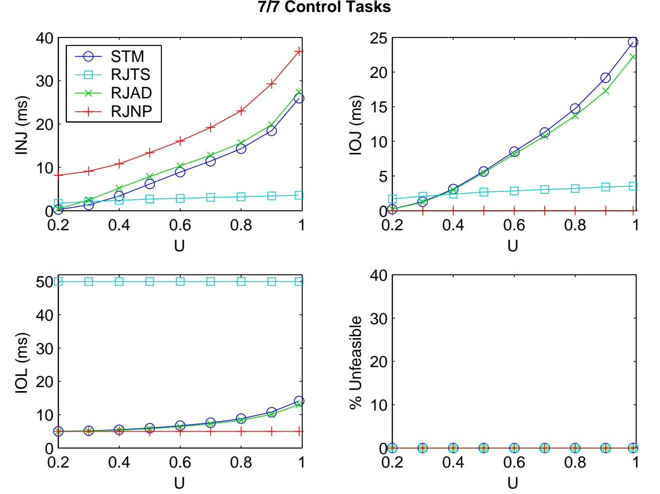


Figure 9. Jitter results with seven control tasks.

5.3 Control Performance Results

The actual control performance degradation resulting from scheduling-induced delay and jitter depends very much on the control application. In general, however, controllers with a high sampling rate (compared to, e.g., the cross-over frequency) are less sensitive to delay and jitter.

In this section, we study two benchmark control problems: one assuming fast sampling and the other assuming slow sampling. In each case, it is assumed that task τ_1 implements a Linear Quadratic Gaussian (LQG) controller [1] to regulate a given plant. The sampling period is hence $T_1 = 50$ ms in both cases. Associated with each plant is a quadratic cost function V that is used both for the LQG control design and for the performance evaluation of the control loop. The two plants and their cost functions are given below:

Plant 1:

$$\frac{dx}{dt} = \begin{bmatrix} 0 & 1 \\ 9 & 0 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u + \begin{bmatrix} 1 \\ 0 \end{bmatrix} v$$
$$y = \begin{bmatrix} 0 & 1 \end{bmatrix} x + \sqrt{0.1} e$$

$$V = \mathrm{E} \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(x^T \begin{bmatrix} 0 & 0 \\ 0 & 10 \end{bmatrix} x + u^2 \right) dt$$

Plant 2:

$$\frac{dx}{dt} = \begin{bmatrix} 0 & 1 \\ -3 & -4 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u + \begin{bmatrix} 35 \\ -61 \end{bmatrix} v$$
$$y = \begin{bmatrix} 2 & 1 \end{bmatrix} x + e$$

$$V = \mathrm{E} \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(x^T \begin{bmatrix} 2800 & 80\sqrt{35} \\ 80\sqrt{35} & 80 \end{bmatrix} x + u^2 \right) dt$$

Here, x is the plant state vector, u is the control signal, y is the measurement signal, v is a continuous-time zero-mean white noise process with unit intensity, and e is a discrete-time zero-mean white noise process with unit variance. Further, v and e are assumed to be independent.

Plant 1 is a linear model of an inverted pendulum (an unstable plant) with a natural frequency of 3 rad/s. Assuming a constant input-output delay of 5 ms, the cost function produces an LQG controller that achieves a cross-over frequency of 5.2 rad/s and a phase margin of 31°. This can be seen as a typical, quite robust control design with a high enough sampling rate. The jitter margin [10] is computed to 82 ms, which is larger than the sampling period and indicates that the control loop cannot be destabilized by scheduling-induced jitter.

Plant 2 (a stable plant) and its associated cost function represent a pathological case where the LQG design method gives a controller that is very sensitive to delay and jitter [24]. Assuming a delay of 5 ms, the resulting cross-over frequency is 20.6 rad/s while the phase margin is only 17°. The jitter margin is found to be only 10 ms, indicating that latency and jitter from the scheduling might destabilize the control loop.

Assuming the same simulation set-up as in the jitter evaluation, for each parameter configuration, $N = 500$ random task sets were generated and the control performance of task τ_1 was recorded for 50 s. For STM, RJAD, and RJNP, the controller was designed assuming a constant delay of 5 ms, while for RJTS, the delay was assumed to be 50 ms. The whole procedure was repeated for each of the two plants, the results being reported in Figure 10. In the figure, the costs have been normalized such that the performance under minimum delay and jitter is 1.

For Plant 1, it can be noted that RJTS performs uniformly worse than the other implementations, including

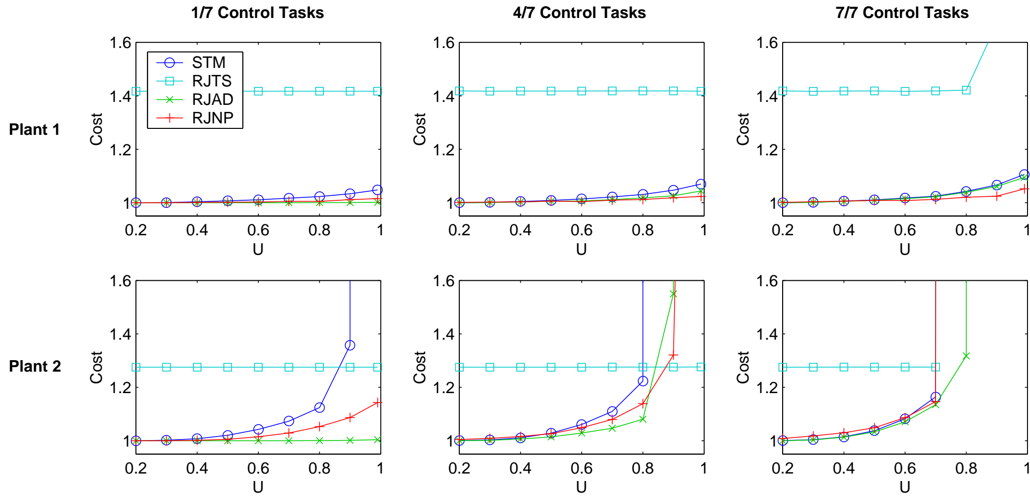


Figure 10. Control performance evaluation for two different plants.

STM. This is due to the long input-output latency, which, despite the exact delay compensation, destroys the performance. As the number of control tasks and the load increase, the performance of STM, RJAD, and RJNP degrades only slightly, RJNP performing the best in the case of multiple controllers, while RJAD works the best for a single controller. Also notice the performance break-down of RJTS for 7 control tasks and high loads, where the extra scheduling overhead causes task τ_1 to miss its outputs.

For Plant 2, the issue of jitter is more critical. While RJTS gives a constant performance degradation (except for the case of seven control tasks and high load) the other implementations exhibit degradations that seem to relate to the total amount of jitter (INJ + IOJ) reported in Figures 7–9. As a consequence, RJAD has an edge over RJNP for almost all system configurations. Note that, for sufficiently high utilizations, some implementations actually cause the control loop to go unstable (the cost approaches infinity) for high loads.

In summary, RJAD gives the better control performance for most system configurations. At the same time, it does not cause any deadlines to be missed. RJTS is the "safest" implementation, in that it gives a more or less constant performance degradation, even for an unrobust control design, many control tasks, and a high CPU load.

6 Conclusions

In this paper we studied three scheduling techniques for reducing delay and jitter in real-time control systems consisting of a set of concurrent periodic tasks. The first method (RJTS) reduces jitter by splitting a task in three subtasks (input, processing and output) and forces the execution of input and output parts at the period boundaries. The second method (RJAD) reduces jitter and delay by assigning the tasks a shorter deadline. The third method (RJNP) simply relies on non preemptive execution to eliminate input-output jitter and delay.

Advantages and disadvantages of the three approaches have been discussed and a number of simulation experiments have been carried out to compare these techniques and illustrate their effect on control system performance. In the simulation results, it was seen that RJTS and RJNP can compromise the schedulability of the system if this issue is not taken into account at design time. RJAD performed very well for a single control task and reasonably well for multiple control tasks. RJTS was able to reduce both the input jitter and the input-output jitter to a minimum but produced a long input-output latency.

In conclusion, for a robust control design (with sufficient phase and delay margins), the performance degradation due to jitter is very small, even for the standard task model. For a single control task, the RJAD method can reduce this degradation all the way down to zero in most cases. On the other hand, a constant one-sample delay gives a very large penalty in comparison. Hence, it is clear that the RJTS method should be avoided for robust control systems.

For unrobust control systems with very small phase and delay margins, RJTS could be considered to be a "safe" choice of implementation. For many system configurations, however, even STM performs better than RJTS. One has to go to quite extreme situations to find examples where RJTS actually gives better control performance than STM. In these situations, the computing capacity is probably severely under-dimensioned, and it is questionable whether *any* implementation can actually meet the control performance requirements.

Finally, in terms of control performance, the RJNP method sometimes performs better and sometimes worse than RJAD in the various configurations. However, one

should keep in mind, that RJNP may cause hard tasks to miss their deadlines and requires a more difficult off-line analysis.

As a future work, we plan to provide support for these techniques in the Shark operating system, in order to evaluate the effectiveness of these approaches on real control applications. Also, we would like to explore the trade-off between jitter and delay for a wider class of plants and controllers.

References

[1] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems: Theory and Design.* Third edition. Prentice-Hall, 1997.

[2] N. C. Audsley, A. Burns, M. Richardson, K. W. Tindell, and A. J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, Vol. 8, No. 5, pp. 284–292, September 1993.

[3] T. P. Baker, "Stack-Based Scheduling of Real-Time Processes," *Real-Time Systems* Vol. 3, No. 1, pp. 67–100, 1991.

[4] S. K. Baruah, R. R. Howell, and L. E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, 2, 1990.

[5] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari, "Scheduling Periodic Task Systems to Minimize Output Jitter," in *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications*, Hong Kong, December 1999.

[6] E. Bini and G. Buttazzo, "Schedulability Analysis of Periodic Fixed Priority Systems," *IEEE Transactions on Computers*, Vol. 53, No. 11, pp. 1462–1473, November 2004.

[7] E. Bini and G. Buttazzo, "Biasing Effects in Schedulability Measures," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, July 2004.

[8] G. Buttazzo and F. Sensini, "Optimal Deadline Assignment for Scheduling Soft Aperiodic Task in Hard Real-Time Environments," *IEEE Transactions on Computers,* Vol. 48, No. 10, October 1999.

[9] G. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day", *Real-Time Systems*, Vol. 28, pp. 1-22, 2005.

[10] A. Cervin, B. Lincoln, J. Eker, K.-E. Arzen, and G. Buttazzo, "The Jitter Margin and Its Application in the Design of Real-Time Control Systems," in *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Gothenburg, Sweden, August 2004.

[11] A. Crespo, I. Ripoll and P. Albertos, "Reducing delays in RT control: the control action interval," in *Proceedings of the 14th IFAC World Congress*, pp. 257–262, 1999.

[12] C. Davidson, "Random sampling and random delays in optimal control," PhD Dissertation 21429, Department of Optimization and Systems Theory, Royal Institute of Technology, Sweden, 1973.

[13] M. Di Natale and J. Stankovic, "Scheduling Distributed Real-Time Tasks with Minimum Jitter," *IEEE Transactions on Computers*, Vol. 49, No. 4, pp. 303–316, 2000.

[14] T. Facchinetti, G. Buttazzo, M. Marinoni, G. Guidi, "Non-Preemptive Interrupt Scheduling for Safe Reuse of Legacy Drivers in Real-Time Systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[15] P. Gai, G. Lipari, and M. di Natale, "Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip," in *Proceedings of the 22th IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[16] R. Gerber and S. Hong, "Semantics-Based Compiler Transformations for Enhanced Schedulability," in *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[17] K. Jeffay and D. L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, USA, pp. 212–221, December 1993.

[18] R. E. Kalman and J. E. Bertram, "A unified approach to the theory of sampling systems," *J. Franklin Inst.*, Vol. 267, pp. 405–436, 1959.

[19] C.-Y. Kao and B. Lincoln, "Simple Stability Criteria for Systems with Time-Varying Delays," *Automatica*, 40:8, pp. 1429–1434, August 2004.

[20] H. J. Kushner and L. Tobias, "On the stability of randomly sampled systems," *IEEE Transactions on Automatic Control*, Vol 14, No 4, pp. 319–324, 1969.

[21] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the 10th IEEE Real-Time Systems Symposium*, Santa Monica, CA, USA, pp. 166–171, December 1989.

[22] C. L. Liu and J. W. Layland, "Scheduling algorithms for Multipprogramming in Hard Real-Time traffic environment," *Journal of the Association for Computing Machinery,* Vol. 20, No. 1, January 1973.

[23] P. Martí, G. Fohler, K. Ramamritham, and J.M. Fuertes, "Jitter Compensation for Real-time Control Systems," in *Proceedings of the 22nd IEEE Real-Time System Symposium*, London, UK, December 2001.

[24] J. Nilsson, B. Bernhardsson and B. Wittenmark, "Stochastic Analysis and Control of Real-Time Systems with Random Time Delays," *Automatica*, 34:1, pp. 57–64, 1998.

[25] Q. Zheng and K. G. Shin, "On the Ability of Establishing Real-Time Chennals in Point-to-Point Packet-Switched Networks," *IEEE Transactions on Communications,* Vol. 42, No. 2/3/4, Feb./March/Apr. 1994.

Reducing Delay and Jitter in Software Control Systems

Hoai Hoang
Centre for Research on Embedded Systems
Halmstad University
Halmstad, Sweden
Hoai.Hoang@ide.hh.se

Giorgio Buttazzo
Real-Time Systems Laboratory
Scuola Superiore Sant'Anna
Pisa, Italy
giorgio@sssup.it

Abstract

Software control systems may be subject to high interference caused by concurrency and resource sharing. Reducing delay and jitter in such systems is crucial for guaranteeing high performance and predictability. In this paper, we present a general approach for reducing delay and jitter by acting on task relative deadlines. The method allows the user to specify a deadline reduction factor for each task to better exploit the available slack according to specific jitter sensitivity. Experimental results confirm the effectiveness and the generality of the proposed approach with respect to other methods available in the literature.

1 Introduction

Complex software systems are often implemented as a number of concurrent tasks that interact with a given set of resources (processor, memories, peripherals, etc.). Tasks related to control activities are typically periodic, and are activated with a specific rate derived by the system's designer. Other tasks related to specific input/output devices (e.g., serial lines, data buses, networks) may be aperiodic and can be activated by interrupts or by the occurrence of particular events.

Although the activation rates of periodic tasks can be precisely enforced by the operating system through proper kernel mechanisms, the execution pattern of each task depends on several factors, including the scheduling algorithm running in the kernel, the overall system workload, the task set parameters, the interaction with the shared resources, and the interference introduced by interrupts. As a consequence, control tasks may experience variable delays and jitter that can degrade the system performance, if not properly handled.

The effects of delays and jitter on real-time control applications have been extensively studied in the literature [10, 19] and several methods have been proposed to cope with them. Marti el al. [18] presented a control technique to compensate the effect of jitter with proper control actions computed based on the temporal distance between successive samples. Cervin et al. [11] presented a method for finding an upper bound of the input-output jitter of each task by estimating the worst-case and the best-case response time under EDF scheduling, but no method is provided to reduce the jitter. Rather, the concept of *jitter margin* is introduced to simplify the analysis of control systems and guarantee their stability when certain conditions on jitter are satisfied.

Other authors proposed suitable scheduling methods for reducing the delay and jitter caused by complex intertask interference. For example, Di Natale and Stankovic [12] proposed the use of simulated annealing to find the optimal configuration of task offsets that minimizes jitter, according to some user defined cost function. Baruah et al. [4] followed a different approach to reduce both delay and jitter by reducing the relative deadline of a task, so limiting the execution interval of each job. Two methods have been illustrated for assigning shorter relative deadlines to tasks while guaranteeing the schedulability of the task set: the first method is based on task utilizations and runs in polynomial time, whereas the second one (more effective) has a pseudo-polynomial complexity since it is based on the processor demand criterion [2].

Brandt et al. [6] also addressed the problem of reducing the deadline of a periodic task, but their approach is based on the processor utilization, hence it cannot be used to find the shortest possible deadline.

Zheng et al. [20] presented a method for computing the minimum deadline of a newly arrived task, assuming the existing task set is feasibly schedulable by EDF; however, their approach is tailored for distributed applications and requires some off-line computation. When the utilization of all the tasks in the task set is high, the number of off-line computations are very large, that make this method become not efficiently, high computational complexity.

Buttazzo and Sensini [7] also presented an on-line algorithm to compute the minimum deadline to be assigned to

a new incoming task in order to guarantee feasibility under EDF. However, their approach only applies to aperiodic requests that have to be executed in a periodic environment.

Hoang et al. [16] and Balbastre et al. [5] independently proposed a method for minimizing the relative deadline of a single periodic task, while keeping the task set schedulable by EDF. Although the approach can be applied sequentially to other tasks in a given order, the deadline reduction achievable on the first task is much higher than that achievable on the other tasks in the sequence. To avoid this problem, in the same paper, Balbastre et al. also proposed a method to perform a uniform scaling of all relative deadlines. The problem with a uniform reduction, however, is that jitter and delay may not necessarily reduce as expected (and for some task they could even increase).

To allow more flexibility in controlling the delay and jitter in software controlled systems, in this paper we present a general approach for reducing task deadlines according to individual task requirements. The method allows the user to specify a deadline reduction factor for each task, to better exploit the available slack according to tasks actual requirements. The deadline reduction factor can be specified as a real number in [0,1], with the meaning that a value equal to one allows the relative deadline to be reduced up to the minimum possible value (corresponding to the task computation time), whereas a value equal to zero means no reduction.

As special cases, the method can minimize the deadline of a single task (by setting its reduction factor to 1 and the others to zero), or perform a uniform deadline rescaling in the task set (by setting all reduction factors to 1). If two tasks have the same delay/jitter requirements and need to reduce their relative deadlines as much as possible, this can simply be achieved by setting both reduction factors to 1 and all the others to zero. Note that this could not be achieved by applying a deadline minimization algorithm [16, 5] to the tasks in a given order, because the first task would steal all the available slack for itself, leaving small space for the second.

The rest of the paper is organized as follows. Section 2 presents the system model and the terminology adopted throughout the paper. Section 3 illustrates the addressed problem with some concrete examples. Section 4 describes the deadline reduction algorithm. Section 5 presents some experimental results and compares the proposed method with other deadline reduction approaches. Finally, Section 6 states our conclusions and future work.

2 Terminology and assumptions

We consider a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic tasks that have to be executed on a uniprocessor system under the Earliest Deadline First (EDF) algorithm [17]. Each task τ_i consists of an infinite sequence of jobs, or task instances, having the same worst-case execution time and the same relative deadline. All tasks are fully preemptive. The following notation is used throughout the paper:

$\tau_{i,j}$ denotes the j-th job of task τ_i, (where $j = 1, 2, \ldots$), that is the j-th instance of the task execution.

$r_{i,k}$ denotes the release time of job $\tau_{i,k}$, that is the time at which the job is activated and becomes ready to execute.

$s_{i,k}$ denotes the start time of job $\tau_{i,k}$, that is the time at which the first instruction of $\tau_{i,k}$ is executed.

$f_{i,k}$ denotes the finishing time of job $\tau_{i,k}$, that is the time at which the job completes its execution.

C_i denotes the worst-case execution time of task τ_i.

T_i denotes the period of task τ_i, or the minimum interarrival time between successive jobs.

D_i denotes the relative deadline of task τ_i, that is, the maximum finishing time (relative to its release time) allowed for any job.

$d_{i,j}$ denotes the absolute deadline of job $\tau_{i,j}$, that is the maximum absolute time before which job $\tau_{i,j}$ must complete ($d_{i,j} = r_{i,j} + D_i$).

U_i denotes the utilization of task τ_i, that is the fraction of cpu time used by τ_i ($U_i = C_i/T_i$).

U denotes the total utilization of the task set, that is, the sum of all tasks utilizations ($U = \sum_{i=1}^{n} U_i$).

$R_{i,j}$ denotes the response time of job $\tau_{i,j}$, that is the interval between its release time and its finishing time:

$$R_{i,j} = f_{i,j} - r_{i,j}. \tag{1}$$

$IOD_{i,j}$ denotes the input-output delay of job $\tau_{i,j}$, that is the interval between its start time and its finishing time:

$$IOD_{i,j} = f_{i,k} - s_{i,k}. \tag{2}$$

RTJ_i denotes the response time jitter of a task, that is the maximum variation in the response time of its jobs:

$$RTJ_i = \max_k\{R_{i,k}\} - \min_k\{R_{i,k}\} \tag{3}$$

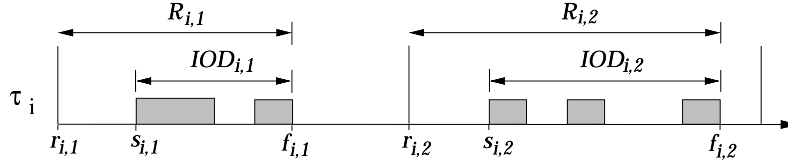


Figure 1. Example of a real-time task.

Figure 1 illustrates some of the parameters defined above.

Moreover, each task τ_i is characterized by a maximum relative deadline D_i^{max} (considered to be the nominal one) and a minimum relative deadline D_i^{min}, specified by the application designer. When not explicitly assigned, we assume $D_i^{max} = T_i$ and $D_i^{min} = C_i$.

3 Problem statement

The method proposed in this work to reduce delay and jitter in periodic tasks requires the application designer to specify an additional task parameter, called the *deadline reduction factor*, δ_i, which is a real number in [0,1]. A value $\delta_i = 1$ indicates that task τ_i is very sensitive to delay and jitter, hence its relative deadline is allowed to be reduced up to its minimum possible value (D_i^{min}). A value $\delta_i = 0$ indicates that task τ_i is not sensitive to delay and jitter, so its relative deadline does not need to be modified. In general, we assume that the sensitivity of τ_i to delay and jitter is proportional to δ_i.

Once all deadline reduction factors have been specified according to delay and jitter sensitivity, the problem we want to solve is to shorten all deadlines as much as possible to respect the proportions dictated by the reduction factors, while keeping the task set feasible.

Note that task specific jitter coefficients have also been defined by Baruah et al. [4] (they were denoted by ϕ_i and called jitter tolerance factors). In that work, however, the objective was to minimize the weighted jitter of the task set, defined as

$$\text{WtdJitter}(\mathcal{T}) = \max_i \left\{ \frac{RTJ_i}{\phi_i} \right\},$$

rather than reducing deadlines proportionally to sensitivity, as done in this paper.

To better motivate the proposed approach, we now illustrate an example that shows the advantage of specifying individual reduction factors.

3.1 A motivating example

Consider a set of three periodic tasks with periods $T_1 = 6$, $T_2 = 9$, $T_3 = 12$, and computation times $C_1 = 1$, $C_2 = 2$, $C_3 = 5$. Suppose that τ_1 and τ_2 are control tasks sensitive to delay and jitter, whereas τ_3 is not and can be executed anywhere within its period. Assuming $D_i = D_i^{max} = T_i$ for each task, the schedule produced by EDF is shown in Figure 2. The response time jitters of the tasks are $RTJ_1 = 2$, $RTJ_1 = 3$, and $RTJ_1 = 2$.

Now observe that, for this task set, the uniform scaling algorithm proposed by Balbastre et al. [5] does not produce any change in the schedule, so it cannot reduce any jitter. For this particular case, in fact, the maximum common reduction factor that guarantees a feasible schedule is 1/3, meaning that for each task we can set $D_i = (2/3)T_i$. As shown in Figure 3, however, the schedule produced by EDF with such deadlines is exactly the same as that shown in Figure 2.

Also notice that minimizing the deadline of a single task (using the algorithm proposed by Hoang et al. [16] or by Balbastre et al. [5]) may not necessarily have the desired effect on the other jitter sensitive tasks. For example, as depicted in Figure 4, minimizing D_2 the jitter of τ_2 becomes zero, but the jitter of τ_1 cannot be reduced below 2 (even if D_1 is minimized after D_2).

In this case, a better solution to reduce the delay and jitter of τ_1 and τ_2 is to reduce the deadlines of both tasks simultaneously, leaving D_3 unchanged. As an example, assuming $\delta_1 = \delta_2 = 1$, the maximum common reduction factor that can be applied to both tasks to keep the task set feasible is 2/3, meaning that we can set $D_1 = T_1/3$, $D_2 = T_2/3$, and $D_3 = T_3$. Figure 5 shows the schedule produced by EDF with such deadlines.

The next section describes the algorithm that computes the new feasible deadlines according to the specified reduction factors δ_i.

4 The algorithm

Before describing the algorithm, it is worth observing that, for the feasibility constraint, the actual deadline reduction will be less than or equal to the one specified by the reduction factor, that is

$$\frac{D_i^{max} - D_i}{D_i^{max} - D_i^{min}} \leq \delta_i.$$

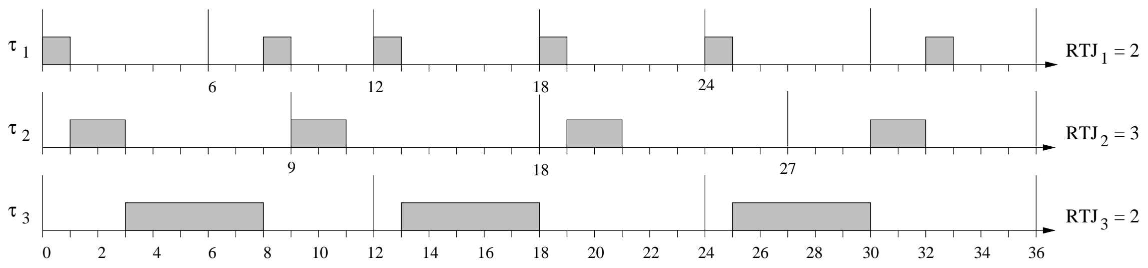


Figure 2. EDF schedule with relative deadlines equal to periods.

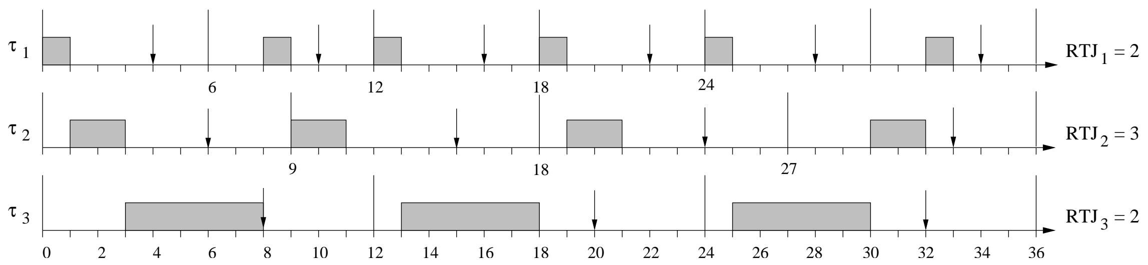


Figure 3. EDF schedule with uniformly scaled deadlines.

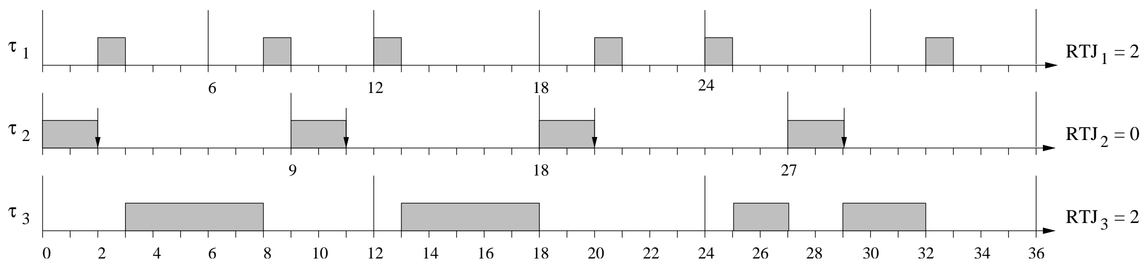


Figure 4. EDF schedule when only τ_2 deadline is minimized.

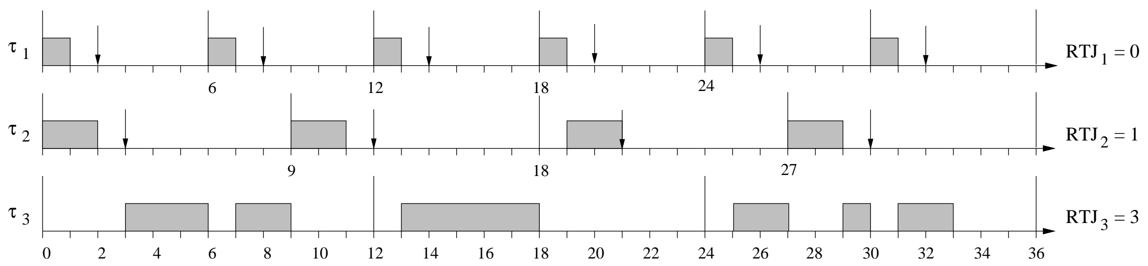


Figure 5. EDF schedule when deadlines of both τ_1 and τ_2 are reduced.

However, to respect the proportions specified by the reduction factors, we must compute the new deadlines in such a way that $\forall i, j = 1, \ldots, n$ (and $\delta_i, \delta_j \neq 0$) we have

$$\frac{D_i^{max} - D_i}{D_i^{max} - D_i^{min}} / \delta_i = \frac{D_j^{max} - D_j}{D_j^{max} - D_j^{min}} / \delta_j.$$

This means that

$$\forall i = 1, \ldots, n \quad \frac{D_i^{max} - D_i}{D_i^{max} - D_i^{min}} / \delta_i = \alpha$$

where α is a constant less than or equal to one. Hence, the problem consists in finding the greatest value of α that keeps the task set feasible, where deadlines are computed as

$$D_i = D_i^{max} - \alpha\delta_i(D_i^{max} - D_i^{min}) \quad (4)$$

The highest value of α that guarantees feasibility can be found by binary search.

The search algorithm assumes that the task set $\mathcal{T}$ is feasible for $\alpha = 0$ (that is, when all tasks are scheduled with the maximum deadlines), and starts by trying feasibility with $\alpha = 1$ (that is, with all tasks having their minimum deadlines). If $\mathcal{T}$ is found feasible with $\alpha = 1$, then the algorithm exits with the best solution, otherwise the binary search is started.

The feasibility test as a function of α can be performed using the function reported in Figure 6, where all relative deadlines are first computed according to Equation (4), and then the test is performed using the Processor Demand Criterion [2, 3]. In particular, the $Processor_demand_test(\mathcal{T})$ function returns 1 if the task set $\mathcal{T}$ is feasible, 0 otherwise.

```
Feasible(T, α)

    for i = 0 to n
        D_i = D_i^max − αδ_i(D_i^max − D_i^min);
    end for

    F = Processor_demand_test(T);

    return (F);
end
```

Figure 6. Feasibility test as a function of α.

The binary search algorithm to find the highest value of α is reported in Figure 7. Note that, besides the task set parameters, the algorithm requires a value ε, needed to bound the complexity of the search and stop the algorithm when the search interval ($\Delta = \alpha_{max} - \alpha_{min}$) becomes smaller than a given error.

```
Best_alpha(T, ε)
    α_max = 1;
    α_min = 0;
    Δ = α_max − α_min;

    if Feasible(T, α_max) then return(α_max);

    while (Δ > ε) do
        α = (α_max + α_min)/2;

        if Feasible(T, α) then α_min = α;
        else α_max = α;

        Δ = α_max − α_min;
    end while

    return(α_min);

end
```

Figure 7. Binary search algorithm for finding the highest α that guarantees feasibility.

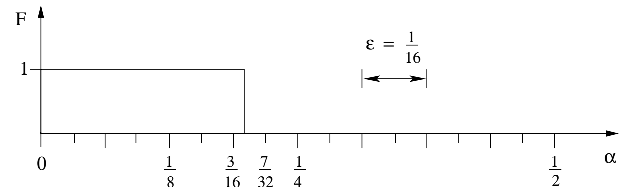


Figure 8. Search values for $\varepsilon = 1/16$.

For example, for the case illustrated in Figure 8, if $\varepsilon = 1/16$, the algorithm will try six values (1, 1/2, 1/4, 1/8, 3/16, and 7/32) and stops by returning the last feasible value, that is $\alpha = 3/16$. In general, the complexity of the algorithm is logarithmic with respect to ε, and the number of steps to find the best α is given by

$$N = 2 - \log_2 \varepsilon.$$

For the given example, $\log_2(1/16) = -4$, so we have $N = 6$. Once the highest feasible α is found, the task deadlines are reduced according to Equation (4), which takes into account the individual reduction factors.

5 Experimental results

This section describes a set of simulation experiments that have been conducted to evaluate the effectiveness of the proposed algorithm to reduce delay and jitter of specific tasks according to given reduction factors. For special cases, the method is also compared with the algorithm that uniformly scales all deadlines [5] and with the algorithm that minimizes the relative deadline of a single task [16, 5].

We have investigated different application scenarios, generated through synthetic task sets with random parameters within given ranges and distributions. To generate a feasible task set of n periodic tasks with given utilization $U_d < 1$, we first generated n random utilizations uniformly distributed in (0,1) and then normalized them to have

$$\sum_{i=1}^{n} U_i = U_d.$$

Then, we generated n computation times as random variables uniformly distributed in $[C_{min}, C_{max}]$ (with $C_{min} = 5$ and $C_{max} = 30$) and then calculated the period of each task as

$$T_i = \frac{C_i}{U_i}.$$

For each task τ_i, D_i^{max} has been set equal to T_i and D_i^{min} has been set equal to C_i.

For each generated task set $\mathcal{T}$, we measured the maximum response time of each task ($R_i = \max_k \{R_{i,k}\}$) and the maximum response time jitter ($RTJ_i = \max_k \{RTJ_{i,k}\}$) caused by EDF under three different deadline setting:

1. **Plain**: all tasks run with their maximum deadlines: $D_i = D_i^{max}$;
2. **Scaled**: all deadlines are uniformly scaled by the same factor according to the algorithm proposed by Balbastre et al. [5];
3. **New**: task deadlines are computed by the proposed algorithm according to given reduction factors.

In the first experiment, a simulation has been carried out with a set of 10 periodic tasks, having fixed utilization $U = 0.9$. The proposed algorithm has been applied to a group of four tasks with the same reduction factor ($\delta_i = 1$), while leaving the remaining tasks with their original deadlines ($\delta_i = 0$). In particular, the four tasks with the longest periods (from τ_7 to τ_{10}) have been selected for reduction. The worst-case response time and the response time jitter (RTJ) have been measured for each task and then averaged over 1000 simulation runs. A value $\varepsilon = 10^{-4}$ was used to find the best α.

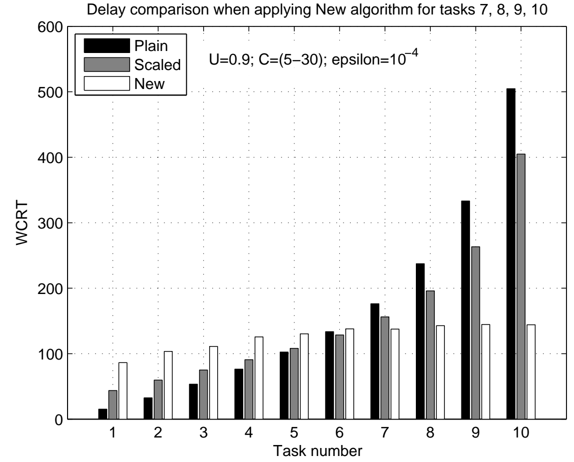


Figure 9. Worst-case response times when applying the proposed algorithm to task 7, 8, 9 and 10.

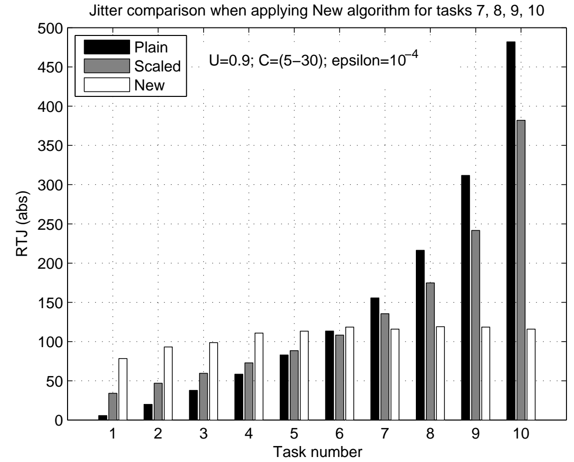


Figure 10. Response Time Jitter when applying the proposed algorithm to task 7, 8, 9 and 10.

Figures 9 and 10 respectively show the response time and jitter achieved for each individual task in this experiment. Note that, the X-axis shows the task identification number, where tasks are ordered by increasing period, so that task number 1 is the one with the shortest period. In particular, Figure 11 reports the jitter experienced by each task under the proposed algorithm, showing the 95% confidence interval around each average value.

As expected, the results show that restricting deadline reduction only to a subset of sensitive tasks allows better control of delay and jitter.

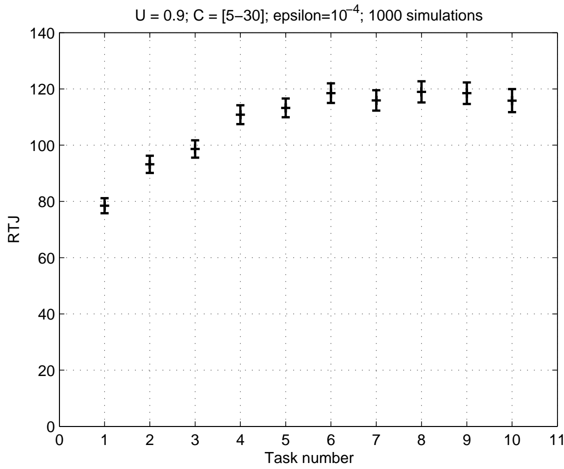


Figure 11. Confidence intervals (95%) of the jitter measures achieved in the first experiment under the proposed algorithm.

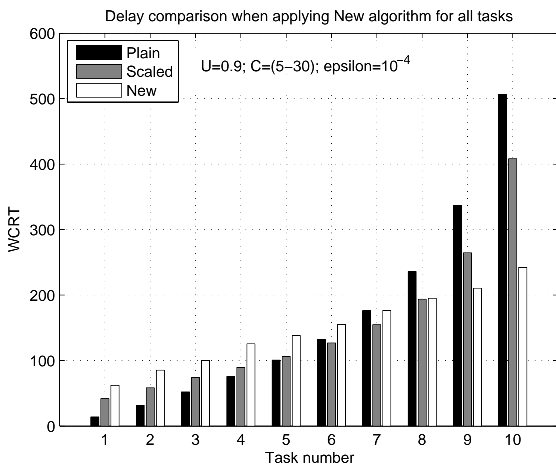


Figure 12. Worst-case response times when applying the proposed algorithm uniformly to all the tasks.

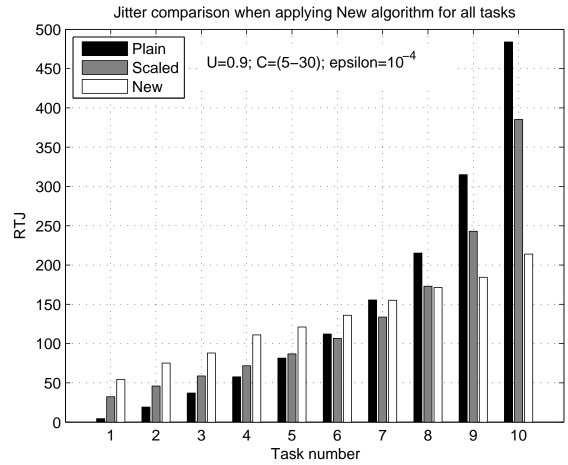


Figure 13. Response Time Jitter when applying the proposed algorithm uniformly to all the tasks.

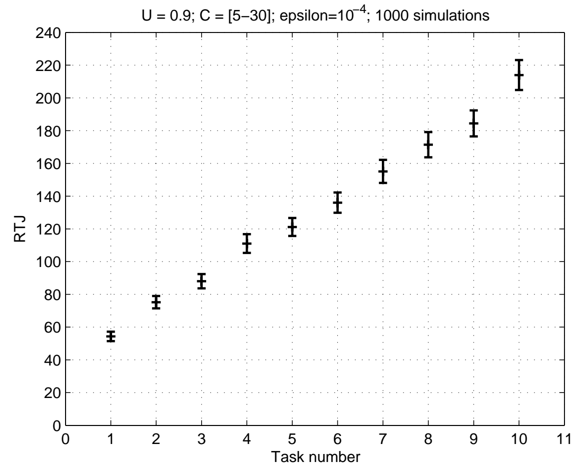


Figure 14. Confidence intervals (95%) of the jitter measures achieved in the second experiment under the proposed algorithm.

Also note that reducing all task deadlines by the same scaling factor (as done by the Scaled algorithm) has not a significant effect on jitter reduction with respect to the Plain scenario (where all deadlines are equal to the periods), thus justifying the need for adopting selective reduction factors.

A second experiment has been carried out to compare our algorithm against the uniform scaling algorithm [5] when all relative deadlines are uniformly scaled by the same reduction factor ($\delta_i = 1$ for $i = 1, \ldots, 10$). We have applied both methods on the same task set taken for the first experiment, using the same value of ε (10^{-4}).

As shown in Figures 12 and 13, our algorithm performs almost the same as the uniform scaling algorithm for tasks with short periods, whereas it performs slightly better for tasks with longer periods. All values plotted in the graphs represent the average over 1000 simulations, and the 95% confidential intervals on the average jitter achieved under the proposed algorithm are shown in Figure 14.

Finally, the performance of the proposed method has also been compared against the plain EDF scheduler and the scaled method when the task set has a lower utilization equal to $U = 0.6$.

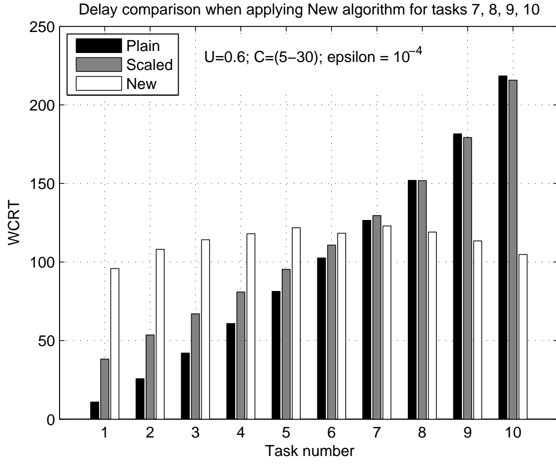


Figure 15. Worst-case Response Time when applying the proposed algorithm to task 7, 8, 9 and 10 ($U = 0.6$).

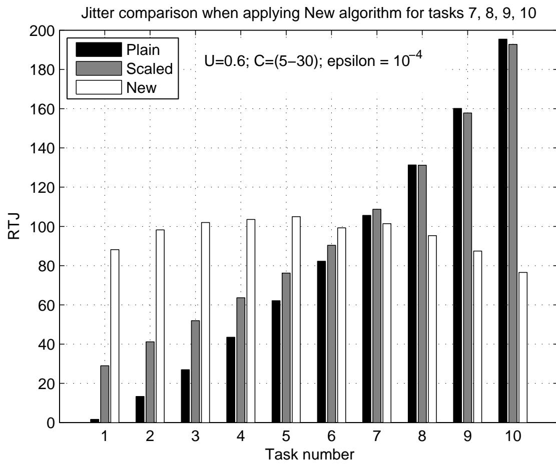


Figure 16. Response Time Jitter when applying the proposed algorithm to task 7, 8, 9 and 10 ($U = 0.6$).

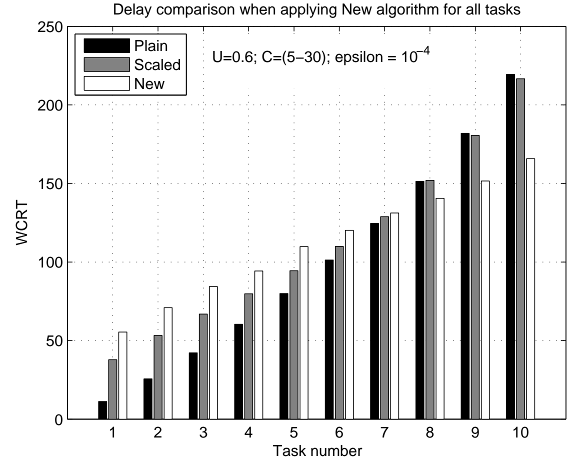


Figure 17. Worst-case Response Time when applying the proposed algorithm uniformly to all the tasks ($U = 0.6$).

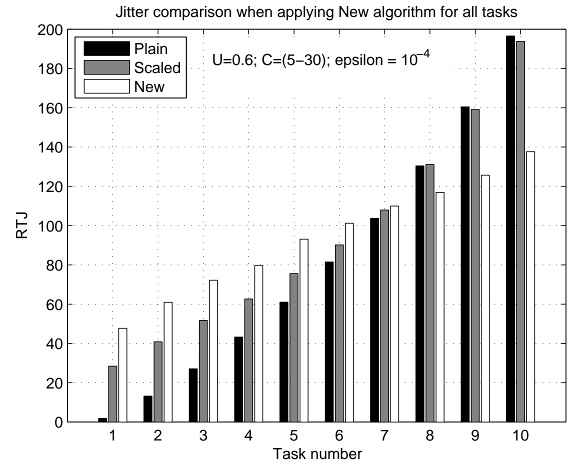


Figure 18. Response Time Jitter when applying the proposed algorithm uniformly to all the tasks ($U = 0.6$).

Figures 15 and 16 show the response time and jitter of each tasks when applying the new algorithm to tasks 7, 8, 9, 10 only, whereas Figures 17 and 18 show the response time and jitter of the tasks when applying the new algorithm uniformly to all the tasks.

All the experiments confirm that the proposed approach is able to reduce both delay and control jitter of specific control tasks according to desired scaling factors and under different load conditions. With respect to the plain EDF and uniform scaling algorithm, the proposed algorithm is more effective when the task set has a high utilization.

6 Conclusions

In this paper we presented a method for reducing the relative deadlines of a set of periodic tasks according to given reduction factors, $\delta_i \in [0, 1]$, denoting task sensitivity to jitter and delay. A value $\delta_i = 1$ denotes high sensitivity to delay and jitter, indicating that the task relative deadline can be reduced as much as possible, up to the minimum possible value which guarantees the task schedulability, whereas a value $\delta_i = 0$ denotes no sensitivity, indicating that the task relative deadline does not need to be modified.

Note that shortening the relative deadline decreases the admissible execution interval of a task, affecting both its response time and jitter.

Moreover, the proposed approach generalizes two other methods presented in the real-time literature for jitter reduction: the deadline minimization algorithm, independently developed by Hoang et al. [16] and by Balbastre et al. [5], and the uniform deadline scaling method, proposed by Balbastre et al. in the same paper. In fact, using the proposed approach, the relative deadline of a single periodic task τ_k can be minimized simply by setting $\delta_k = 1$ and all other reduction factors to zero. Similarly, a uniform reduction of all task deadlines can simply be achieved by setting all reduction factors to 1.

Experimental results confirm the effectiveness of the proposed approach, showing that deadline reductions are more significant when acting only on a subset of selected tasks.

As a future work, we plan to investigate the issue also under fixed priorities. Here, the deadline reduction algorithm cannot be trivially extended, because changing relative deadlines may also affect the priority order, and hence the feasibility test.

Acknowledgement

This work has been partly funded by the CERES research profile grant from The Knowledge Foundation.

References

[1] K. J. Astrom and B. Wittenmark, *Computer Controller Systems: Theory and Design*, Prentice-Hall, 1984.

[2] S. K. Baruah, R. R. Howell, and L. E. Rosier, “Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor,” *Real-Time Systems*, 2, 1990.

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier, “Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor,” *Proc. of the 11th IEEE Real-Time Systems Symposium*, Orlando, FL, USA, Dec. 1990.

[4] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari, “Scheduling Periodic Task Systems to Minimize Output Jitter,” *Proc. of the 6th IEEE International Conference on Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

[5] P. Balbastre, I. Ripoll, and A. Crespo, “Optimal Deadline Assignment for Periodic Real-Time Tasks in Dynamic Priority Systems,” *Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, Dresden, Germany, July 5-7, 2006.

[6] S. A. Brandt, S. A. Banachowski, C. Lin, and T. Bisson: “Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes,” *Proc. of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, USA, December 2003.

[7] G. Buttazzo and F. Sensini, “Optimal Deadline Assigment for Scheduling Soft Aperiodic Task in Hard Real-Time Environments,” *IEEE Transactions on Computers,* Vol. 48, No. 10, Oct. 1999.

[8] G. Buttazzo, M. Velasco, P. Marti, and G. Fohler, “Managing Quality-of-Control Performance Under Overload Conditions,” *Proc. of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)* , Catania, Italy, July 2004.

[9] G. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications - Second Edition, Springer, 2005.

[10] A. Cervin, “Integrated Control and Real-Time Scheduling,” Doctoral Dissertation, ISRN LUTFD2/TFRT-1065-SE, Department of Automatic Control, Lund, Sweden, April 2003.

[11] A. Cervin, B. Lincoln, J. Eker, K.-E. Arzn, and G. C. Buttazzo, “The Jitter Margin and Its Application in the Design of Real-Time Control Systems,” *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 25-27, 2004.

[12] M. Di Natale and J. Stankovic, “Scheduling Distributed Real-Time Tasks with Minimum Jitter,” IEEE Transactions on Computers, Vol. 49, No. 4, pp. 303-316, 2000.

[13] J. A. Stankovic, M. Spuri, K. Ramamritham, G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems - EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[14] D. Ferrari and D. C. Verma, “A Scheme for Real-Time Channel Establishement in Wide-Area Networks,” *IEEE Journal of Selected Areas in Communications,* Vol. 8, No. 3, pp. 368-379, Apr. 1990.

[15] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, “A New Kernel Approach for Modular Real-Time systems Development,” *Proc. of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, Netherlands, June 2001.

[16] H. Hoang, G. Buttazzo, M. Jonsson, and S. Karlsson, "Computing the Minimum EDF Feasible Deadline in Periodic Systems," *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[17] C. L. Liu and J. W. Layland, "Scheduling algorithms for Multipprogramming in Hard Real-Time traffic environment," *Journal of the Association for Computing Machinery,* Vol. 20, No. 1, Jan. 1973.

[18] P. Marti, G. Fohler, K. Ramamritham, and J.M. Fuertes, "Jitter Compensation for Real-time Control Systems," *Proc. of the 22rd IEEE Real-Time System Symposium*, London, UK, December 2001.

[19] P. Marti, "Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints," PhD Thesis, Department of Automatic Control, Technical University of Catalonia, Barcelona, Spain, July 2002.

[20] Q. Zheng and K. G. Shin, "On the Ability of Establishing Real-Time Chennals in Point-to-Point Packet-Switched Networks," *IEEE Transactions on Communications,* Vol. 42, No. 2/3/4, Feb./March/Apr. 1994.

Task Handler Based on (*m,k*)-firm Constraint Model for Managing a Set of Real-Time Controllers

Jia Ning, Song YeQiong, Simonot-Lion Françoise
LORIA – Nancy Université
Campus Scientifique – BP 239
54506 – Vandoeuvre lès Nancy France
{Ning.Jia,song,Francoise.Simonot}@loria.fr

Abstract

In this paper, we study how to schedule a set of real-time tasks where each task implements a control law. These tasks share a limited computing resource. The set of tasks can switch on line from one given configuration to another one depending on the working modes of the global application. This means that some tasks may appear while other ones may be no longer activated and that the WCET of some tasks may be modified. We propose a scheduling architecture for the handling of such task instances. At each mode switching, the task handler consider the new task parameters; then it determines on line a (m,k)-constraint based scheduling strategy to apply to each task; this strategy aims to selectively discard task instances so that the schedulability of tasks is guaranteed and the overall control performance is maintained at a high level.

1. Introduction

Let us consider an application composed of *n* physical sub-systems. Each sub-system is controlled by one dedicated controller that is implemented as a real-time task responsible for carrying out the control law computation for this sub-system. Therefore, a centralized implementation of all the controllers raises the problem of the schedulability of these *n* tasks. The tasks are generally considered as hard real-time tasks characterized by a fixed activation period (defined by the sampling period of the sub-system outputs) and a known worst-case execution times. Each instance of each task has to respect a deadline by which it is expected to complete its computation. However, due to the timing non-determinism of control application (non constant execution time, activation of new tasks, etc), using only worst-case execution time generally results in over sizing the necessary computing resource, and the overall control performance may not be optimal in the sense that they do not make a full use of the computing resource. Furthermore, many control systems are quite robust against variations in execution parameters such as a number of sample losses or a given variation in sampling period. Therefore, a scheduling approach that uses the current system configuration information to correctly adjust control task parameters in order to achieve higher resource utilization and better control performance is desirable.

The key to successful design of such a scheduling approach is the co-design of the controller and the resource management process which has attracted considerable attentions. In [15], an algorithm was proposed that selects the sampling periods for a set of control tasks so that the overall control performance was optimized under the schedulability constraints. The authors suppose that the cost function of each sub-system can be approximated by an exponential function. Due to the high computational cost of the algorithm, the proposed approach can only be used off-line. An exact off-line approach for sampling period selection was developed in [4] by supposing that the cost function is convex. To make on-line use of the proposed approach, the cost function is then supposed to be possibly approximated by a quadratic function so that the high computational costs can be reduced. In [3], a feedback-feedforward scheduling approach was proposed to improve the reaction speed of the feedback scheduling approach developed in [4] to a change in the computing resource load. Again, a linear function is proposed to approximate the cost function.

For all these above mentioned approaches, the cost functions of sub-systems are supposed convex (their theories also hold for the case where all the cost functions are concave) or can be approximated by a convex function. However, this assumption is restrict. In practice, one can not guarantee that the cost functions of sub-systems in the application are all convex, and since the cost functions of some control systems are not convex nor concave, they can not be well approximated by a convex function. Although it was shown in [1] that for some control systems, the cost function is quadratic for small sampling intervals, but the restriction on the choice of sampling period may lead to the non-schedulability of some tasks and therefore to the

degradation of the overall control performance. Furthermore, these approaches maintain the control performance optimality and control task schedulability by the regulation of the sampling periods of sub-systems. However, changing the period of a task may necessitate a change in the periods of related tasks as task periods are often carefully selected for an efficient exchange of information between relative tasks; in addition, the change in sampling period alters dynamics of the sub-system and leads to an unavoidable additional study for the approaches based on the regulation of the sampling periods.

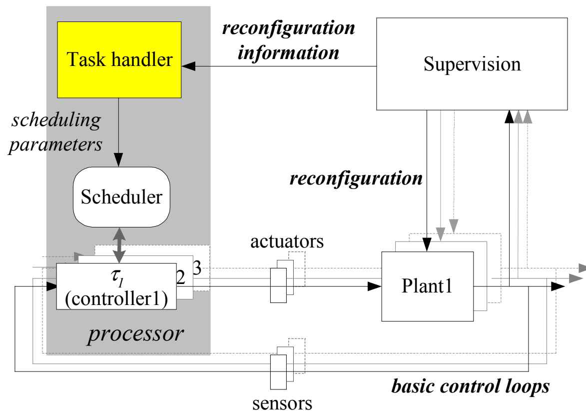


Figure 1. Overall system architecture

In this work, we consider a system architecture as shown in Figure 1. We suppose that a supervision function of all the controlled plants is implemented in a separate computer. The purpose of this function is to detect when the plants have to be controlled in a different way, meaning the change of control algorithm; it detects also when a plant is no more needed to be controlled or if a plant becomes operational and requires therefore the activation of a controller. Thanks to this supervision, the task handler can be notified when such a system configuration change leads to a new task set. Several controllers are implanted as real-time tasks in one processor, and each of them controls a physical plant. The number of control tasks and their execution times may change over time, for example when the overall system enters in a new working mode. At a system configuration change, the task handler is activated and it receives the information about these two execution parameters of control tasks from supervision component. Based on these information, it determines a set of scheduling parameters and transfer them to the scheduler. The scheduler then selectively discards the instances of control tasks according to these execution parameters so that the schedulability of control tasks is guaranteed and the overall performance of control application is maintained at a high level.

Specifically, the scheduling parameters transferred to scheduler are (m,k)-firm constraints [13][14], which indicate that the deadlines of at least m among any k consecutive instances of a control task must be met, where m and k are two positive integers with $m \leq k$ (the case where m=k is equivalent to the ideal case, which is noted by (k,k)-firm). Since the discarded instances will not be executing the control law, this tends to degrade the control performance. However, if a control system is designed to accept a control performance degradation until k-m deadlines misses among k consecutive task instances (this can be justified by the observation that most control systems can tolerate misses of the control law updates to a certain extent), the system can then be conceived according to the (m,k)-firm approach to offer the varied levels of control performance between (k,k)-firm (ideal case) and (m,k)-firm (worst case) with as many intermediate levels as the possible values between k and m. This results in a control system with graceful degradation of control performance.

The control performance can be described by different performance criterion that could be cost function, state error, maximum overshoot, settling time, etc. The proposed scheduling solution does not make any assumption on the type and property of control performance function. That is, it will hold whatever the control performance functions of sub-systems are all convex or can be approximated by a convex function. Furthermore, since the original sampling periods of control tasks will not be changed at a system configuration change, no period adaptation of the related tasks will be needed and the dynamics of sub-system will not be altered.

Notice that in [17][18], a scheduling algorithm is presented, which uses feedback information about the current workload in processor to regulate the deadline miss-ratio for a set of tasks disregarding the specific purpose of these tasks. Our approach aims to control the scheduling of a set of tasks while taking into account the fact that these tasks execute specific control laws. For short in [17][18], the QoS is an objective while in our proposal both QoS and QoC (quality of control) are targeted. Therefore our strategy is also based on the performance of the plant control. The scheduling objective is to explicitly maintain the overall control performance at a high level rather than maintain the deadline miss-ration at certain level.

The paper is organized as follows. A survey of related work is contained in section 2. Section 3 gives the task model of the control application and the schedulability analysis of control tasks under (m,k)-firm constraint. A formal description of the problem is presented in section 4. Section 5 gives the heuristic algorithm for computing the sub-optimal (m,k)-firm constraint for each control task. A numerical example of the proposed scheduling approach is presented in section 6. Finally, we summarize our work and show the perspectives.

2. State of the art: (*m,k*)-firm model and its use for control application

The previous work falls into two categories. The first one is the field of real-time scheduling based on (*m,k*)-firm constraint model.

In [14], a scheduling approach is presented for the general (*m,k*)-firm constraint model. A simple algorithm is used to partition the instances of each task in the system into two sets: mandatory and optional. All mandatory instances are scheduled according to their fixed priorities, while all optional instances are assigned the low priority. It follows that if all mandatory instances meet their deadlines, the (*m,k*)-firm constraint is satisfied. A sufficient and necessary condition for determining the schedulability of the mandatory instances is also given. In [13], it's proved that in general case, the problem of determining the schedulability of a set of tasks under (*m,k*)-firm constraint is NP-hard. In [6], we show that if the task instances are partitioned using the approach in [14], the distribution in the instance sequence of the mandatory instances corresponds to a mechanical word [11]. A series of mathematical tools for the schedulability analysis are also given using the theories of mechanical word.

The second area of previous work is the analysis of impact of (*m,k*)-firm constraint model on the control performance of a single control system. In [7], we presented a formal analysis that derived, for a one-dimensional control system, the *m* and *k* values that guarantees the control system stability. We also proposed an approach for deriving the controller minimizing the variance of the process state in order to minimize the deterioration in the control system behavior due to the control low update misses. This work is extended in [5] and in [8] to a multiple dimension control system. We showed, in [5] how to determine the maximum value of *k* for a control system so that one can get as many varied levels of control performance as possible subject to the stability of control system. In [8], we gave a general method to derive the optimal LQ-controller under (*m,k*)-firm constraint, and based on the works in [6], we showed that for a single control task under a given (*m,k*)-firm constraint, the control performance of the corresponding control system is sub-optimal if the task instances are partitioned and scheduled using the scheduling approach in [14].

3. Task Model and Schedulability

In this section, we first give the task model of the real-time control application under study. Then, a sufficient condition for determining of schedulability of tasks set is given.

3.1 Task Model

Let an application be composed of *n* periodic control tasks, τ_1 τ_2 .. τ_n, arranged in decreased order of their priorities. The following timing parameters are defined for each task τ_i:

- T_i: the time interval between two consecutive instances of τ_i, referred to as its period;
- C_i: the maximum time needed for completing the execution of each instance of τ_i, referred to as its execution time;
- D_i: the deadline of each instance of τ_I; we consider that the deadline is equal to the period of the task;
- m_i and k_i: the (*m,k*)-firm constraint for τ_i with $m_i \leq k_i$; it means that at least the deadlines of m_i out of k_i consecutive instances of the task must be met.

Furthermore, we assume that the algorithms of the control laws are independent in the sense that they do not share any resources except the processor; therefore the control tasks are assumed to be preemptive.

3.2 Schedulability under (*m,k*)-firm Constraint

The problem of determining the schedulability of a set of tasks under (*m,k*)-firm constraints has been proved in [13] to be NP-hard; however, when the tasks are scheduled using the algorithm proposed in [14], the schedulability of tasks can be explicitly judged. Furthermore, as stated formerly, if the instances of a control task under a given (*m,k*)-firm constraint are partitioned and scheduled by the scheduling approach in [14], the control performance of the corresponding control system is sub-optimal. Therefore we adopt in this work the scheduling algorithm proposed in [14] for the scheduling of the set of control tasks.

Concretely, the instances of each task are partitioned into two sets: the mandatory instances and the optional instances. The problem is to determine for given m and k which instances in a sequence of instances are mandatory. We propose in [6] to fit the distribution of mandatory instances using theory of mechanical word. Under this approach, an instance of τ_i, activated at time aT_i, for a = 0,1,… is classified as mandatory if the following condition is verified

$$a = \left\lfloor \left\lceil \frac{am_i}{k_i} \right\rceil \frac{k_i}{m_i} \right\rfloor \quad (1)$$

and as optional, otherwise. For example, if τ_i is under (3,5)-firm constraint, the condition (1) is verified for a=0+αk, 1+αk and 3+αk ($\alpha \in N$) and not verified for a=2+αk and 4+αk. Therefore the instances activated at 0,1,3,5, etc are mandatory while those activated 2,4,7, etc are optional. Notice that using the classification equation (1), there are exactly *m* mandatory instances and *k-m* optional instances among any *k* consecutive instances, and the mandatory instances are uniformly distributed in the instance sequence. The reader interested in could refer to [6] for a more detail information about the task instance classification theory.

The control tasks are scheduled using the fixed priority policy. The mandatory instances of all the tasks are assigned the rate-monotonic priorities [10]. That is, the mandatory instances of τ_i are assigned a higher priority than the mandatory instances of τ_j if $T_i<T_j$. The optional instances are assigned the lowest priority.

A sufficient and necessary schedulability condition for the above scheduling strategy is given in [14]. However, the condition contains a timing non-deterministic term, this prevents it from a application in our optimization routine presented below. We therefore give a sufficient schedulability condition:

Theorem 1. *Given a task set ($\tau_1, \tau_2 ... \tau_n$) such that $T_1<T_2< ..<T_n$. Let:*

$$n_{ij} = \left\lceil \frac{m_j}{k_j} \left\lceil \frac{T_i}{T_j} \right\rceil \right\rceil \qquad (2)$$

If $C_i + \sum_{j=1}^{i-1} n_{ij} C_j \le T_i$ *for all* $1 \le i \le n$, *then the (m_i, k_i)-firm constraint of each task* τ_i *is satisfied.*

Proof. In [6], we proved that the task instance partition algorithm results in the most mandatory instances from [0, t] compared with those in any other interval of the same length t. Therefore, to analyze the schedulability of a set of tasks, it's enough to study if the first instance of each task respects its deadline. From [6], we know that (2) gives the number of the mandatory instances of task τ_j before instant T_i, therefore, if $\left(C_i + \sum_{j=1}^{i-1} n_j C_j \right) / T_i \le 1$, then the first instance of τ_i will complete prior to its deadline. The theorem follows if the results holds for all i.

□

Note that the above schedulability condition is sufficient and necessary if the period of a task is multiple of the periods of all the lower priority tasks. In the other case, it degenerates to a sufficient condition.

4. Problem Formulation and scheduling architecture

Recall that we consider a real time control application composed of several control tasks that share the same processor. It's assumed that the control tasks in the application can switch between different modes. Going from one mode to another mode can lead to consider another task set configuration: the execution time can be different for some tasks, control tasks can be shutdown and new control tasks can be activated. We aim to implement a task handler that, at each change in the task set configuration, guarantees the schedulability of control tasks and keeps the overall control performance at a high level.

Based on the results in [5], we suppose that the value of k_i of each τ_i has been carefully chosen and is constant for each mode. The value of m_i is dynamically chosen in [1 .. k_i] on line. Specifically, for each control task τ_i, each possible m_i is associated with a level of control performance v_{ij}. We assume that a larger v_{ij} corresponds to a better control performance. As we do not make any assumption on the control performance property, the control performance is not necessarily improved when the value of m_i is increased [5]. Furthermore, theorem 1 shows that the increase of the value of m_i may increase the resource requirement. From task scheduling point of view, there is no reason to select a larger value for m_i with lower control performance. Therefore, only the values of m_i which give a better control performance are considered.

So, let us consider that m_i can take l_i different values, noted as m_{ij} for $j\in$ [1.. l_i] where $l_i \le k_i$. These values are arranged in increasing order, that is if $j' < j''$, then $m_{ij'} < m_{ij''}$ and therefore $v_{ij'} < v_{ij''}$. The aim of the task handler is to find, for each τ_i, a value m_{ij} for $j\in$ [1.. l_i] so that the sum of v_{ij} for $i\in$ [1.. n] is maximized, and the schedulability of control tasks is guaranteed. This is formulated as the following optimization problem:

to determine the sequence $x_{i1}, x_{i2}, .., x_{il_i}$ for each task τ_i, $i=1,..,n$

that maximizes $$\sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} v_{ij} \qquad (3)$$

with $x_{ij} \in \{0,1\},\ \sum_{j=1}^{l_i} x_{ij} = 1,\ i=1,..,n,\ j=1,..,l_i$

and such that

$$C_i + \sum_{j=1}^{i-1} \left\lceil \frac{\sum_{p=1}^{l_i} x_{jp} m_{jp}}{k_j} \left\lceil \frac{T_i}{T_j} \right\rceil \right\rceil C_j \le T_i,\ i=1,..,n, \qquad (4)$$

If the control performance is represented by the control cost (LQ cost for example), then a smaller v_{ij} gives a better control performance. The optimization problem thus becomes a minimization problem. However, it can be easily transformed as a maximization problem by take the additive inverse of v_{ij} for each τ_i.

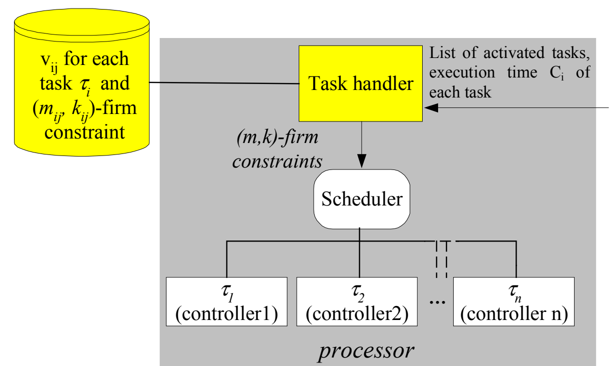


Figure 2. scheduling architecture

The scheduling architecture is given in Figure 2, which is a close-up of figure 1. At a task set configuration change, the task handler receives the information about the number of tasks sharing the processor and the actual execution time C_i of each task, and deduces the new (m_i,k_i)-firm constraint for each control task by resolving the optimization problem (3). The scheduler then schedules the tasks according to these (m,k)-firm constraints.

The control performance level v_{ij} corresponding to a (m_{ij},k_{ij})-firm constraint for task τ_i can be determined on-line or off-line. However, this determination is very time-consuming for some control performance criterion (e.g. the control performance criterion used in [8]). For this reason, The control performance level v_{ij} are calculated off-line and arranged in a table that is consulted by the task handler when solving the problem (3).

5. Performance Optimization

In this section, we first show that the optimization problem (3) is a NP-hard problem, then, based on the algorithm proposed in [9], a computationally cheaper heuristic algorithm is proposed for finding a sub-optimal solution.

5.1 Solution of the Optimization Problem

The optimization problem enounced in (3) is qualified as the multiple-choice multi-dimension knapsack problem (MMKP) which is defined as following:

Suppose there are n groups (stacks) of items. Group i has l_i items. Item j of group i has value v_{ij}, and requires resources given by vector $r_{ij}=(r_{ij1},r_{ij2},..,r_{ijm})$. The amounts of available resources are given by $R=(R_1,R_2,..,R_m)$. The MMKP is to pick exactly one item from each group in order to maximize the total value of the pick, subject to the resource constraints. Formally, the MMKP is expressed as follows:

$$\text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{l_i} x_{ij} v_{ij}$$

such that

$$\sum_{i=1}^{n}\sum_{j=1}^{l_i} x_{ij} r_{ij} \le R_k, \quad k = 1,..,m,$$

$$x_{ij} \in \{0,1\}, \ \sum_{j=1}^{l_i} x_{ij} = 1, \ i = 1,..,n, \ j = 1,..,l_i$$

MMKP is one of the harder variants of the 0-1 knapsack problem [12][16]. The problem is proved to be NP-hard problem. Hence algorithms for finding the exact solution of MMDP are not suitable for application in real time decision-making application.

A computationally cheaper heuristic algorithm (HEU) for MMKP is presented in [9]. For our optimization problem, the proposed algorithm is to find a feasible solution at first, that is, select m_{ij} for each τ_i while satisfying the constraints enounced in (4), and then iteratively improve the solution by replacing, for each τ_i, m_{ij} by a $m_{ij'}$ corresponding to a better performance $v_{ij'}$ while keeping the constraints (4) satisfied. If no such solution can be found, the algorithm tries an iterative improvement of the solution which first replaces m_{ij} for a task τ_i which is not schedulable with the value m_{ij} (constraint (4) violated), and then replace $m_{i'j}$ of $\tau_{i'}$ for all $i \neq i'$ with $m_{i'j'}$ corresponding to a worse performance $v_{i'j'}$. The iteration finishes when no feasible solution can be find. It's shown in [9] that the algorithm HEU is efficient (the solution given by the algorithm is within 6% of the optimal value) and suitable for an on-line use for real-time application (the computation time is less than one millisecond if the number of tasks does not exceed 30 on a 700 MHz Pentium III processor).

As in our problem setup, the values m_{ij} and v_{ij} for $j \in [1..\ l_i]$ are arranged in increasing order for each τ_i, therefore in contrast with HEU in which an infeasible solution may be selected at first and iterations are needed to make it feasible, we modify HEU by always picking the lowest value of m_i of each τ_i at first (if the solution is infeasible in this case, no other solution will be feasible). Furthermore, the algorithm HEU tries to find, after the first step, a better solution requiring less resource consummation which, however, does not exist in our model (augmentation of m increases the processor utilization), therefore this property also help us to delete a unprofitable researching procedure in HEU. These modifications can reduce the execution time of the algorithm. The modified algorithm is presented in Algorithm 1.

In this algorithm, replacing m_{ij} by $m_{ij'}$ is called an *exchange*. An exchange giving a better overall control performance is called *upgrade* while an exchange degrading the overall control performance is called *downgrade*. An exchange is called *feasible* if the solution after the exchange is feasible, otherwise it is called *infeasible*.

Algorithm 1. Algorithm for finding the value of the parameter m of (m,k)-firm constraint for all tasks

// Symbols and formalization:
// n : number of the tasks
// m_i and k_i: the (m_i,k_i)-firm constraint of task τ_i
// l_i : number of the possible values for m_i
// m_{ij}: the jth possible value of m_i
//T_i: period of task τ_i; C_i : execution time of task τ_i
// $\rho = (\rho_1, \rho_2,.., \rho_n)$ denotes the current solution, where τ_i gives the index of the value of m_i in $[1..l_i]$
// $\rho|Z$: solution vector after exchange Z from ρ
// L: current resource requirement vector

// $U(\rho)$: total control performance with the solution vector ρ, with $U(\rho)=\sum_{i=1}^{n} v_{i\rho_i}$

// $X=(i,j)$ denotes an exchange where the m_{ij} is selected instead of $m_{i\rho_i}$

1. **Start with a feasible solution.**

   Set $\rho_i = 1$ for all $i \in [1,..,n]$.

   Compute
$$L=\begin{pmatrix} C_1 \\ C_2+\sum_{j=1}^{1}\left\lceil \frac{m_{j1}}{k_j}\left\lceil \frac{T_2}{T_j}\right\rceil\right\rceil C_j \\ \vdots \\ C_n+\sum_{j=1}^{n-1}\left\lceil \frac{m_{j1}}{k_j}\left\lceil \frac{T_n}{T_j}\right\rceil\right\rceil C_j \end{pmatrix}$$

   Find α such that $L_\alpha / T_\alpha = \max_{i=1,2,..n} \frac{L_i}{T_i}$.

   if $L_\alpha / T_\alpha \le 1$, go to step 2, else exit the procedure with "no feasible solution".

2. **Iterative improvement using feasible upgrades.**

   Define
$$\Delta\alpha(\rho,i,j)=\frac{\sum_{l=i+1}^{n}\left(\left\lceil \frac{m_{i\rho_i}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i-\left\lceil \frac{m_{ij}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i\right)L_l}{|L|}$$

   and
$$\Delta p(\rho,i,j)=\frac{v_{i\rho_i}-v_{ij}}{\Delta\alpha(\rho,i,j)}$$

   Find feasible upgrade $X'=(\delta,\eta)$ that maximizes $\Delta p(\rho,i,j)$.

   If X' is found and $\Delta p(\rho,i,j)>0$, set $r = (r|X')$ and repeat step 2.

3. **Iterative improvement using upgrades followed by one or more downgrade(s).**

   3.1

   Define
$$\Delta t(\rho,i,j)=\sum_{l=i+1}^{n}\frac{\left\lceil \frac{m_{i\rho_i}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i-\left\lceil \frac{m_{ij}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i}{T_l-L_l}$$

   and
$$\Delta p'(\rho,i,j)=\frac{v_{i\rho_i}-v_{ij}}{\Delta t(\rho,i,j)}$$

   Find an upgrade $Y=(\delta',\eta')$ maximizing $\Delta p'(\rho,i,j)$, set $\rho'=(\rho|Y)$

   3.2

   Define
$$\Delta t'(\rho,i,j)=\sum_{l=i+1}^{n}\frac{\left\lceil \frac{m_{i\rho_i}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i-\left\lceil \frac{m_{ij}}{k_i}\left\lceil \frac{T_l}{T_i}\right\rceil\right\rceil C_i}{L_l}$$

   and
$$\Delta p''(\rho,i,j)=\frac{v_{i\rho_i}-v_{ij}}{\Delta t'(\rho,i,j)}$$

   Find a downgrade $Y'=(\delta'',\eta'')$ maximizing $\Delta p''(\rho',i,j)$ such that $U(\rho'|Y')>U(\rho)$.

   If Y' is found and $(\rho'|Y')$ is feasible, set $\rho=(\rho'|Y')$ and go to step 2.

   If Y' is found and $(\rho'|Y')$ is not feasible, set $\rho'=(\rho'|Y')$ and go to step 3.2.

5.2 Complexity analysis of Algorithm 1

As stated [9], we suppose $l_1=\ldots=l_n$ for simplifying of the complexity analysis. In the first step, the solution is set directly to be $\rho=(1,1,..,1)$ which allows to reduce the computational complexity of the step 1 to $O(n^2)$ instead of $O(n^2(l-1)^2(n-1))$ in HEU. In the other steps, the modification that we proposed doesn't change their worst-case computational complexities in the original algorithm that are $O(n^2(l-1)^2(n-1))$. As it is mentioned in [9], the combined complexity of steps 1 and 2 gives the computational behavior of the algorithm, the worst-case computational complexity of the algorithm 1 is thus $O(n^2(l-1)^2(n-1))$.

6. Numerical Example

In this section, we present a numerical example to illustrate the proposed scheduling approach. We consider a cart-control system whose objective is to control the position of a cart along a rail according to a position reference. We study the problem of simultaneously controlling four cart systems, $Cart_1$, $Cart_2$, $Cart_3$, $Cart_4$. To illustrate the benefit of our approach, both the traditional scheduling approach and ours are evaluated.

6.1 Plants and Controllers

The continuous model of the cart system is given by:

$$dx=\begin{bmatrix}0 & 1 \\ 0 & \frac{-11.4662}{M}\end{bmatrix}xdt+\begin{bmatrix}0 \\ \frac{1.7434}{M}\end{bmatrix}udt+dv_c$$

where M is the mass of the cart; v_c is the process noise with incremental covariance $R_{1c}=\begin{bmatrix}3.24 & -1.8 \\ -1.8 & 1\end{bmatrix}$. The four cart systems, $Cart_1$, $Cart_2$, $Cart_3$, $Cart_4$ have different masses: $M_1 = 1.5$, $M_2 = 1.2$, $M_3 = 0.9$, $M_4 = 0.6$, given in kg. In the following, the controller of $Cart_i$ is denoted $Controller_i$. Its sampling periods (second) is h_i (resp. 0.007, 0.0085, 0.01, 0.0115).

The control performance is measured using a quadratic cost criterion:

$$J=\lim_{N\to\infty}\frac{1}{N}E\left(\int_0^N x^T(t)Qx(t)+u^T(t)Ru(t)dt\right) \quad (5)$$

where

$$Q=\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \text{ and } R=0.00006$$

The value of k for each cart system is chosen using the approach proposed in [5] to guarantee the system stability. In [5], we identified that the maximum value of k calculated for $Cart_1$ and $Cart_2$ (masses 1.5kg and 1.2kg) is greater than 10; nevertheless, in this paper, we consider, for an easier demonstration, values of k that are less than 10 and for which, the systems remain stable. Furthermore, note that since the tasks are assigned the rate-monotonic priority, the task with the largest period has the lowest priority, its execution has no influence on the other tasks. Therefore the task will be executed without task instance drop, or in other words, it is executed under (k,k)-firm constraint. So the value of k for $Cart_4$ has no use. Finally, the parameters k for the other $Cart_1$, $Cart_2$ and $Cart_3$ are: k = [5,8,10]. For each cart system, the value of m may vary within [1..k].

The parameters of LQ-controller that minimizes the quadratic cost criterion (5) and the minimum cost obtained for each possible (m,k)-firm constraint are calculated using the approach presented in [8]. When the (m,k)-firm constraint is changed for a task during running time, the controller parameters are replaced by the ones corresponding to the new (m,k)-firm constraint. To allow for fast changes between different (m,k)-firm constraints, the parameters are calculated off-line for each (m,k)-firm constraint and stored in a table.

6.2 Experiment Setup

The simulation model is created using MATLAB/Simulink and the TrueTime toolbox [2].

The execution time of each control task is approximately 3 ms, and that of the task handler varies with the amount of tasks and the value of k for each tasks. For this example, the execution time of task handler is fixed as 2.5 ms. The task handler is assigned the highest priority, and the priorities of control tasks are assigned according to the rate-monotonic period assignment policy.

During the simulation, the release time of a task is always set to be the current release time plus the task period. Therefore, for a task, if a instance misses its deadlines, the release time of the following instance will have a release time back in time.

The experiment is done for both traditional scheduling approach which schedules the task instances without taking into account the processor overload and ours. At $t = 0$, $Controller_1$ and $Controller_2$ are on, while $Controller_3$ and $Controller_4$ are off. At t=1, $Controller_4$ switches on, and at t=2, $Controller_3$ also switches on. At starting of each controller, a position reference (0.5 cm from current position) is entered to each controller.

The accumulated cost for $Controller_i$ is used to measure the performance of a controller, which is given by:

$$J_i(t)=\int_0^t x^T(s)Qx(s)+u^T(s)Ru(s)ds \quad (6)$$

The cost (6) is calculated at each time instant. A good control performance is therefore represented by a smooth increase in cost.

Finally, for comparing the scheduling results obtained with different scheduling approaches, the four plants are subjected to identical sequence of process noise in the two scheduling cases.

6.3 Simulation results

The simulation results in the two different scheduling approach are presented and discussed below.

6.3.1 Traditional scheduling approach

The accumulated costs of each controller (*J1, J2, J3,J4*) are shown in Figure 3. The close-up of schedule at t=1 and t=2 are shown in Figure 4 and Figure 5.

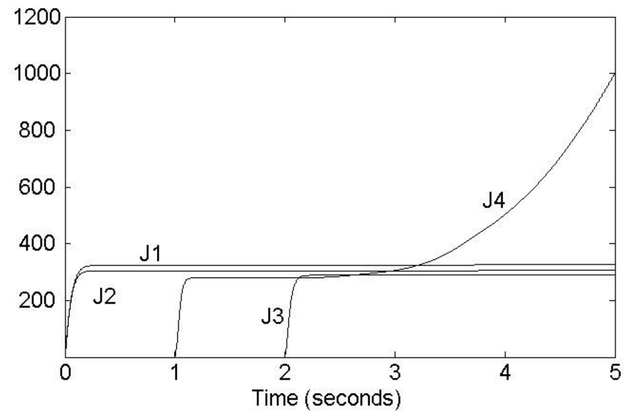


Figure 3. Accumulated costs under traditional scheduling

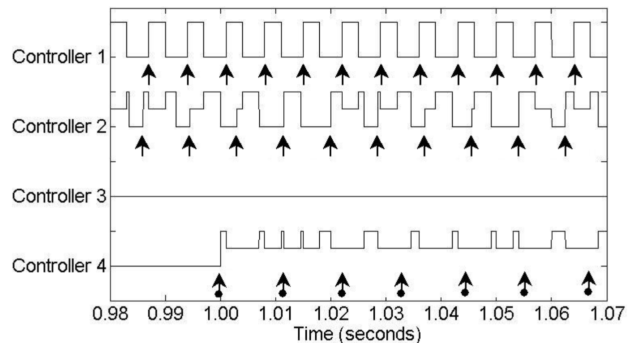


Figure 4. Close-up of schedule at *t*=1 under traditional scheduling

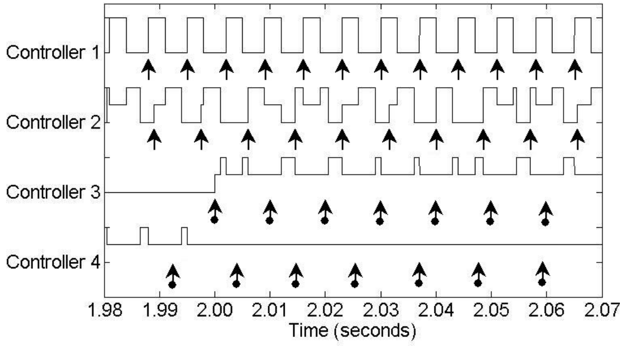


Figure 5. Close-up of schedule at *t*=2 under traditional scheduling

We will comment below the system evaluations represented on these figures. In figures 4 and 5, the state of each task is given. For each task, the state can take 3 values as represented in figure 6:

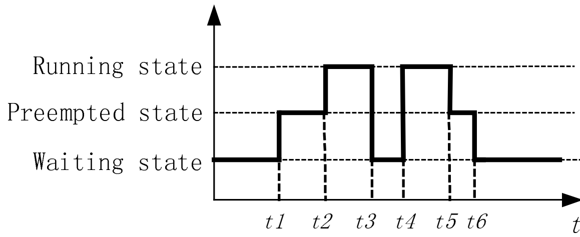


Figure 6. Task state representation

1. Running : being executed by the processor (e.g. state between *t2* and *t3*, *t4* and *t5*);
2. Preempted : the execution is preempted by other task (e.g. between *t1* and *t2*, *t5* and *t6*);
3. Waiting: the task waits for an activation : (e.g. state before *t1*, between *t3* and *t4,* after *t6*).

The arrows indicate the arrivals of task instances. Arrows with a solid point represent arrivals of task instances that miss their deadline.

During the observation, τ_1 and τ_2 are scheduled without any deadline violation. The accumulated costs increase steadily and the two systems perform well.

At t=1, τ_4 starts to execute. Under the preemption due to the execution of τ_1 and τ_2, the deadline of τ_4 is violated. However, the control performance is acceptable before the activation of τ_3 (at t=2) since the task instances are executed although their deadlines are all missed.

At t=2, τ_3 is turned on. Together with τ_1 and τ_2, the preemption due to these tasks makes the execution of τ_4 impossible. As a result, the cart system $Cart_4$ becomes instable (see *J4* in figure 3 for $t > 2$). Note that despite the execution preemption due to τ_1 and τ_2, the performance of $Controller_3$ does not decrease rapidly as for τ_4 because the task instances are executed although their deadlines are all missed.

6.3.2 Scheduling approach with (*m,k*)-firm constraint regulation

The simulation results obtained with the proposed approach are given in this subsection.

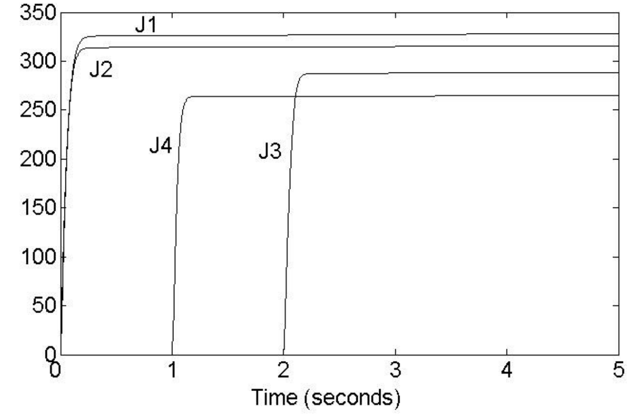


Figure 7. Accumulated costs under proposed scheduling approach

The accumulated costs for the four controllers are shown in Figure 7. The close-up of schedule at t=1 and t=2 are shown in Figure 8 and Figure 9. The signification of signs are the same as for the traditional scheduling case. Furthermore, the grey arrows mean that the task instances activated at the indicated instants are classified as optional instances and therefore they are not executed.

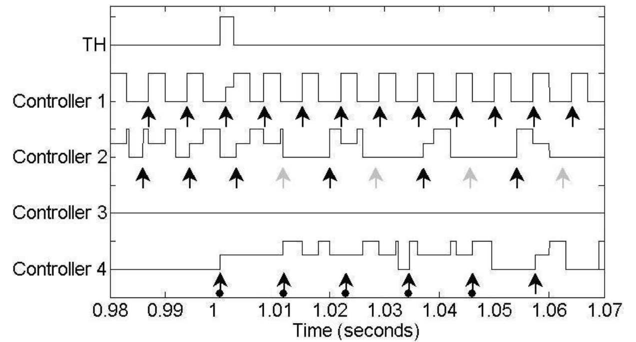


Figure 8. Close-up of schedule at *t*=1 under proposed scheduling approach

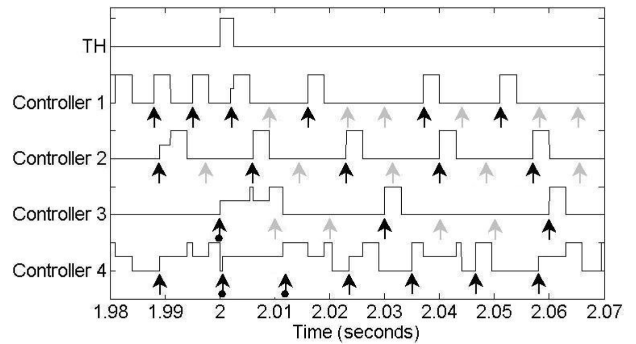


Figure 9. Close-up of schedule at *t*=2 under proposed scheduling approach

At the system starting, only τ_1, and τ_2 are activated. Their (m,k)-firm constraint are (k,k). Then, the system configuration change is detected at t=1 and the task handler is immediately activated. At t=1.025, the (m,k)-firm constraint of τ_2 are adjusted to (4,8)-firm constraint, and that of τ_1 is remained as (k,k). The overload condition is therefore avoided. The same thing is repeated at t=2, the (m,k)-firm constraints of the task τ_1, τ_2, τ_3 are adjusted to respectively (2,5), (4,8), and (3,10)-firm constraint; τ_4 is under (k, k)-firm constraint (see end of section 6.1). Note that the task deadline violations after the (m,k)-firm constraint adjustments in the two figures are due to the transient overload, however, the overload condition is removed rapidly.

Compared with the traditional scheduling approach, the controllers perform much better. The $Controller_4$ is stable and the control costs for $Controller_3$ and $Controller_4$ do not exceed 300 at t=5.

7. Conclusion

A scheduling architecture based on the (m,k)-firm constraint model is proposed. When a change in system configuration is detected, the task handler determines a strategy for selectively discarding task instances for each control task so that the schedulability of control tasks is guaranteed and the overall control performance is maintained at a high level.

Compared with formerly defined feedback scheduling approaches for control tasks, the proposed solution does not depend on the type and property of the control performance. That is, whatever the functions describing the control performance are convex, the proposed approach can always keep the overall control performance at high level while guarantying the schedulability of control tasks. Furthermore, at a system configuration change, the proposed solution avoids the change in the periods of related tasks, and does not alter the dynamics of the sub-system.

References

[1] Aström, K. J., “On the choice of sampling rates in optimal linear systems”, Technical Report RJ-243, San José Research Laboratory, IBM, San José, California.

[2] Cervin, A., Eker, J., Bernhardsson. B., and Årzén, K.-E., "Feedback feedforward scheduling of control tasks," *Real-Time System.*, vol. 23, no. 1-2, pp. 25--53, 2002.

[3] Cervin, A., Henriksson, D., Lincoln, B., Eker, J., d Årzén, K.-E., "How does control timing affect performance" IEEE Control Systems Magazine, 23:3, pp. 16-30, June 2003

[4] Eker, J., Hagander, P., and Årzén, K.E., “A Feedback Scheduler for Real-Time Controller Tasks”, *Control Engineering Practice*, vol. 12, no.8, p. 1369-1378, 2000.

[5] Felicioni. F., Jia. N., Song. Y.Q and Simonot-Lion. F., “Impact of a (*m,k*)-firm data dropouts policy on the quality of control”, *6th IEEE International Workshop on Factory Communication Systems - WFCS’2006*, Torino, Italy, 2006.

[6] Jia, N., Hyon, E., Song, Y.Q., “Ordonnancement sous contraintes (*m,k*)-firm et combinatoire des mots”, *13th International Conference on Real-Time Systems*, RTS'2005, Paris, France, 2005.

[7] Jia, N., Song, Y.Q., and Lin, R.Z., “Analysis of networked control system with packet drops governed by (m,k)-firm constraint”. *Proc of the 6th IFAC international conference on fieldbus systems and their applications (FeT'2005)*, Puebla Mexico, 2005.

[8] Jia, N., Song, Y.Q., and Simonot-Lion, F., “Optimal LQ-controller design and data drop distribution under (*m,k*)-firm constraint”, submitted to ECC’2007, available as Technical report at LORIA.

[9] Khan, S., LI, K.F., Manning, E.G. and Akbar, M. “Solving the knapsack problem for adaptive multimedia systems”, *Studia Informatica universalis*, Vol. 2, No. 1, pp. 157-178, 2002.

[10] Liu. C.L., and Layland. J.W., “Scheduling algorithm for multi-programming in a hard real-time environment”, *J.ACM*, vol. 20, pp.46-61, 1973.

[11] Lothaire., M. “Algebraic Combinatorics on Words”, *Cambridge University Press*, 2002.

[12] Pisinger, D., “Algorithms for Knapsack Problems”, Ph.D. thesis, DIKU, University of Copenhagen, Report 95/1, 1995.

[13] Quan, G., and Hu, X., “Enhanced Fixed-priority Scheduling with *(m,k)-firm* Guarantee ”, *Proc. Of 21st IEEE Real-Time Systems Symposium*, pp.79-88, Orlando, Florida, USA, 2000.

[14] Ramanathan, P., “Overload management in Real-Time control applications using (m,k)-firm guarantee “, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 549-559, 1999.

[15] Seto, D., Lehoczkyn, J. P., Sha, L. and Shin, K. G., “On task schedulability in real-time control systems”, *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 13-21, Washington, DC, USA, 1996.

[16] Silvano Martello, Paolo Toth (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons. ISBN 0-471-92420-2, 1990.

[17] Stankovic, J., Lu, C., Son, S. H., and Tao, G. “The case for feedback control real-time scheduling” In Proceedings of *the 11th Euromicro Conference on Real-Time Systems*, pp. 11-20.

[18] Stankovic, J., He, Tian., Abdelzaher, Tarek F., Marley, Mike., Tao, Gang., Son, Sang H and Lu Chenyang. “Feedback Control Scheduling in Distributed Systems”. In *22nd IEEE Real-Time Systems Symposium(RTSS 2001)*, December 2001.

Networks and distributed systems

Interface Design for Real-Time Smart Transducer Networks – Examining COSMIC, LIN, and TTP/A as Case Study

Wilfried Elmenreich
Institute of Computer Engineering
Technische Universität Wien
Austria
wil@vmars.tuwien.ac.at

Hubert Piontek
Dep. of Embedded Systems/RT Systems
Universität Ulm
Germany
hubert.piontek@uni-ulm.de

Jörg Kaiser
Institut für Verteilte Systeme
Universität Magdeburg
Germany
kaiser@ivs.cs.uni-magdeburg.de

Abstract *– This paper analyzes and discusses the interface models of the three real-time smart transducer networks COSMIC, LIN, and TTP/A.*

The COSMIC architecture follows a publish/subscribe model, where the producing smart devices broadcast their event data on the basis of a push paradigm. Subscribers receive data in form of a message-based interface.

LIN follows a strict pull principle where each message from a device node is requested by a respective message from a master. Applications have a message-based interface in order to receive and transmit data.

The nodes in a TTP/A network derive its sending instants from predefined instants in time. TTP/A maps communicated data into an Interface File System (IFS) that forms a distributed shared memory.

1 Introduction

The availability of cheap microcontrollers and network solutions has enabled distributed architectures with networked smart transducer devices. The hardware for a smart transducer consists of a physical sensor or actuator, a microcontroller or FPGA with on-chip memory and analog I/O, and a network interface. The software in the microcontroller contains transducer-specific routines, like de-noising, linearization, and feature extraction, and a communication protocol establishing a standardized interface to the smart transducer. This interface provides access to the transducer values, like sensor measurements or actuator set values, as well as configuration and management data (calibration data, error logs, etc.). In order to support an automatic (plug-and-play) or semi-automatic configuration, a smart transducer may also host an electronic description, i. e., a machine-readable datasheet describing its features and interfaces.

Real-time and bandwidth requirements make the design of an interface to a smart transducer a difficult task. This paper addresses the specific requirements and issues of interface design for smart transducers and examines three architectures for real-time smart transducer networks, i. e., the Cooperating SMart devices (COSMIC) middleware, the Local Interconnect Network (LIN), and the Time-Triggered Communication Protocol for SAE class A applications (TTP/A).

The paper is structured as follows: Section 2 states the basic concepts and requirements for smart transducer interface design. Section 3 describes communication model, interface design and node description approach for the COSMIC middleware. Accordingly, Section 4 and Section 5 examine the LIN and TTP/A approach. Section 6 elaborates common features and differences of the three approaches. The paper is concluded in Section 7.

2 Interface Concepts

A smart transducer interface can be decomposed into several sub-interfaces with different purposes and requirements [1]: The *real-time (RT) service interface* is required for transmitting transducer data such as measurements or set values. The *configuration and planning (CP) interface* provides access to protocol-specific functions like new node identification, obtaining electronic datasheets, and configuration of communication schedules. The CP interface is not time-critical. The *diagnostics and management (DM) interface* is used for accessing sensor and actuator-specific functions like monitoring, calibration, etc. The DM interface is not time-critical, however, some monitoring applications require timestamping.

In the following we discuss real-time requirements, flow control and interaction design patterns which are mostly relevant for the RT service. The DM interface has different requirements and should not interfere with the RT service.

2.1 Real-Time Requirements

There are different kinds of real-time requirements for a distributed system of smart transducers. As a common feature, there is always a *deadline* that specifies a point in time when a specific action has to be completed.

Performing some action locally with respect to real time, like generating a particular Pulse Width Modulation (PWM) signal or making a measurement every 100 ms is relatively easy to achieve if drift of the local clock source is sufficiently low to provide a useful time base.

Timestamping events can be used to temporally relate measurements to each other. In order to create

timestamps with a global validity, a synchronized global time is required among the participating nodes. In most cases, clock synchronization has to be done periodically in order to compensate for the drift of the local clocks. Once a global time is established, timestamping does not pose real-time requirements on the communication system, since timestamped events can be locally stored.

Bounded maximum reaction time requires the communication system to deliver messages within a specified time interval. Standard feedback control algorithms also require low message jitter in order to work correctly.

Globally synchronized actions require the synchronized generation of action triggers in different nodes. This can be achieved by a multi-cast message or assigning actions to an instant on the globally synchronized time scale.

Moreover, a real-time requirement can be *hard*, i. e., deadlines must be held under all circumstances or *soft*, the system is still of use if deadlines are violated infrequently.

Many architectures implement a subset of the described features or provide different features with hard or soft real-time behavior. For example the LAAS architecture [2] for component-based mobile robots specifies local hard-real-time such as a locally closed control loop or the instrumentation of an ultrasonic sensor, while at higher levels, e. g., for globally synchronized actions it provides only soft real-time behavior.

2.2 Models of Flow Control

Communication between subsystems takes place in the *time domain* and the *value domain*. In the value domain, the message data is exchanged, while in the time domain *control information* is transmitted [3].

The communication partner that generates the control information influences the *temporal control flow* of the other communication partner(s). If a communication is controlled by the sender's request we speak of a *push* model, if communication is requested by the receiver, we speak of a *pull* model.

For explanation, let us assume that two or more subsystems need to exchange data over a network. Further, without restrictions to generality, we assume message data to be transmitted from a *producer* to one or more *consumers*. Different from the very popular client-server communication pattern it is necessary to support a one-to-many or many-to-many communication pattern in smart transducer networks because in the common case, the sensor readings of a transducer is needed in more than one place in control applications. Therefore, all of considered networks support this property, however, in different ways. In order to transfer data between the subsystems, they must agree on the mechanism to use and the direction of the transfer.

Figure 1 a) shows the *push* method. The producer is empowered to generate and send its message spontaneously at any time and is therefore independent of the consumer. This loose coupling enables independence between the supplier and consumers of information [4], [5],

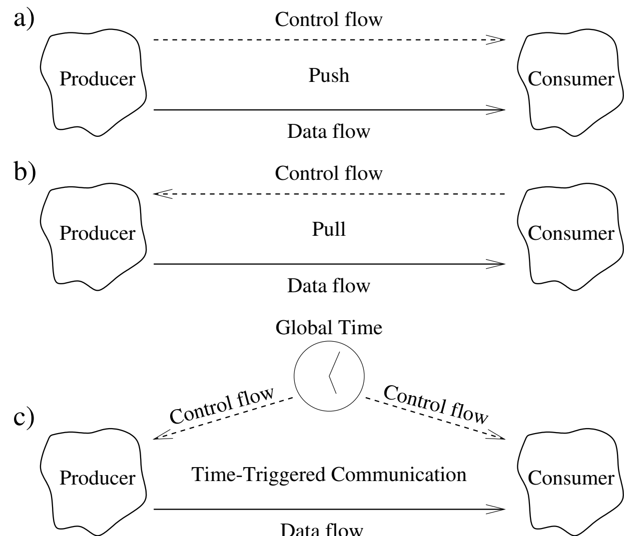


Figure 1. Push-, Pull-, and Time-Triggered Communication

but makes it difficult to enforce temporal predictability in a purely event driven model. Without the option to enforce further temporal constraints, the communication system and the receiving push consumer have to be prepared for data messages at any time, which may result in high resource costs and difficult scheduling. Popular "push" mechanisms are messages, interrupts, or writing to a memory element [6]. The push-style communication is the basic mechanism of event-triggered systems.

In the *pull* model depicted in Figure 1 b) the consumer governs the flow control. Whenever the consumer wants to access the message information, the producer has to respond to the consumer's request.. This facilitates the task for the pull consumer, but the pull supplier (producer) must be watchful for incoming data requests. Popular "pull" mechanisms are polling or reading from a memory element [6]. Pull-style communication is the basic mechanism of client-server systems.

Figure 1 c) depicts a communication model where the flow control is derived from an external trigger. This can be another physical system or the derivation of the triggers from the progress of physical time. In the latter case, the control signals are known *a priori*, which requires predefined scheduling and error detection in the control domain.

2.3 Interaction Design Patterns

We distinguish three basic interaction design patterns for network communication.

In a *master/slave relationship*, at any time one node is considered the *master* while the other nodes are considered to be *slaves*. The master is able to issue synchronization events or to start communications. All slave nodes depend on one master, while the master is independent of a particular slave.

In a *client/server relationship*, a *client* issues a request

to a *server*, which has to answer the request. The client and the server are thus tightly coupled via a pull model.

In a *publish/subscribe relationship*, a *publisher* generates data using the push model. A number of nodes may subscribe to a particular publisher but there is no control flow from subscriber to publisher. Depending on the implementation, a publisher may broadcast its data immediately, transmit its data to an intermediary broker, or transmit its data via point-to-point connections to a list of subscribers. Thus, the number of subscribers may influence the time it takes for a publisher to publish its data.

An architecture may hide the communication model by implementing a distributed shared memory on top of the communication. This way, an application uses the same interface to access data locally or remote. However, a memory interface does not transport control data, e. g., in order to launch a particular task upon reception of an event. Such functionality can either be achieved via polling, but that requires an adequate poll frequency and comes with a noticeable overhead [7] or solved via additional features like interrupt generation after update of a specific data field.

2.4 Diagnostics and Management

While the RT interface provides only access to a limited data set consisting of the actual needed transducer data, for debugging or monitoring purposes, additional data about the operation of the transducer is of interest.

Transmission of these data typically is not time-critical, but must not interfere with the RT service leading to an unwanted probe-effect [8]. Furthermore, monitored RT data should either have time stamps or it must be transmitted before a (typically soft) deadline.

2.5 Configuration and Planning

Large smart transducer systems require support by automatic setup facilities in order to keep up with the complexity of setting parameters correctly. Therefore, smart transducer system should be supported by a tool architecture enabling a plug-and-play-like integration of new nodes.

We consider different configuration and planning scenarios:

In the *replacement scenario*, a broken node is to be exchanged by a new one of the same type. Therefore, the new node has to be detected and a backup of the configuration of the broken node has to be uploaded. However, specific parameters like calibration data will have to be created anew.

In the *initial set-up scenario*, a set of nodes is configured in order to execute a particular communication. This action requires a system specification, and, in most cases, a human operator to perform tasks that cannot be solved automatically.

In the *extension scenario*, a distributed application is extended by extra nodes in order to improve the performance and possibilities of the system. In this case the system managing the configuration must have knowledge how to integrate new transducers in order to upgrade the system. An example of such an approach is outlined in [9].

For the purpose of node identification and documentation, a node is assigned a machine-readable description describing the node's features. Example for such descriptions are the Transducer Electronic Datasheets of IEEE 1451.2 [10] or the Device Profiles in CANopen [11].

3 COSMIC

3.1 COSMIC communication abstractions

COSMIC is middleware which is designed for small embedded systems, supporting heterogeneous networks and cross network communication. It provides a publish/-subscribe abstraction over different addressing and routing mechanisms as well as it considers different latency properties of the underlying networks. COSMIC provides typed event messages (EM) identified by an event UID which identifies the content of a message rather than a source or destination address. Further, EMs have attributes which define a temporal validity of the EM. It should be noted that the term "event" does not refer to a specific synchrony class but just denotes a typed message. As indicated previously, in a real–time embedded environment the pure push model creates problems because the consumers of the information must be ready to receive and process this information at any time. This may lead to situations where some of the messages are lost. Therefore, COSMIC introduces the notion of event channels (EC) which allow specifying temporal constraints and delivery guarantees of individual communication channels explicitly. COSMIC supports three event channel classes: A hard real-time event channel (HRTEC) offers delivery guarantees based on a time-triggered scheme. EMs pushed to a soft real-time event channel (SRTEC) are scheduled according to the earliest deadline first (EDF) algorithm. The respective deadline is determined by the temporal validity information in the attribute field of the EM. Because soft real-time EMs which have already missed their transmission deadline may cause further deadline misses of other soft real-time EMs, they are discarded and the respective local application is notified. The application then can decide about re-sending in a lower real-time class or just skip it. Finally, a non real-time event channel (NRTEC) disseminates events that have no timeliness requirements.

ECs are established prior to communication allowing the middleware to reserve the necessary resources and perform the binding to the underlying mechanisms of the communication network.

The COSMIC architecture is not bound to a particular network. An implementation based on Controller Area Network (CAN) [12] is described in the next section.

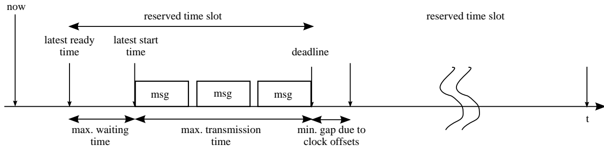


Figure 2. Structure of a time slot

3.2 COSMIC–on–CAN architecture

Implementing the event model requires to map the abstractions of that model (publisher, subscriber, event channel, event instance) to the elements provided by the infrastructure of the communication system. The respective functionality to perform these mapping in COSMIC is encapsulated in the Event Channel Handler which resides in every node. Given the constraints in bandwidth and in message length on CAN, the implementation of events and event channels has to exploit the underlying CAN mechanisms. To save the rare space in the message body [13] and to reduce the inherent CAN message overhead, significant information is also encoded via the CAN ID.

The implementation is based on the extended 29 bit CAN ID of the CAN 2.0 B specification. The 29-Bit CAN identifier (CAN-ID) is structured into three fields, i. e., an 8 bit priority field used to prioritize messages according to HRTEC, SRTEC, and NRTEC, a 7 bit node- ID ensuring unique identifiers and a 14-Bit event tag. The assignment of an event UID to an event tag is performed dynamically by the COSMIC middleware infrastructure. A description of this infrastructure and the respective binding protocol is described in [14].

3.3 Enforcing temporal constraints in COSMIC

HRTECs provide delivery guarantees and use reserved time slots in a Time Division Multiple Access (TDMA) scheme organized in periodic rounds. The intention of the reservation-based scheme is to avoid collisions by statically planning the transmission schedule. Hence, any conflict between HRTECs is avoided. COSMIC is implements clock synchronization based on the algorithm proposed in [15].

Because a CAN message cannot be preempted a non hard real-time message transmission may delay a hard real-time message by the maximum length of one CAN message in the worst case. Furthermore, transient transmission faults may increase the time needed to transmit a hard real-time message. Therefore, a hard real-time slot is extended according to Fig. 2. The protocol relies on the fault–handling mechanisms of the standard CAN which has an impact on the fault classes which we can handle. For a message with b bytes of data, the maximum length of the message including header and bit–stuffing is: $Length_{message} = 75 + \lfloor b \cdot 9.6 \rfloor$[1]. Under the assumption of f single transmission failures, the required minimum time–slot length is: $slot\ length = 2 \cdot t_{message} + (t_{message} + 18) \cdot f + 3 bittimes$. Assuming a single message failure of an 8 Byte message at 1 Mbit/sec (msg transmission time: 151μsec, fault detection and retransmission 18μsec) and a gap between the slots of 50μsec, approx. 1900 slots/sec can be allocated. If it is necessary to tolerate a permanent controller failure, this number drops down to an approximate number of 350 slots/sec. Compared to a maximum throughput of about 6500 maximum length messages per second, the number of possible HRT slots is low. However, these numbers refer to the number of *guaranteed* HRTECs not to the number of messages which actually can be sent. Unlike in pure time-triggered systems, the CAN priority mechanism can be used to transmit SRT or NRT messages in cases where a HRT message has been received a message successfully by all operational nodes[2]. Thus, time redundancy only costs bandwidth if faults really occur, which may be relatively rare compared to the overall traffic. The priority-based arbitration mechanism is also exploited to schedule SRTECs and NRTECs. HRT messages always reserve the highest priority. The relation between the priorities of HRT, SRT and NRT messages can be expressed by the relation: $P_{HRT} < P_{SRT} < P_{NRT}$ (a lower numerical value represents a higher priority). The assignment enforces that a message of a lower real–time class never will interfere with one of a higher class during bus arbitration. We assume the highest priority (0) for HRT messages and a small number of fixed low priorities for NRT messages. The remaining priority levels are available for scheduling SRT messages. They have to be mapped on a time scale to express the temporal distance of a deadline. The closer the deadline, the higher the priority. Mapping deadlines to priorities will cause the problem that static priorities cannot express the properties of a deadline, i.e. a point in time. A priority corresponding to a deadline can only reflect this deadline in a static set of messages. When time proceeds and new messages become ready, a fixed priority mechanism cannot implement the deadline order any more. It is necessary to increase the priorities of a message when time approaches the deadline, i.e. with decreasing laxity.

3.4 Device Descriptions

COSMIC devices are described in an XML-based language called CODES (COSMIC embedded DEvice Specification) [18]. The descriptions are structured into three parts. Part one, *General Information* contains the node's name, its type, its manufacturer, its unique identifier, its networking facilities, its supported event channel types, recycling information, and a clear text description of the device. This part also contains version information about the component. The second part contains all *event definitions*, i. e., the description of all events produced or consumed by the device.

[1]The factor 9.6 is because of the bit stuffing mechanism

[2]There are situations of inconsistent replicas and even inconsistent omissions (according to [16], inconsistent omissions occur with a probability in the order of 10^{-9}). Kaiser and Livani [17] describe a transparent mechanism to handle these situations.

For each event is described by a plain text tag and a unique identifier. The event's definition includes a list of attributes giving non-functional details about the event, e. g., the event's expiration time. Whenever an event is disseminated, it is sent as a compact message. This message's data structure is specified in the event definition. For each field in the data structure, its name, data type and byte order are included in the description. This information can be used by tools to automatically create decoder for a compact message. Fields representing a measurement are annotated by the corresponding physical dimension in a machine-readable format. Non-measurement fields are described by lists or state machines. Each field may contain also attributes, e. g., the valid data range. The last part contains the declaration of all *event channels* and their properties. Each event channel definition contains the subject UID linking it to the respective event definition, the class of event channel, the direction of the event channel as seen locally, and again a list of attributes, e. g., the channel's period.

While the greater part of the description contains static information, some elements are not suitable for integration into a static description document, e.g. the period of an event channel, which certainly will vary depending on the application. To overcome this problem, parameterization was introduced. Any non–static element can be marked as a parameter in the static description. The element's actual value is then defined and stored external to the static description. Parameters are stored in path–value–pairs, similar to well–known name–value–pairs. Instead of naming the parameter, it's XPath expression within the static description acts as the identifier. A scheme for mapping this structure down to a binary parameter storage scheme suitable for small devices exists. Whenever the description is used, the parameters are included beforehand. The query service (see below) is a suitable place that will handle this inclusion in running systems. Having each parameter's path expression eases the integration into the description document.

CODES descriptions play a central role for COSMIC components. The life of a COSMIC component starts with the description document created during the component's design phase. It is used during the following implementation phase to generate parts of the component's code [18]. Black–box tests of the component can be assisted, e.g. tests for timing behavior or testing the compliance of disseminated events with their description in terms of the data structure, value ranges, or precision. The descriptions are further useful throughout the component's life–cycle: During the integration phase into a larger system, a number of compatibility checks can be performed automatically. Schedules for the HRT communication can be derived from the respective set of descriptions. While the component is in use, the ready availability of its description forms the basis for dynamic use of formerly unknown components. Currently, this requires a priori knowledge or the interaction with a user. In the long run, the integration of semantic web technology is planned to enable true autonomous dynamic cooperation of components. Whenever a system is in need of maintenance, the availability of the descriptions is beneficial, too. They provide a quick overview of the system, i.e. what components are available, and how they are configured.

The descriptions are stored within the devices themselves. They can be retrieved and queried at run–time: On system start–up, and whenever a new component is added to a system, an automatic configuration is necessary for the component to be able to participate in communication. During this configuration, the components' descriptions and parameters are uploaded to the node running the event channel broker. This node also runs a query service which makes the descriptions accessible from outside the system. The parameters are included in the static part of the description, yielding a single document describing the current configuration of the components. Requests to the query service are given as XSLT transformations [19]. The transformations are applied to the CODES descriptions on the node running the query service, thus enabling even rather low–power nodes to make use of the query service. XSLT transformations represent a suitable technology not only for the query service, but throughout the different application areas of the CODES descriptions. They are e.g. also used for code generation.

4 LIN

4.1 System Architecture

Each message in LIN is encapsulated in a single message cycle. The message cycle is initiated by the master and contains two parts, the frame header sent by the master and the frame response, which encompasses the actual message and a checksum field. The frame header contains a sync brake (allowing the slave to recognize the beginning of a new message), a sync field with a regular bit pattern for clock synchronization and an identifier field defining the content type and length of the frame response message. The identifier is encoded by 6 bit and 2 bits for protection. Figure 3 depicts the frame layout of a LIN message cycle.

The frame response contains up to 8 data bytes and a checksum byte. Since an addressed slave does not know *a priori* when it has to send a message, the response time of a slave is specified within a time window of 140% of the nominal length of the response frame. This gives the node some time to react on the master's message request, for example to perform a measurement on demand, but introduces a noticeable message jitter for the frame response.

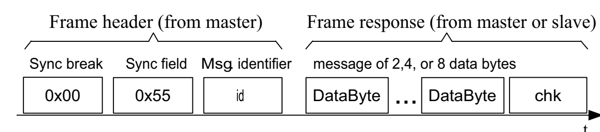


Figure 3. LIN frame format

The interaction between master and slave is a plain pull mechanism, since the slaves only react on the frame header from the master. It is the master's task to issue the respective frame headers for each message according to a scheduling table. From a data-centric perspective, the communication is defined by messages that are subscribed by particular slaves for reception or transmission. The configuration of the network must ensure that each message has exactly one producer.

In 2003, LIN was enhanced by extra features leading to the LIN 2.0 specification. New features introduced in LIN 2.0 are an enhanced checksum, sporadic and event-triggered communication frames, improved network management (status, diagnostics) according to ISO 14230-3 / ISO 14229-1 standards, automatic baud rate detection, standardized LIN product ID for each node, and an updated configuration language to reflect the changes.

In addition to the unconditional frames (frames sent whenever scheduled according to the schedule table) provided by LIN 1.3, LIN 2.0 introduces *event-triggered frames* and *sporadic frames*.

Similar to unconditional frames, event-triggered frames begin with the master task transmitting a frame header. However, corresponding slave tasks only transmit their frame response if the corresponding signal has changed since the last transmission. Unlike unconditional frames, multiple slave tasks can provide the frame response to a single event-triggered frame, assuming that not all signals have actually changed. In the case of two or more slave tasks writing the same frame response, the master node has to detect the collision and resolve it by sequentially polling (i.e., sending unconditional frames) the involved slave nodes. Event-triggered frames were introduced to improve the handling of rare-event data changes by reducing the bus traffic overhead involved with sequential polling.

Sporadic frames follow a similar approach. They use a reserved slot in the scheduling table, however, the master task only generates a frame header when necessary, i. e., when involved signals have changed their values. As this single slot is usually shared by multiple sporadic frames (assuming that not all of them are sent simultaneously), conflicts can occur. These conflicts are resolved using a priority-based approach: frames with higher priority overrule those with lower priority.

In addition to signal-bearing messages, LIN 2.0 provides *diagnostic messages*. These messages use 2 reserved identifiers (0x3c, 0x3d). Diagnostic messages use a new format in their frame response called *PDU* (Packet Data Unit). There are two different PDU types: *requests* (issued by the client node) and *responses* (issued by the server node).

The LIN 2.0 configuration mode is used to set up LIN 2.0 slave nodes in a cluster. Configuration requests use SID values between 0xb0 and 0xb4. There is a set of mandatory requests that all LIN 2.0 nodes have to implement as well as a set of optional requests. Mandatory re-

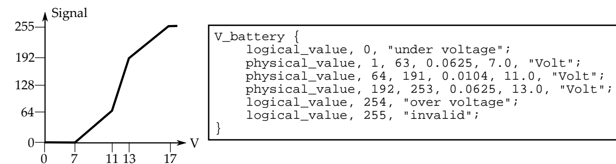


Figure 4. Example for a LIN signal definition

quests are:

- Assign Frame Identifier: This request can be used to set a valid (protected) identifier for the specified frame.
- Read By Identifier: This request can be used to obtain supplier identity and other properties from the addressed slave node.

Optional requests are:

- Assign NAD: Assigns a new address to the specified node. Can be used to resolve address conflicts.
- Conditional Change NAD: Allows master node to detect unknown slave nodes.
- Data Dump: Supplier specific (should be used with care).

4.2 Device Descriptions

Each LIN 2.0 [20] node is accompanied by a *node capability file* (NCF). The NCF contains:

- The node's name.
- General compatibility properties, e.g. the supported protocol version, bit rates, and the LIN product identification. This unique number is also stored in the microcontroller's ROM and links the actual device with its NCF. It consists of three parts: supplier ID (assigned to each supplier by the LIN Consortium), function ID (assigned to each node by supplier), and variant field (modified whenever the product is changed but its function is unaltered)
- Diagnostic properties, e.g. the minimum time between a master request frame and the following slave response frame.
- Frame definitions. All frames that are published or subscribed by the node are declared. The declaration includes the name of the frame, its direction, the message ID to be used, and the length of the frame in bytes. Optionally, the minimum period and the maximum period can be specified. Each frame may carry a number of signals. Therefore, the frame's declaration also includes the associated signals' definitions. Each signal has a name, and the following properties associated with it: **Init value** specifies the value used from power on until the first message from the publisher arrives. **Size** specifies the signal's size in bits. **Offset** specifies the position within the frame. **Encoding** specifies the signal's rep-

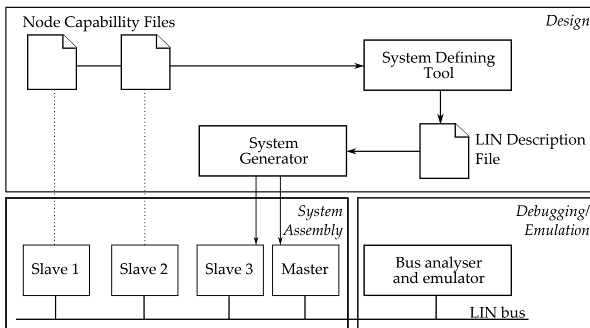


Figure 5. Development phases in LIN

resentation. The presentation may be given as a combination of the four choices *logical value*, *physical value*, *BCD value*, or *ASCII value*. Declarations of physical values include a valid value range (minimum and maximum), a scaling factor, and an offset. Optionally, this can be accompanied by a textual description, mostly to document the value's physical unit. An example is given in figure 4.

- Status management: This section specifies which published signals are to be monitored by the master in order to check if the slave is operating as expected.
- The free text section allows the inclusion of any help text, or more detailed, user–readable description.

The node capability file is a text file. the syntax is simple and similar to C. Properties are assigned using `name = value;` pairs. Subelements are grouped together using curly braces, equivalent to blocks in C.

LIN clusters are configured during the design stage using the *LIN Configuration Language*. This language is used to create a *LIN description file* (LDF). The LDF describes the complete LIN network. The development of a LIN cluster is partitioned into three phases (see figure 5). During the *design phase*, individual NCFs are combined to create the LDF. This process is called *System Definition*. For nodes to be newly created, NCFs can be created either manually or via the help of a development tool. From the LDF, communication schedules, and low–level drivers for all nodes in the cluster can be generated (*System Generation*). Based on the LDF, the LIN cluster can be emulated and debugged during the *Debugging and Node Emulation* phase. In the *System Assembly* phase, the final system is assembled physically, and put to service.

In addition to the LIN configuration language and LDF, which are the most important tools to design a LIN cluster, the LIN specification defines a (mandatory) interface to software device drivers written in C. Also, many tools exist that can parse a LDF and generate driver modules by themselves. The LIN C API provides a signal based interaction between the application and the LIN core (core API).

5 TTP/A

5.1 Communication System Architecture

The information transfer between a smart transducer and its communication partners is achieved by sharing information that is contained in an internal interface file system (IFS), which is situated in each smart transducer. The IFS provides a unique address scheme for transducer data, configuration data, self-describing information, and internal state reports of a smart transducer [1]. It also serves as decoupling element, providing a push interface for producers writing to the IFS and a pull interface for consumers reading from the IFS. Each transducer can contain up to 64 files in its IFS. An IFS file is an indexed array of up to 256 records. A record has a fixed length of four bytes. Every record of an IFS file has a unique hierarchical address (which also serves as the global name of the record) consisting of the concatenation of the cluster name, the logical node name, the file name, and the record name.

A time-triggered sensor bus will perform a periodical time-triggered communication by sending data from IFS addresses to the fieldbus and writing received data to IFS addresses at predefined points in time. Thus, the IFS is the source and sink for all communication activities. Furthermore, the IFS acts as a temporal firewall that decouples the local transducer application from the communication activities.

Communication is organized into rounds consisting of several TDMA slots. A slot is the unit for transmission of one byte of data. Data bytes are transmitted in a standard UART format. Each communication round is started by the master with a so-called fireworks byte. The fireworks byte defines the type of the round and is a reference signal for clock synchronization. The protocol supports eight different firework bytes encoded in a message of one byte using a redundant bit code supporting error detection.

Generally, there are two types of rounds:

Multipartner round: This round consists of a configuration-dependent number of slots and an assigned sender node for each slot. The configuration of a round is defined in a data structure called "RODL" (ROund Descriptor List). The RODL defines which node transmits in a certain slot, the operation in each individual slot, and the receiving nodes of a slot. RODLs must be configured in the slave nodes prior to the execution of the corresponding multipartner round. An example for a multipartner round is depicted in Figure 6.

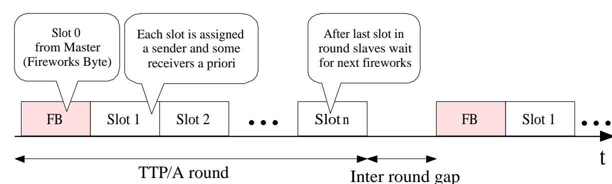


Figure 6. TTP/A Multipartner Round

Master/slave round: A master/slave round is a special round with a fixed layout that establishes a connection between the master and a particular slave for accessing data of the node's IFS, e. g., the RODL information. In a master/slave round the master addresses a data record using a hierarchical IFS address and specifies an action like reading of, writing on, or executing that record.

The multipartner (MP) round establishes a real-time communication service with predefined access patterns. Master/slave (MS) rounds are scheduled periodically between multipartner rounds, whereas the most commonly used scheduling scheme consists of MP rounds alternating with MS rounds. The MS rounds allow maintenance and monitoring activities during system operation without a probe effect. The MS rounds enable random access to the IFS of all nodes, which is required for establishing two conceptual interfaces to each node, a *configuration and planning* (CP) interface and a *diagnosis and management* DM interface. These interfaces are used by remote tools to configure node and cluster properties and to obtain internal information from nodes for diagnosis.

5.2 Smart Transducer Descriptions

For a uniform representation of all system aspects, an XML-based format is used [21]. A *smart transducer descriptions* (STD) describe the node properties.

There are two types of STDs: *Static STDs* describe the node properties of a particular field device family. Static STDs contain node properties that are fixed at node creation time and act as a documentation of the nodes' features. In contrast, *Dynamic STDs* describe properties of individual nodes, as they are used in a particular application.

Instead of storing the STDs directly on a smart transducer, the node contains only a unique identifier consisting of a series and a serial number, whereas the serial number identifies the node type and the serial number differentiates instances of the same node type. This unique identifier is used to access the node's datasheet on an external server. Thus, node implementations keep a small footprint, while the size of the descriptions is not significantly limited.

5.3 Cluster Configuration Description

The cluster configuration description (CCD) contains descriptions of all relevant properties of a fieldbus cluster. It acts as the central structure for holding the meta-information of a cluster. With help of a software tool capable of accessing the devices in a smart transducer network it is possible to configure a cluster with the information stored in the CCD. A CCD consists of the following parts:

- *Cluster description meta information:* This block holds information on the cluster description itself, such as the maintainer, name of the description file, or the version of the CCD format itself.
- *Communication configuration information:* This information includes round sequence lists as well as round descriptor lists, which represent the detailed specification of the communication behavior of the cluster. Other properties important for communication include the UART specification and minimum/maximum signal run times.
- *Cluster node information:* This block contains information on the nodes in a cluster. These nodes are represented either by a list of dynamic STDs or by references to static STDs.

6 Discussion

Table 1 lists the main features of the three transducer networks with respect to the concepts describe in Section 2. All three approaches provide a real-time service with hard real-time message guarantees, but use different interaction design patterns. COSMIC comes with a publish-subscribe approach where nodes publish their data using the push principle. LIN is a master-slave network where each message is activated by the master. TTP/A uses a master-slave configuration in order to establish a common time base and then follows a predefined communication schedule based on the physical progression of time.

The basic scheduling mechanisms for hard real-time messages by using a static TDMA scheme is the same in all three approaches. The mechanisms for other data is different – TTP/A provides a polling mechanism via master-slave rounds, LIN 2.0 introduced event messages. COSMIC is the most flexible by providing EDF-scheduled soft real-time messages as well as non real-time messages. However, a full implementation of the SRTC requires substantial software because of the dynamic priorities and the more complex handling of discarded messages. Therefore, it has so far only be implemented on more powerful hardware under RT-Linux. Additionally, COSMIC relies on synchronized clocks while LIN and TTP/A require less effort for the proper protocol operation.

The advantages of COSMIC's publish-subscribe are a loose coupling between producer and consumer which facilitates the configuration of a network. The type of the channel to which EM of a certain type is pushed is defined by the publisher. The subscription and the respective guarantees for delivery at the subscriber side, however, may be of the same or a lower real-time class. This enables reception of a critical hard real-time message also for applications which do not need the respective delivery guarantees, e.g. a navigation task which uses critical messages from an obstacle avoidance system.

The pull principle in LIN makes a node's implementation very simple, but causes an overhead on the network due to the frequent message requests from the master. Moreover, since the LIN slaves do not know the time of a request *a priori*, it becomes difficult to time a measurement adequately or to synchronize measurements.

The time-triggered approach of TTP/A comes with

Table 1. Feature comparison

	LIN	COSMIC	TTP/A
Criticality Levels	HRT, SRT	HRT, SRT	HRT
Flow control model	pull	push	TT, pull
Interaction pattern	master/slave	publish/ subscribe	TT, master/slave
Bounded transmission time	yes	yes	yes
Global Time	no	yes	yes
Synchronized actions	no	no	yes
Middleware abstraction	messages	event messages and channels	IFS
Device Descriptions Language	LIN-specific	XML	XML

high efficiency, predictability, and the possibility to synchronize actions. However, the configuration effort of a TTP/A network is higher than for a LIN device or COSMIC devices not requiring stringent real-time guarantees. For example, a TTP/A node has to be configured with the correct schedule before it can participate in the RT communication. In contrast, a LIN node or a COSMIC node might be reused in another application without reconfiguration of the node. Anyway, all three approaches depend on an adequate tool support.

The IFS concept of TTP/A is an abstraction mechanism that hides the time-triggered messages from the application. The IFS implements a distributed shared memory that provides a simple interface for applications. Therefore, TTP/A applications are not triggered by the reception of a message, which allows for a separation of communication and computation.

LIN is designed to serve as sub-bus in automobiles and is therefore specified in a very rigid way towards use in a specific end product. This makes the LIN architecture, though the approach is resource efficient and interesting, less suitable for applications which require a higher degree of cooperation between the nodes and also the rather constraint LIN message format restricts larger sensor-actuator systems. Also the LIN device description is rather focussed on the specific LIN application area.

In contrast, COSMIC and TTP/A specify several high-level features, while leaving details of physical and data link layer up to the implementer. The XML-based datasheets of COSMIC and TTP/A are easily extendable in order to support future extensions.

The mechanisms of the three approaches are different, which makes them incompatible in the first place. In order to achieve interoperability between heterogeneous networks, an adequate interface system, whereas the mechanisms of COSMIC and TTP/A are candidates rather than LIN. COSMIC provides a versatile message interface that abstracts over the underlying communication protocol. On the other hand, the IFS approach of TTP/A allows to abstract over the communication by establishing a distributed shared memory. The IFS comes with the main advantage of being easily adapted to a different protocol, however for convenient application development, tools supporting the set up of the distributed communication schedules are required. Thus, it is up to the application developer if a message-based interface (COSMIC) or a memory-based interface (TTP/A / IFS) is preferred.

7 Conclusion

The contributions of this paper are threefold: Firstly we have elaborated a set of requirements for different kinds of real-time constraints for a distributed system of smart transducers.

Secondly, we have presented and analyzed the concepts of three different smart transducer interface implementation approaches. Each approach has its specific focus concerning an application area. LIN is the protocol with the lowest hardware and cost requirements, however several design decisions restrict its use to an isolated sub-bus for automotive body electronics or simple control systems in industrial automation. LIN is supported by mature tools from automotive suppliers. TTP/A has a similar resource footprint as LIN but firstly substantially benefits from the strict time-triggered communication scheme and secondly provides a convenient distributed shared memory programming model where consistency problems are solved by the synchrony of the communication system. Parameters such as communication speed can be adapted in a rather flexible way depending on the physical network. This makes TTP/A an interesting choice for all kind of low-cost embedded time-triggered applications with real-time requirements. Additionally, the IFS is standardized by OMG in the Smart Transducer Interface Standard [22]. COSMIC provides flexible real-time support and will integrate well into distributed applications with a publish-subscribe communication scheme. The main objective of COSMIC was interoperability between networks with different real-time properties. Thus, a higher overhead in the nodes may be needed. COSMIC and TTP/A come with different configuration support providing similar features

A third contribution of the paper is the discussion of device description. We think that this is an important issue because it firstly underlines the hardware/software (and probably mechanical) nature of a smart transducer and the intrinsically component-based system structure and secondly is indispensable in a complex reliable control system. Presently, device descriptions are mainly used during system configuration to avoid faults from manual set-up. The LIN NFC and also LDF exactly meet these requirements. Device description of TTP/A and COSMIC go beyond the needs of configuration and also are intended for

dynamic use. This can range from diagnostic purposes to dynamic device discovery and use during operation.

Although being quite different, we think that it will be possible to establish methods and tools that operate on a meta-level and can be used to configure an application using different underlying fieldbus systems. In order to achieve this, a generic interface model for transducer data has to be found. The Interface File System (IFS) presented with TTP/A seems to be a promising approach for forming a generalized interface, since it is relatively easy to convert transducer data onto an IFS. We will further investigate ways to provide coexistence and cooperation between the different network and programming models.

Acknowledgments

This work was supported in part by the ARTIST2 Network of Excellence on Embedded Systems Design under contract No. IST-004527 and by the Austrian FWF project TTCAR under contract No. P18060-N04. We would like to thank Christian Paukovits, Stefan Pitzek, Gernot Klingler, Christian El-Salloum and Andreas Pfandler for constructive comments on an earlier version of this paper.

References

[1] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, March 2001.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337, April 1998.

[3] A. Krüger. *Interface Design for Time-Triggered Real-Time System Architectures*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, April 1997.

[4] K. Mori. Autonomous decentralized systems: Concepts, data field architectures, and future trends. In *International Conference on Autonomous Decentralized Systems (ISADS93)*, 1993.

[5] J. Kaiser and M. Mock. Implementing the real–time publisher/subscriber model on the controller area network (CAN). In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 172–181, Saint Malo, France, May 1999.

[6] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1999.

[7] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, April-June 1997.

[8] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[9] S. Pitzek and W. Elmenreich. Plug-and-play: Bridging the semantic gap between application and transducers. In *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA05)*, volume 1, pages 799–806, Catania, Italy, September 2005.

[10] Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1451.2-1997, Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Micro-processor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*, 1997.

[11] CAN in Automation e.V. CANopen - Communication profile for industrial systems. available at http://www.can-cia.de/downloads/.

[12] Robert Bosch GmbH. CAN specification version 2.0, September 1991.

[13] L.-B. Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro*, 22(4):28–36, 2002.

[14] J. Kaiser and C. Brudna. A publisher/subscriber architecture supporting interoperability of the CAN–bus and the internet. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS 2002), Västerås, Sweden*, 2002.

[15] M. Gergeleit and H. Streich. Implementing a distributed high–resolution real–time clock using the CAN–bus. In *1st International CAN Conference*, 1994.

[16] J. Ruffino, P. Verissimo, C. Almeida, and L. Rodrigues. Fault–tolerant broadcasts in CAN. In *Proceedings FTCS–28, Munich, Germany*, 1998.

[17] J. Kaiser and M. A. Livani. Achieving fault–tolerant ordered broadcasts in CAN. In *Proceedings of the Third European Dependable Computing Conference (EDCC–3), Prague*, September 1999.

[18] J. Kaiser and H. Piontek. CODES: Supporting the development process in a publish/subscribe system. In *Proceedings of the fourth Workshop on Intelligent Solutions in Embedded Systems WISES 06*, 2006.

[19] M. Kay, Ed. W3C XSL transformations (XSLT) version 2.0. http://www.w3.org/TR/xslt20, June 2006.

[20] Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc. Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification v2.0, 2003.

[21] S. Pitzek and W. Elmenreich. Configuration and management of a real-time smart transducer network. In *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, pages 407–414, Lisbon, Portugal, September 2003.

[22] Object Management Group (OMG). *Smart Transducers Interface V1.0*, January 2003. Specification available at http://doc.omg.org/formal/2003-01-01 as document ptc/2002-10-02.

Delay-Bounded Medium Access for Unidirectional Wireless Links[1]

Björn Andersson, Nuno Pereira, Eduardo Tovar
IPP Hurray Research Group
Polytechnic Institute of Porto, Portugal
{bandersson, npereira, emt}@dei.isep.ipp.pt

Abstract

Consider a wireless network where links may be unidirectional, that is, a computer node A can broadcast a message and computer node B will receive this message but if B broadcasts then A will not receive it. Assume that messages have deadlines. We propose a medium access control (MAC) protocol which replicates a message in time with carefully selected pauses between replicas, and in this way it guarantees that for every message at least one replica of that message is transmitted without collision. The protocol ensures this with no knowledge of the network topology and it requires neither synchronized clocks nor carrier sensing capabilities. We believe this result is significant because it is the only MAC protocol that offers an upper bound on the message queuing delay for unidirectional links without relying on synchronized clocks.

1 Introduction

Consider a computer node A that can broadcast a message and computer node B that can receive this message but if B broadcasts then A cannot receive it. We say that the network topology has a *unidirectional link* from A to B. Empirical data show that unidirectional links exist and they are not uncommon; typically, in networks with low-power radios, 5-15% of all links are unidirectional [1-7]. This has been recognized at the routing layer but the MAC layer is still poorly developed for unidirectional links. Traditional MAC protocols fail on unidirectional links. For example, consider a node A that performs carrier-sensing before it sends a message to node B. Node B transmits as well but due to the fact that the link A→B is unidirectional, node A perceives that there is no carrier. Consequently, node A transmits and it collides with the transmission from node B. Also, RTS/CTS (Request-to-Send/Clear-to-Send) exchanges fail as well because node A sending a RTS-packet does not receive a CTS packet from node B. In addition, protocols that allow collisions but let a sender A wait for an acknowledgement from node B can fail too. Node B received the message but since the link A→B was unidirectional, node B cannot send an acknowledgement back to the sender A: the sender A has to wait forever. The only existing solutions today for medium access in the presence of unidirectional links require synchronized clocks [8] or cause unbounded number of collisions.

In this paper we study medium access of wireless links which may be unidirectional. We show, informally, that designing a collision-free MAC protocol is impossible. For this reason, we design a replication scheme; every message that an application requests to transmit is replicated in time by the MAC protocol with carefully selected pauses between the transmissions of replicas. This guarantees that for every message, *at least one of its replicas is transmitted without collision*.

The protocol proposed in this paper is quite heavy-weight. We will see that such a high overhead is necessary in order to bound the number of collisions and hence to achieve real-time guarantees in the presence of unidirectional links; this is our main focus. But less time-critical applications (such as file transfer) demand high throughput and networks often have links that are mostly bidirectional, and unidirectional only occasionally. In such networks, the robustness and delay guarantees offered by the bounded number of collisions of our scheme is not worth the high overhead. For this reason, we will discuss how the protocol can be adapted to obtain an *average-case* overhead similar to "normal" protocols designed for bidirectional links, while retaining the upper bound on the number of collisions in the presence of unidirectional links.

The remainder of this paper is organized as follows. Section 2 presents the system model, the impossibility of designing a collision-free MAC protocol and the main idea of the protocol. Section 3 presents schedulability analysis of sporadic message streams. Section 4 presents implementation and experimental validation of the protocol. Section 5 reviews previous work and discusses unidirectional links in its larger context. Finally, Section 6 offers conclusions.

[1] Due to space limitations, some results in this conference version are omitted. See the extended version for details [11].

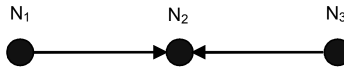


Fig. 1. A network topology which illustrates the impossibility of collision-free medium access in the presence of unidirectional links. N_1 can transmit to N_2 but N_2 cannot transmit to N_1. Analogously for N_2 and N_3. When N_1 and N_3 transmit there will be a collision on node N_2.

2 Preliminaries and the Main idea

2.1 Network and Message Model

The network topology is described using a graph with nodes and links. A node represents a computer node. A link is directed. Consider a node N_i that broadcasts a message or any signal (for example an unmodulated carrier wave). Then node N_k will receive it if and only if there is a link in the topology graph from node N_i to node N_k. A node can only transmit by performing a broadcast and it is impossible for a node N_i to broadcast such that only a proper subset of its neighbor nodes receive it. No assumption on the topology of each node is made. It is allowed that a node has only outgoing links or only ingoing links or no links at all. Unless otherwise stated, the topology is assumed to be unknown to the MAC protocol. In Section 5, we will discuss how knowledge of the network topology can be exploited.

Let *mtotal* denote the number of nodes and let *m* denote the number of nodes that can transmit. Nodes are indexed from 1 to *mtotal*, where the *m* nodes that can transmit have the lowest index. As an illustration, consider a network with $mtotal = 5$ nodes but 2 nodes will never transmit; these nodes will have index 4 and 5. The other $m = 3$ nodes are permitted to request to transmit and these nodes have index 1, 2 and 3.

We will initially assume that on each node with index 1..*m*, there is a single application and it makes only a single request to the MAC protocol to transmit a message. The exact time of the request is unknown before run-time and the MAC protocol does not know about the time of the request before it occurs. Let J_i denote this single message on node N_j. (J_i is analogous to a job in processor scheduling.) It is assumed that when the MAC protocol sends a message it takes one time unit. We are interested in finding a value *z* such that it holds for any node that the time from when a message transmission request is made at a node until this message is successfully transmitted without collision is at most *z*.

Let $prop_{i,j}$ denote the propagation delay of the medium between nodes N_i and N_k. We assume that $prop_{i,k}$ is unknown but it is bounded such that $\forall i,k \in \{1..mtotal\}: 0 < prop_{i,k} \leq prop$. Hence, *prop* is an upper bound on the propagation delay of the medium; we expect that a typical value is $prop = 1\mu s$ for distributed real-time systems in a small geographical area, such as a ship, a factory or a virtual caravan of cars. We assume that *prop* is finite but we make no assumptions on its actual value. However, we assume the following: (i) nodes can "boot" at different times and when they boot, they do not have synchronized clocks; (ii) when a node is transmitting it cannot receive anything; and (iii) the MAC protocol can be represented as a set of timed automata, with potentially different automata on different computer nodes.

2.2 Impossibility

Let us now show that, under these assumptions, it is impossible to design a collision-free MAC protocol when there are unidirectional links. Consider Figure 1. It illustrates a simple exemplifying topology. For such topology and links characteristics, it is necessary that N_1 does not transmit simultaneously with N_3, in order to guarantee that collisions will not occur. This requires that N_1 can get some information about the other nodes on whether there is an ongoing transmission on the other link. But N_1 cannot hear anything so the transmission from N_1 may overlap with the transmission from N_3, and then N_2 will not receive any of them. Hence, it is impossible to design a MAC protocol that is guaranteed to be collision-free in the presence of unidirectional links. Even if a node knows the topology but it does not know the time when other nodes will transmit then a collision can occur, and hence the above mentioned impossibility also extends to the case where the topology is known to the MAC protocol.

Given the impossibility of collision-free medium access in the presence of unidirectional links we will now design a solution.

2.3 The Main Idea

For each message J_i the MAC protocol transmits the message several times. Each one of them is called a *replica*. Of those replicas from message J_i let $J_{i,1}$ denote the one that is transmitted first. Analogously, let $J_{i,2}$ denote the one that is transmitted second, and so on. The number of replicas transmitted for each message of J_i is $nreplicas(J_i)$, and the time between the start of transmission of $J_{i,j}$ until the start of transmission of $J_{i,j+1}$ is denoted as $\Delta_{i,j}$. Figure 2 illustrates these concepts for the case when all messages request to transmit

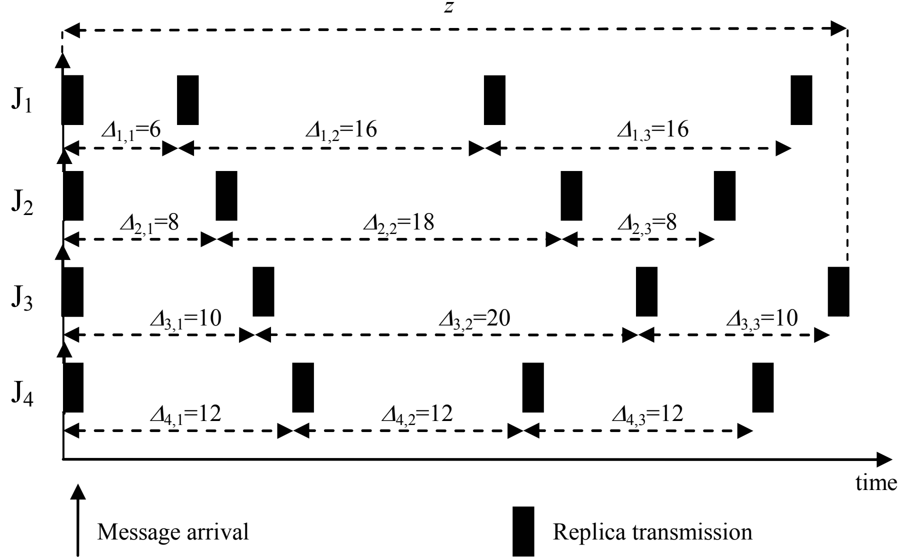


Fig. 2. Transmission of replicas with a possible assignment of Δ:s to messages. J_1, J_2, J_3, J_4 requested to transmit simultaneously at time 0. As it can be seen, at least one replica is collision-free. It turns out that for every possible combination of times of requests of J_1, J_2, J_3, J_4 this is true as well.

simultaneously. We let $J_{i,1}$ be transmitted immediately when J_i is requested to be transmitted. For convenience, we assume in this section (Section 2) that $prop = 0$ and this is known to the MAC protocol. In Section 5, we will discuss a simple technique to extend the results to the case where $prop > 0$.

We will now reason about how to select *nreplicas*(J_i) and then select $\Delta_{i,j}$. It is necessary to select *nreplicas*(J_i) ≥ *m* because otherwise there is a topology for which it is possible that all replicas of J_i collide. To see this, consider *m* nodes where one central node N_k has ingoing links from all other nodes; one of these other nodes is node N_i. There is also a link from N_k to N_i. Let us now consider the case where N_i broadcasts its replicas. Let N_l denote any other node than N_k and N_i. The first message transmission of J_l can happen at any time, so it can collide with one of the replicas from J_i. Analogously, the first replica of another message J_l can collide with another replica of J_i. In addition, the first replica from J_k can occur any time too, so this first replica can be transmitted when J_i sends a replica to N_k. Then N_k will not hear the replica from J_i Hence, if J_i transmits *nreplicas*(J_i) < *m* replicas, it can happen that none of them are received at node N_k. Therefore, *nreplicas*(J_i) must be selected such that:

$$nreplicas\left(J_i\right) \geq m$$

Later in this section, we will select $\Delta_{i,j}$ such that at most one replica from J_i can collide with a replica of J_l. With such an assignment of $\Delta_{i,j}$, the assignment of *nreplicas*(J_i) is as follows:

$$\forall i \in \{1,..,m\}: nreplicas\left(J_i\right) = m \qquad (1)$$

Having selected nreplicas(Ji) = m, the issue of selecting i,j will now be considered. Clearly, since a node i transmits nreplicas(Ji) replicas, it is necessary to specify nreplicas(Ji) – 1 values of i,j for node i. Consider the time span starting from when an application requests to transmit on a node until the last replica has finished its transmission on that node. The maximum duration of this time span over all nodes is z (as mentioned in Section 2.1). Figure 2 illustrates this. Clearly, we wish to minimize z. This can be formulated as a mixed linear/quadratic optimization problem. Therefore, the objective is to minimize z subject to:

$$\begin{gathered} \forall i \in \{1,..,m\}: \sum_{j=1}^{nreplicas(J_i)-1} \Delta_{i,j} + 1 \leq z \\ \forall i \in \{1,..,m\}, \forall j \in \{1,..,nreplicas(\tau_i)-1\}: 0 \leq \Delta_i \end{gathered} \qquad (2)$$

and (1), and subject to an additional third constraint that will be described now. Let *u* and *v* denote the indices of two nodes that may transmit. Hence, *u* and *v* belong to the set {1..*m*}. Let j_u and j_v denote the indices of the first replica of the sequence of replicas transmitted in nodes N_u and N_v, respectively. Hence j_u belongs to {1..*nreplicas*(J_u)–1} and j_v belongs to {1..*nreplicas*(J_v)–1}. Let l_u and l_v denote the lengths of these subsequences in terms of the number of replicas. l_u should be selected such that $j_u + (l_u - 1) \leq$ *nreplicas*(J_u) – 1. Analogous for l_v. Hence, l_u belongs to {1.. *nreplicas*(J_u) – j_u} and l_v belongs to {1.. *nreplicas*(J_v) – j_v}. We say that a combination of u, v, j_u, j_v, l_u, l_v is valid if: (i) these 6

variables are within their ranges; and (ii) $u \neq v \wedge (j_u \neq j_v \vee l_u \neq l_v)$. For every valid combination of u, v, j_u, j_v, l_u, l_v, the optimization problem must respect the following constraint:

$$\left(\left(\sum_{j=j_u}^{j_u+l_u-1}\Delta_{u,j}\right)-\left(\sum_{j=j_v}^{j_v+l_v-1}\Delta_{v,j}\right)\right)^2 \geq 2^2 \qquad (3)$$

Intuitively, (3) states that there is no sum of consecutive Δ*:s* on node *u* which is equal to a consecutive sum of Δ*:s* on node *v*. In addition, the difference is larger than 2; this implies that it is enough to be sure that there is no collision. (To understand why the difference must be 2, consider the following system: $m = 2$, *nreplicas*(J_1) = 2 and *nreplicas*(J_2) = 2 and $\Delta_{1,1} = 2$ and $\Delta_{2,1} = 3.98$, and J_1 arrives at time 0.99 and J_2 arrives at time 0. Then the first replica of J_1 and J_2 will collide, and the second replicas of J_1 and J_2 will collide as well. One can see that the sum of Δ*:s* must differ by the duration of two.).

Therefore, (3) states that at most one replica from node *u* can collide with a replica from node *v*. Hence, of those *nreplicas*(J_u) replicas sent from node *u*, at most $m - 1$ of them can collide. Naturally, this permits stating Theorem 1 below.

Theorem 1. If the differences between transmission start times of replicas are selected according to (1)-(3), then it holds that: (i) for every node *i*, at least one replica does not collide; and (ii) the time from when an application requests to transmit on node *i* until the last replica is transmitted on node *i* is at most *z*.

Proof: Follows from the discussion above. □

We will now illustrate the use of (1)-(3) in Example 1.

Example 1. Consider $m = 4$ to be solved using (1)-(3). The solution that is obtained is as follows:

$$\Delta_{1,1}=6 \quad \Delta_{1,2}=16 \quad \Delta_{1,3}=16$$
$$\Delta_{2,1}=8 \quad \Delta_{2,2}=18 \quad \Delta_{2,3}=8$$
$$\Delta_{3,1}=10 \quad \Delta_{3,2}=20 \quad \Delta_{3,3}=10$$
$$\Delta_{4,1}=12 \quad \Delta_{4,2}=12 \quad \Delta_{4,3}=12$$

This is illustrated in Figure 2. □

It is easily perceived that the number of inequalities in (3) grows as $O(m^6)$. Hence, it is only possible to solve small problems with this approach. (There were 232 constraints for $m = 4$ and 3411 constraints for $m = 6$. We used a modeling tool (AMPL [9]) and a back-end solver (LOQO [10]), and with these tools it was only possible to solve (1)-(3) for $m \leq 6$.) Many interesting systems are larger though. For those systems the optimization problem phrased in (1)-(3) simply cannot be solved because the number of inequalities in (3) is too large. (The extended version of this paper [11] presents techniques that find Δ*:s* for $m \leq 100$ by trading off the optimality of *z*.)

3 Sporadic Message Streams

Let us now consider that traffic is characterized by the *sporadic model* [12]. Each node has exactly one message stream. Node N_i is assigned the message stream τ_i. This message stream makes an infinite sequence of requests, and for each request, the message stream requests to transmit a message. The exact time of a request is unknown before run-time and the MAC protocol only knows about the time of the request when it occurs. But for every message stream τ_i there is at least T_i time units between two consecutive requests in τ_i and the MAC protocol knows all T_i. For every such request, the MAC protocol must finish the transmission of one replica of a message from stream τ_i without collisions at most T_i time units after the request. If this is the case, then we say that deadlines are met; otherwise a deadline is missed. Naturally, we assume $0 \leq T_i$.

From Section 2.3 it results that the maximum time it takes from when a message requests to send until the MAC protocol has transmitted a collision-free replica is *z*, if a message stream only makes a single request. Based on this, it would be tempting to think that if $\forall i \in \{1..m\}$: $z \leq T_i$ then all deadlines are met. Unfortunately, this is false, as illustrated by Figure 3, even if $T_1 = T_2 = T_2 = \ldots = T_m$. A correct schedulability analysis is given now.

Let w_i be defined as:

$$w_i = \sum_{j=1}^{nreplicas(\tau_i)-1}\Delta_{i,j} + 1 \qquad (4)$$

Theorem 2. If (Δ:*s* satisfy (1)-(3))$\wedge$(w_i is computed according to (4))$\wedge$($w_i \leq T_i$)$\wedge$($\forall k, k \neq i$: $w_i \leq T_k - w_k - 1$) then every message released from τ_i transmits at least one replica collision-free at most T_i time units after the message transmission request occurred.

Proof: Follows from the fact that during a time interval of duration $T_k - w_k - 1$, message stream τ_k can release at most one message. □

4 Implementation and Experiments

Having seen that the replication scheme can guarantee that at least one replica is collision-free in theory, we now turn to practice. We want to test the following hypotheses:

1. The replication scheme is easy to implement.
2. The number of lost or corrupted messages at the receiver is smaller when the replication scheme in this paper is used, as compared to a replication scheme with random pauses. This applies even if the random scheme transmits only a single replica per message.

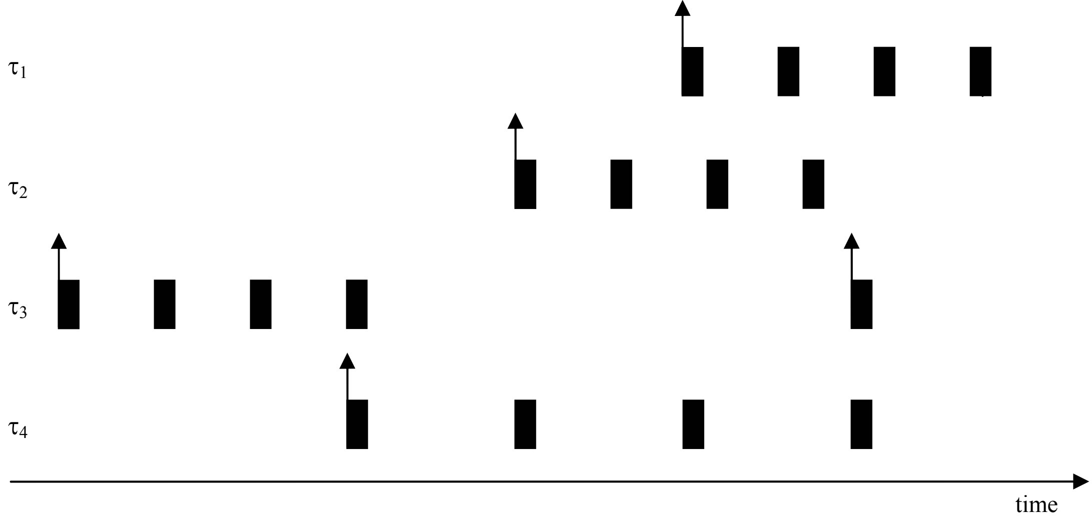


Fig. 3. Consider *Δ:s* that are selected based on the assumption a transmission request on a node occurs at most once. If these *Δ:s* are used for sporadic message streams with $T_1 = T_2 = T_3 = T_4 = z$ then a deadline miss can occur. All replicas from τ_4 collide and τ_4 misses its deadline.

3. The replication scheme guarantees that for each message, at least one replica is indeed collision-free.
4. If a link is bidirectional then our replication scheme can be extended so that it still offers a bounded number of collisions but it also has a low average-case overhead.

In order to test these hypotheses, we implement the replication protocol both on a real platform and use simulation (for details, see [11]). The following sections describe the implementation, experimental setup and results obtained. For these experiments, we used more than 6 computer nodes and hence the optimal algorithm described in Section 2.3. could not be used. We developed a heuristic algorithm (see [11]) for assigning Δ:*s* such that (1)-(3) are satisfied.

4.1 Implementation and Experimental Setup

The replication protocol was implemented on the MicaZ platform [13] and this implementation was dubbed HYDRA. MicaZ is a platform offering a low power microcontroller, 128 Kbytes of program flash memory and an IEEE 802.15.4 compliant radio transceiver, capable of 250 kbps data rate. The MicaZ supports running TinyOS [14] an open-source operating system. This platform was found to be attractive for the implementation of our experiments because of some particularly relevant characteristics: (i) it allowed us to replace the MAC protocol; (ii) the timers available where reasonably precise for our application; (iii) the radio transceiver makes automatic CRC checks and inserts a flag indicating the result of this check along with the packet, and (iv) the spread spectrum modulation used makes data frames resistant to noise and distortion. Hence, collisions due to medium access are the main source of lost frames or corrupted frames.

The experimental application setup consisted of one receiving node and a many sending nodes. Efforts where made such that the experiments took place under a similar, noise-free, environment. The sending nodes send messages with sequence numbers so that the receiving node detects when a message has been lost. Additionally, the receiver collected other statistics, such as total number of replicas and redundant replicas received (by redundant replicas we mean replicas for which a previous replica of the same message has already been received). The time to transmit a replica is 928μs. So, we let one time unit represent 1 ms to improve robustness against propagation delay and clock inaccuracy.

First, to acquire the probability that a replica is not correctly received (this is due to noise or distortion), we set up a scenario with one sending node (N1) and one receiving node (N2). Node N1 transmitted 2 replicas per message and N2 gathered statistics on the number of received replicas. We obtained that the probability of having a replica lost is approximately 0.002737%. If the events "a replica is lost" were independent, we would expect that the probability that two consecutive replicas are lost is 0.000027372. Hence we would expect the probability that a message was lost is 0.000027372 as well. However, we observed a 0.00153% probability for messages loss; this indicates that errors are correlated, which was expected.

After that, we ran experiments with different number of nodes, for three different MAC protocols: (i) one where we use our scheme with deterministic Δ:*s* (HYDRA); (ii) another where we used a similar scheme, but where the Δ:*s* were random variables within an

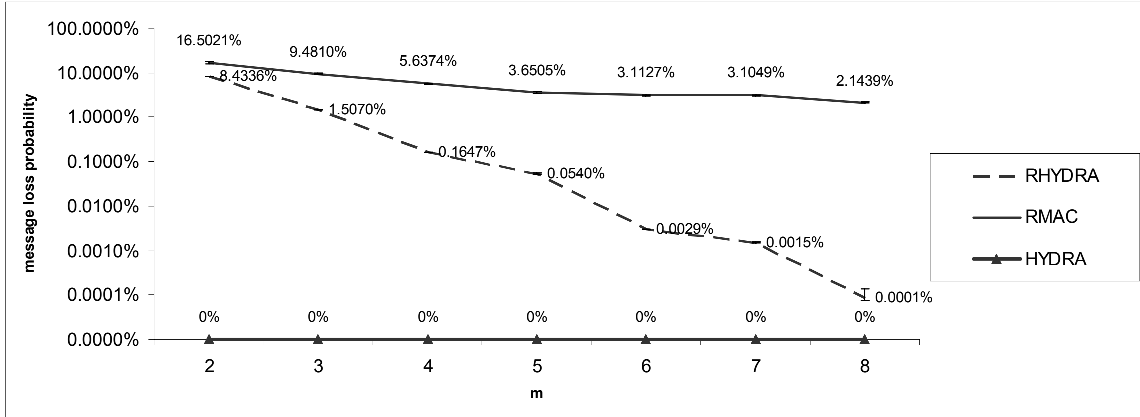


Fig. 4. Message loss ratio in simulation

interval between 1 ms and $(T_i - 1)/(nreplicas(\tau_i) - 1)$ time units, which was named Random HYDRA (RHYDRA) and (iii) finally a third MAC protocol where only one replica is sent at a random time within the interval $[0, T_i - 1]$ time units after the message was requested, which will be referred to as Random MAC (RMAC). The Δ:*s* were obtained from a close-to-optimal algorithm (see [11] for details). From these Δ:*s*, we derived z and T_i. The application on N_i generated message transmission requests such that the time between two consequtive requests is a uniform random variable with minimum T_i and maximum $T_i \times 1.25$.

The experiments where performed until each node transmitted 100000 messages, for $m = 2$ and $m = 4$. The resulting message loss rate is shown in Figure 5, which is presented in a logarithmic scale. By these results, we can observe that HYDRA obtained a message loss rate always better to the replica loss rate (0.002737%) previously obtained, indicating that noise was the cause for application message loss.

Performing statistically significant experiments with the actual implementations was very time consuming. Therefore, in order to test our protocol further, a simulation model for the protocol in OMNeT++ [15] was implemented. With this model we study the message loss ratio for different numbers of nodes with HYDRA, RHYDRA and RMAC (see [11] for details). The simulator assumes that replicas cannot get lost or corrupted due to noise, but it does model collisions which is the only source of lost messages.

All simulations were executed for a length of 10 simulated hours. For simulations involving random numbers generation, several independent runs were executed to verify the statistical validity of the results. The results of the simulations are given in Figure 4 with respective error bars which are mostly not visible due to the small variation found throughout the simulation runs.

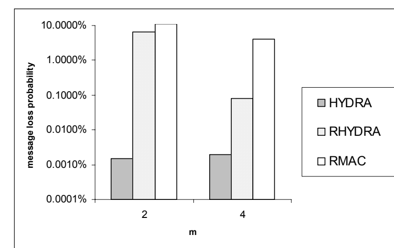


Fig. 5. Message loss ratio of experiments with MicaZ platforms

Observe that the application message loss for the scheme using deterministic Δ:s is always zero. This is expected as the simulation only models collisions, no noise in transmission was introduced, whereas the other schemes suffer from application message loss.

4.2 Support of Hypotheses

§Hypothesis 1. In order to test Hypothesis 1 the time required to implement HYDRA was measured. We spent approximately 3 days on implementing the protocol. Almost a third of this time was spent on getting familiar with the platform details. The time for coding the protocol was less than a day and we encountered no relevant bugs that were related to the implementation of the protocol. However, we encountered and fixed some bugs related to the platform. This suggests that Hypothesis 1 withstood our test.

§Hypothesis 2. The experiments presented in Section 4.1. corroborate Hypothesis 2.

§Hypothesis 3. Testing Hypothesis 3 is difficult because it is difficult to know if a lost frame is due to a collision or due to noise/distortion. Corrupt CRC may be because of noise or it may be because of collisions. Based on the experiments with the actual implementation of HYDRA in Section 4.1, it results that the number of lost messages is less than the probability

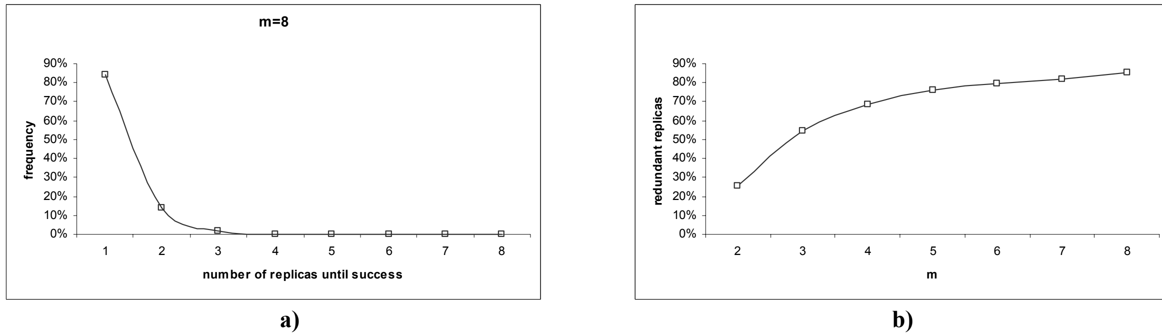


Fig. 6. The frequency of the number of necessary replicas and variation of the number of redundant replicas with *m*.

of a single message with a single sender being lost; this corroborates our hypothesis that the implementation of our protocol indeed guarantees that at least one replica is collision-free. Furthermore, we have run simulations during a period of 100 simulated hours for the scheme using deterministic Δ:s for $2 \leq m \leq 8$ and found that no application messages were lost during these simulation runs. This suggests that Hypothesis 3 withstood our test.

§Hypothesis 4. In order to test Hypothesis 4, we considered the simulation experiments used to test Hypothesis 3 and acquired both the frequency of the number of replicas necessary until the first replica is transmitted without collision (Figure 6a) and the number of redundant replicas for $2 \leq m \leq 8$ (Figure 6b). Observe, in Figure 6b, that for the case with $m = 8$ we obtain that approximately 84% of the first replicas of a message are collision-free. Hence, if most (but not all) links are bidirectional and we would have used a scheme where the receiver sends an acknowledgement when it receives the first successful replica then in approximately 84% of the cases the sender N_s only needs to send one replica. Hence, in 84% of the cases, N_s can send 7 non real-time messages instead of the replicas that N_s would normally send. This discussion supports, Hypothesis 4.

In order for the acknowledgement scheme described above to be efficient, it is necessary that the time required to send acknowledgements is negligible. Nonetheless, it could easily been the case by using longer packets (say 1500 bytes) for data and short packets (say 20 bytes) for the acknowledgements. But unfortunately this is not supported by our experimental platform so we did not implement it.

5 Discussion and Previous Work

Bidirectional links are useful for MAC and routing protocols. Let us categorize a MAC protocol based on whether it can suffer from collisions. If it can suffer from collisions then a sender typically retransmits data packets until it receives an acknowledgement from the intended receiver. Typically the data and the acknowledgement are transmitted on the same link, so this requires bidirectional links. This is exemplified by ALOHA [16] and some CSMA/CA protocols. MAC protocols that are collision-free typically rely on that senders receive feedback from the intended receiver. Some protocols, such as MACA [17] do this using an RTS/CTS exchange before the data packet is sent. In other protocols, a receiver sends a busy tone when it receives a packet and other senders can hear it, thus avoiding a collision. Common to all these MAC protocols is that they depend on bidirectional links. Routing algorithms also typically assume that links are bidirectional, being one notable exception the Dynamic Source Routing (DSR) [18]. We can conclude that the current communication protocols are heavily dependent on bidirectional links.

Unfortunately, unidirectional links are not rare and they are caused by a variety of reasons such as: (i) differences in antenna and transceivers even from the same type of devices; (ii) differences in the voltage levels due to different amounts of stored energy in the battery; (iii) different properties of the medium in different directions (anisotropic medium) and (iv) different interferences from neighboring nodes.

Given that protocol stacks tend to be implemented based on the assumption that unidirectional links do not exist, three techniques have been used to "hide" the unidirectional links: (i) *tunneling*; (ii) *blacklisting* and (iii) *transmission power increase*. If a link from node *u* to *v* is unidirectional, the tunneling approach attempts to find a path from *v* to *u* and give higher level protocols the illusion of a link from *v* to *u*. In order to achieve this, some routing functionality has to be performed at the lower layers of the protocol stack [19]. Packets sent across the tunnel have larger delays because they have to cross several hops. This is not too important though, because often the tunnel is used only for acknowledgements to packets that were sent across the unidirectional link. It is important however to avoid the *ACK explosion* [20]. Consider a unidirectional link from node N_u to node N_v. Consider also that there is a path from N_v to N_u. A data message has been sent across the link N_u to N_v and now the node N_v should send an ACK across the path back to N_u. However, the path from N_v to N_u contains a unidirectional link too. This link is from node N_x to N_y. When a packet has crossed the hop from

N_x to N_y, node N_y should send an ACK to N_x. In order to do this, it may have to find a path to N_x. It is possible that the path from N_y to N_x uses the link from N_u to N_v. This may generate an ACK from N_u to N_v and this process continues forever.

The technique of blacklisting detects unidirectional links when sending data messages, and does not use them in the future. The technique "hello" is similar but here "hello" messages are exchanged so a node i knows about the existence of a neighbor and whether they can hear i. This exchange is periodic and occurs regardless of whether the nodes are involved in routing data traffic or not. These techniques are sometimes called *ignoring* [21] or *check symmetry* [6]. Yet another technique to ignore unidirectional links is to treat it as a fault. This technique has been applied in conjunction with Ad-hoc On-Demand Distance Vector Routing (AODV) and it works as follows. When a source node attempts to find a route to the destination, it floods the network with Route-Request (RREQ) packets. In the normal AODV when RREQ packet reaches a node which knows a route to the destination, this node sends Route Reply (RREP) back on the same paths as the RREQ was sent on. With the normal AODV, RREP would fail on unidirectional links but instead this technique attempts to find a new path back to the source. When it finds a node with RREQ it knows a route back to the source node [22]. A similar scheme was proposed in [6] called *Bidirectional flooding*. Another technique (which we call "transmission power increase") permits a downstream node of a unidirectional link to temporarily increase its power for sending responses such as acknowledgements and clear-to-send [23]. This technique is based on the sender to piggyback its geographical position obtained by GPS and the receiver should use this information to calculate the distance, which in turn is used to know how much the transmission power should be increased. We think the idea of increasing transmission power is interesting but in [23] the authors do neither give any details on how this increase transmission power is computed nor state the assumed path loss. Common to these techniques is that they require no or minimal changes to routing protocols.

Several routing algorithms have been proposed for unidirectional links. A common challenge that faces routing with unidirectional links is *knowledge asymmetry*; that is, if a link from u to v is unidirectional, only v can detect the existence of the link (by hearing a broadcast from u) but u is the one that will use the knowledge of the link for routing purposes. One technique builds on *distance vector*. The classic distance vector algorithm maintains a vector at each node and this vector stores the hop count to every other node N_i and the next node that should be used for forwarding to this node N_i (sometimes a sequence number is added too; it is used for updates).

Consider a node N_u with a neighbor N_v. Node N_v knows a route to node N_w. The number of hops from N_u to N_w is no larger than the number of hops from N_v to N_w plus one. If the link N_u to N_v is bidirectional this fact can be easily exploited in the design of a routing protocol because the length of the route N_v to N_w can simply be communicated over one hop to N_u. However, if the link N_u to N_v is unidirectional this is more challenging.

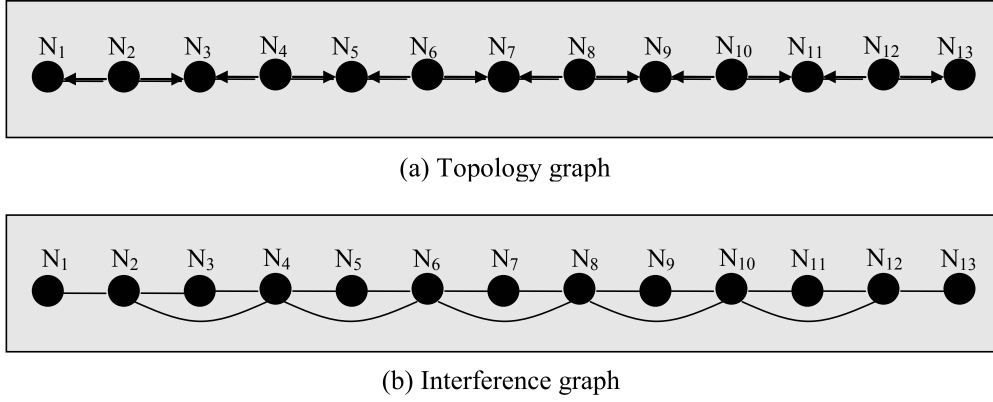
One extension of distance vector [21] however stores all distance vectors of all nodes in the network (hence it requires $O(m^2)$ storage). Another extension [24] sends information "downstream" until every node knows a circuit to itself. The node selects the shortest circuit and informs its upstream neighbors, and then the standard distance vector algorithm is used. Other techniques [19, 25] and [26] disseminate link state information across a limited number of hops. This is based on the assumption that the reverse path of a unidirectional link is short and this assumption has been supported empirically [19].

Pure link-state routing disseminates the topology information to all nodes and then the routes are calculated. This avoids the problem of asymmetric information (mentioned earlier) but the overhead of this scheme is large already.

In order to reduce the routing cost in networks with unidirectional links, it has been suggested that a subset of nodes should be selected and only they should maintain routing information about all nodes in the network. It is required that all nodes which are not in this subset have a link from the subset and a link to the subset. Algorithms for selecting this subset of nodes have been proposed and they have very low overhead [27].

It has often been pointed out that unidirectional links should be avoided altogether because existing MAC protocols cannot deal with them (as we already mentioned, MACA, which was the basis for the RTS/CTS exchange in IEEE 802.11, relies on bidirectional links). But recently, this view has been challenged. For example [28] mentioned that their routing protocol works well for multicast and that it could be used for unicast routing as well – if there was a MAC protocol for unidirectional links.

To the best of our knowledge, the only previous MAC protocol that work for unidirectional links require synchronized clocks and it suffers from (an unbounded number of) collisions [8]. The technique in [8] addresses medium access control on unidirectional links. The technique generates pseudo-random numbers on each node and these numbers act as priorities. Every node knows the seed of the pseudo-random numbers on other nodes and hence a node knows if it has a higher priority than its neighbors. If it has, then it is the winner; otherwise it is not a winner. If it is a winner then it transmits in that time slot. Every new time slot, a new pseudo-random number is generated. This protocol is designed to deal with hidden nodes in the following



(a) Topology graph

(b) Interference graph

Fig. 7. An example of how the performance of our MAC protocol can be significantly improved if the topology is known.

way: if a node N_i has a neighbor with higher priority two hops way then node N_i simply does not transmit. This scheme is collision-free but it depends on synchronized clocks. Our protocol does not require that.

In the theory we assumed that $prop = 0$. We can easily extend the theory for the case when $prop > 0$. We can do it as follows. Select the time unit such that $(1-prop)$ is the time it takes to transmit a replica. Hence, if $prop = 1\mu s$ and the time to transmit a replica is 1 ms, then let 1.001 ms denote a time unit.

In this paper, we assumed topology is not known. However, if the topology is known we can perform significantly better (assuming that we also know the interference graph). Every node in the topology graph also exists in the interference graph. The links in the interference graph are non-directed. The links in the interference graph cannot simply be computed from the topology graph. However, there are some links in the interference graph that are necessary. Consider two nodes in the topology graph N_i and N_j. If there is a link from N_i to N_j or from N_j to N_i, then there is a link between N_i and N_j in the interference graph as well. If there is a node N_k with a link from N_i to N_k and a link from N_j to N_k then there is a link between N_i and N_j in the interference graph as well. From the interference graph, it is possible calculate Δ:s that cause a significant decrease in the overhead. This is illustrated in Example 2.

Example 2. Consider $m = 13$ nodes ordered in a line such that every node with an even index has two outgoing links; node i has a link to node i-1 and node i+1. This is illustrated in Figure 7a. If the topology is unknown, then we must assume that all 13 nodes can transmit simultaneously and can collide. A solution to (1),(2),(3) in Section 2.3 is the following Δ:s: $\Delta_1 = 22$, $\Delta_2=26$, $\Delta_3 = 34$, $\Delta_4 = 38$, $\Delta_5 = 46$, $\Delta_6 = 58$, $\Delta_7 = 62$, $\Delta_8 = 74$, $\Delta_9 = 82$, $\Delta_{10} = 86$, $\Delta_{11} = 94$, $\Delta_{12} = 106$, $\Delta_{13} = 118$ and $z = 1417$. From the interference graph shown in Figure 7b, we observe that every node has at most 4 links. This gives us $m = 5$, and we calculate the following Δ:s: 6, 10, 14, 22, 26. Now we can assign $\Delta_1 = 6$, $\Delta_2 = 10$, $\Delta_3 = 14$, $\Delta_4 = 22$, $\Delta_5 = 26$, $\Delta_6 = 6$, $\Delta_7 = 10$, $\Delta_8 = 14$, $\Delta_9 = 22$, $\Delta_{10} = 26$, $\Delta_{11} = 6$, $\Delta_{12} = 10$, $\Delta_{13} = 14$. Observe that we reuse Δ:s and this does not cause any collisions. In this way, we obtain $z = 105$, which is significantly lower. □

In general this requires solving the problem Achromatic Number which is known to be NP-hard (see page 191 in [29]) but several approximation algorithms are available. We can see from Figure 7 that z is unaffected by the size of the network; only the number of neighbors 2-hops away matters. Hence, this approach is efficient in large networks if they are not dense.

6 Conclusions

We have presented the first MAC protocol that can guarantee that the time from when an application requests to transmit until the message is transmitted is bounded even in the presence of unidirectional links and without using synchronized clocks or taking advantage of topology knowledge. We have implemented the protocol and observed that: (i) the effort required to implement it is small; (ii) by observing the number of lost messages we found that the implementation guaranteed that at least one replica of a message is collision-free and (iii) the number of lost messages at the receiver is significantly lower using our protocol than a replication scheme with random delays between replicas. We also run a scheme with random time for transmission with only one replica. We expect such a scheme to perform similarly to ALOHA [16], and we found that our protocol performed significantly better.

Acknowledgements

This work was partially funded by Fundação para Ciência e Tecnologia (FCT) and the Network of Excellence on Embedded Systems Design ARTIST2 (IST-004527).

References

[1] A. Cerpa, J. L. Wong, L. Kuang, M. Potkonjak, and D. Estrin, "Statistical Model of Lossy Links in Wireless Sensor Networks," in Information Processing in Sensor Networks (IPSN'05). Los Angeles, California, USA, 2005.

[2] A. Cerpa, N. Busek, and D. Estrin, "SCALE: A Tool for Simple Connectivity Assessment in Lossy Environments," UCLA Center for Embedded Network Sensing (CENS) Technical report 0021, September 2003.

[3] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in Conference On Embedded Networked Sensor System. Los Angeles, California, USA, 2003.

[4] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker, "Complex Behavior at Scale: An Experimental Study of Low-Power Wireless Sensor Networks," UCLA Technical Report CSD-TR 02-0013, February 2002.

[5] D. Kotz, C. Newport, R. Gray, J. Liu, Y. Yuan, and C. Ellliot, "Experimental Evaluation of Wireless Simulation Assumptions," in International Workshop on Modelling Analysis and Simulation of Wireless and Mobile Systems, 2004.

[6] G. Zhou, T. He, S. Krishnamurthy, and J. Stankovic, "Impact of Radio Irregularities on Wireless Sensor Networks," in International Conference on Mobile Systems, Applications, and Services, 2004.

[7] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in On Embedded Networked Sensor Systems,. Los Angeles, California, 2003.

[8] L. Bao and J. J. Garcia-Luna-Aceves, "Channel access scheduling in Ad Hoc networks with unidirectional links," in Workshop on Discrete Algothrithms and Methods for MOBILE Computing and Communications. Rome, Italy, 2001.

[9] "AMPL, www.ampl.com."

[10] "LOQO, http://www.princeton.edu/~rvdb/."

[11] B. Andersson, N. Pereira, and E. Tovar, "Delay-Bounded Medium Access for Unidirectional Wireless Links," Institute Polytechnic Porto, Porto, Portugal HURRAY-TR-060701, Available at http://www.hurray.isep.ipp.pt/asp/show_doc.asp?id=255, July 2006.

[12] A. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," in Electrical Engineering and Computer Science. Cambridge, Massachusetts: Massachusetts Institute of Technology, 1983.

[13] "Crossbow, "MICAz - Wireless Measurement System Product Datasheet," 2005.

[14] J. Hill, "System Architecture for Wireless Sensor Networks," in Computer Science Department: University of California, Berkeley, 2003.

[15] A. Varga, OMNeT++ Discrete Event Simulation System. Tech. University of Budapest, Budapest, 2003.

[16] N. Abrahamson, "The ALOHA system - another alternative for computer communications," in 1970 fall joint computer communications, AFIPS Conference Proceedings. Montvale., 1970.

[17] P. Karn, "MACA - A New Channel Access Method for Packet Radio," presented at ARRL/CRRL Amateur Radio 9th Computer Networking Conference, 1990.

[18] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in Mobile Computing, T. I. a. H. Korth, Ed.: Kluwer Academic Publishers, 1996.

[19] V. Ramasubramanian, R. Chandra, and D. Mossé, "Providing a Bidirectional Abstraction for Unidirectional Ad Hoc Networks," in IEEE INFOCOM. New York NY, 2002.

[20] S. Nesargi and R. Prakash, "A Tunneling Approach to Routing with Unidirectional Links in Mobile Ad-Hoc Networks," in IEEE International Conference on Computer Communications and Networks (ICCCN). Las Vegas, 2000.

[21] R. Prakash, "A routing algorithm for wireless ad hoc networks with unidirectional links," Wireless Networks, vol. 7, pp. 617 - 625, 2001.

[22] M. K. Marina and S. R. Das, "Routing performance in the presence of unidirectional links in multihop wireless networks," in 3rd ACM international symposium on Mobile ad hoc networking & computing. Lausanne, Switzerland, 2002.

[23] D. Kim, C.-K. Toh, and Y. Choi, "GAHA and GAPA : Two Link-level Approaches for Supporting Link Asymmetry in Mobile Ad Hoc Networks," IEICE Transaction on Communication, vol. E-86B, pp. 1297-1306, 2003.

[24] M. Gerla, L. Kleinrock, and Y. Afek, "A Distributed Routing Algorithm for Unidirectional Networks," in IEEE GLOBECOM. . 1983.

[25] L. Bao and J. J. Garcia-Luna-Aceves, "Unidirectional Link-State Routing with Propagation Control," in IEEE Mobile Multimedia Communications (MoMuC). Tokyo, Japan, 2000.

[26] T. Ernst, "Dynamic Routing in Networks with Unidirectional Links," INRIA, Sophia Antipolis 1997.

[27] J. Wu and H. Li, "Domination and Its Applications in Ad Hoc Wireless Networks with Unidirectional Links," in International Conference on Parallel Processing. Toronto, Ontario, Canada, 2000.

[28] M. Gerla, Y.-Z. Lee, J.-S. Park, and Y. Yi, "On Demand Multicast Routing with Unidirectional Links," in IEEE Wireless Communications & Networking Conference (WCNC). New Orleans, LA, USA, 2005.

[29] M. R. Garey and D. S. Johnson, Computers and Intractability A Guide to the Theory of NP-Completeness. New York: W. H. Freeman and Company, 1979.

Tolerating Arbitrary Failures in a Master-Slave Clock-Rate Correction Mechanism for Time-Triggered Fault-Tolerant Distributed Systems with Atomic Broadcast

Astrit Ademaj Alexander Hanzlik Hermann Kopetz
Vienna University of Technology
Real-Time Systems Group
Treitlstr. 3/182-1, A-1040 Vienna, Austria
{ademaj,hanzlik,hk}@vmars.tuwien.ac.at

Abstract

In a previous work we have shown that by deploying a node with a high-quality oscillator (rate-master node) in each cluster of a real-time system, we can integrate internal and external clock synchronization by a combination of a distributed mechanism for clock state correction with a master-slave mechanism for clock rate correction. By means of hardware and simulation experiments we have shown that this combination improves the precision of the global time base in single- and multi-cluster systems while reducing the need for high-quality oscillators for non rate-master nodes. Previous experimental results have shown that transient fail-silent failures of the rate-master node will not affect the operation of the distributed clock state correction algorithm, and the cluster will remain internally synchronized. In this paper we consider all possible failure modes of the rate-master node by taking the system structure into account, and present a solution that tolerates arbitrary rate-master failures by using replicated rate-master nodes. Possible arbitrary failure modes are listed in the paper where a main part of them is handled by the fault-tolerant mechanisms of the system architecture, whereas the remaining failure modes are handled by a fault-tolerant rate correction mechanism.

1. Introduction

The temporal accuracy of an information in a distributed real-time system depends on the precision of the global view of time. This precision depends on the jitter of the message transmission delay, the clock drifts and the clock synchronization mechanism. A predictable real-time communication system must guarantee message transmission within a constant transmission delay and with bounded jitter [8].

The duration of the message transmission over the network depends on the assumptions made about the network traffic. In principle we can distinguish between two different scenarios: competing senders or cooperative senders. If senders are competing (as in standard Ethernet or CAN) there is always the possibility that many messages are sent to a single receiver simultaneously. There are two possibilities to resolve this conflict: back-pressure flow control to the sender or the storage of the messages within the network. Both alternatives involve increased message transmission jitter which is unsatisfactory from the point of view of real-time performance for applications that demand highly precise time measurements.

The solution of cooperative senders (which deploys a communication schedule that ensures conflict free message transmissions, e.g. a TDMA scheme) provides constant transmission delays and bounded jitter. This paper focuses on the generation of a fault-tolerant global time base of high precision (in the range of half a microsecond) in a distributed real-time system with cooperating senders within constraints imposed by a market of mass production.

One challenge in a mass production market, such as the emerging automotive market for drive-by-wire systems, is to provide ultra-high dependability at affordable cost. In such a market the cost of every single component is scrutinized in order to find alternatives that are less costly. One target for cost savings in the distributed control system for a mass production market is the oscillator at a computer node. Today upscale cars have more than 50 computer nodes, each one with its own local oscillator.

The precision (the maximum deviation between any two clocks in an ensemble) of the global time base is the key factor to the accuracy of real-time information in a distributed system. The precision depends, among other factors, on the quality (and price) of the given oscillators.

In [10] we have presented an algorithm that establishes and maintains a global time base of high precision in a distributed system while reducing the need for high-quality oscillators. This algorithm introduces the notion of a rate master node that dictates the rate of the global time base in a cluster. However, the approach presented in [10] was only resilient against *transient fail-silent* failures of the rate

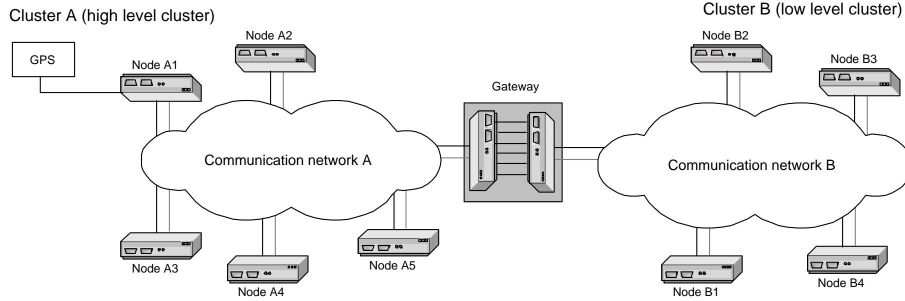


Figure 1. Two-cluster system

master node. This failure mode assumption is too weak for a safety critical environment, where a node may exhibit arbitrary faulty behavior.

In this paper we consider the possible node failure modes in a system with cooperating senders and present a solution for tolerating this restricted set of failure modes imposed by the presented system model.

2. System Structure

We assume that a distributed real-time computer system can be built from one or more *clusters*. A cluster consists of a set of node computers that communicate with each other using a shared communication medium. Figure 1 shows a distributed real-time computer system consisting of two clusters.

2.1. Node

Each node contains a host computer that executes the application program and a communication controller for message-based communication with the other nodes. Every node maintains an oscillator that drives a clock local to this node. We call this clock the *local clock* of the node.

2.2. Communication

All nodes communicate by periodic message exchange using an *a-priori* known global *communication schedule* available at all nodes. The communication schedule is a periodic and static scheme based on a fault-tolerant global time base. It contains, among other information, which node is allowed to send at which point in global time. One period of communication is referred to as *TDMA round*. Every node is allowed to transmit a message at least one message each TDMA round. A message sent from a correct sender will arrive correctly at all correct receivers at about the same point in time.

2.3. Oscillators

We assume that every oscillator of a node can be characterized by a *nominal frequency* and a maximum *drift rate* that defines a *permitted drift window* around this nominal frequency [8]. Every device that oscillates within this permitted drift window is classified as a *correct* oscillator, otherwise as a *faulty* oscillator. The data concerning the nominal frequencies and the permitted drift windows are supplied by the oscillator manufacturer.

2.4. Timer Control Unit

We assume that each node has a *timer control unit* (TCU) in its communication controller that is responsible for the generation of a local view of global time. Each node has its local clock which measures the time with the granularity that we call *microtick* (μt). The granularity of the node-local view of global time is called a *macrotick* (MT). To maintain a reasonable global time base, all nodes must be synchronized within a precision of one MT [8]. One macrotick is made from a number of microticks initially derived from the nominal frequency of the oscillator. The number of microticks (μt) per macrotick (MT) is denoted as *microtick-macrotick conversion factor* (MMCF). By changing the MMCF the frequency divider shown in Figure 2 can be adjusted in order to manipulate the generation rate of the MT, the node-local representation of global time.

A clock state synchronization algorithm (executed in each node) periodically calculates a clock state correction term for the local clock (in terms of microticks) and stores the value in the Macrotick Correction Term (MTCT) register (see Figure 2) of the communication controller. A hardware mechanism assures that the value of the MTCT register is added to the microtick counter in order to correct the clock deviations among the nodes in the cluster.

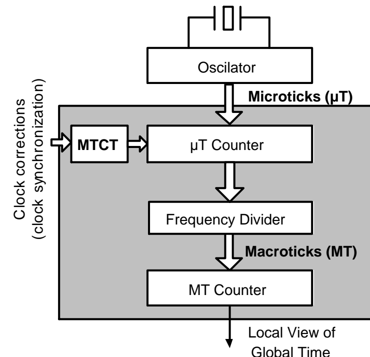


Figure 2. Timer unit of a communication controller

The clock state correction algorithm periodically corrects the local clock of the node at so called *state synchronization instants* contained in the communication schedule (Section 2.2). The time interval between two state synchronization instants is denoted as *state synchronization interval*. Because of the imperfect manufacturing process of oscillators, each clock has a slightly different drift rate and therefore each node will have a slightly different microtick duration. During a synchronization interval the local clocks of the nodes are running free and may thus deviate from each other due to different clock drifts. These deviations are corrected at the next synchronization instant by the fault-tolerant clock state synchronization algorithm.

2.5. Time difference capturing

A node has to know the deviation of its local clock to the local clocks of other nodes to be able to synchronize to these nodes. The process of reading a remote clock is referred to as *time difference capturing*. The expected message receive instants of all messages are *a-priori* known to all nodes (Section 2.2). The difference between the expected message receive instant and the actual message receive instant is measured by a *time-difference capturing unit* of the communication controller. The measured differences in terms of microticks (μt) are used by the clock state synchronization algorithm.

2.6. Fault Hypothesis

The fault hypothesis for a system specifies the smallest unit of failure (fault containment region - FCR), plus the type, the number and the frequency of FCR failures that shall be tolerated. The fault hypothesis makes assumptions about the behavior of the system in the presence of faults. In our system model we consider either a computer node or a communication channel as a fault containment region. The fault hypothesis assumes at most one arbitrary faulty FCR at any point in time.

3. Clock Correction Mechanism

In our earlier work [10], we have proposed to establish the fault-tolerant global time base by a combination of two algorithms, a distributed *state synchronization algorithm* and a central *rate correction algorithm*. An example for the *state synchronization algorithm* is the fault-tolerant average (FTA) algorithm, the correctness of which has been established by formal analysis [16]. All clocks estimate their deviation from the other clocks and calculate a *clock state correction term* on the base of these estimates. At the end of each synchronization interval, all nodes apply the clock state correction term to their local clocks by means of state correction to get into agreement with the nodes in their cluster. This algorithm remains unchanged and therefore the arguments for its correctness remain valid.

In our system model we use at least one node (denoted as *rate-master node*) with a high-quality oscillator as a *rate-master clock*. Other nodes with low-quality oscillators, denoted as *time-keeping nodes*, correct their rate (by proper adjustment of the MMCF value of the TCU). In such a system model, the cluster drift rate is determined by the drift rate of the rate-master node [10].

The rate-master node has a high-quality clock that serves as a reference for the rate of all other clocks in the cluster. We assume that the rate-master clock has a drift-rate that is better than $10^{-6}s/s$ (that is the drift-rate of an ordinary wrist watch oscillator). The drift rate of the rate-master clock determines the drift rate of the whole cluster. Available field-data from the automotive industry shows that the permanent failure rate of oscillators is better than 100 FIT [14].

Time-keeping nodes establish the fault-tolerant global time base by a distributed clock state correction algorithm. In addition, they periodically adjust their rate to the rate of the rate-master clock by manipulation of their MMCF values according to their deviations from the rate-master node. The time-keeping nodes are assumed to have standard computer oscillators with a drift rate in the range of 10^{-4} s/s. The permanent and transient failure rates of the time-keeping clocks are assumed to be equal to those of the rate-master clock [10].

3.1. Rate Correction Algorithm

All nodes communicate by periodic exchange of messages using an *a-priori* known communication schedule. The instant when the message transmission has started is called *message send instant*. The instant of arrival of a message is called *message receive instant*. The expected message receive instants of all messages are *a-priori* known to all nodes. The difference between the expected message receive instant and the actual message receive instant is measured by a *time-difference capturing unit* of the communication controller. The differences are expressed in terms of microticks (μt) and are used by the FTA algorithm to perform clock state correction. These measurements are denoted as *time-difference capturing values* and they represent the difference in the local view of global time between the sender node and the receiver node at the instant when a message is transmitted by the sender node. A positive time-difference capturing value means that the receiver's clock is running faster than the sender's clock. Consequently, a negative time-difference capturing value means that the receiver's clock is running slower than the sender's clock. This information can be used to perform clock-rate correction. Based on the time-difference capturing values at the *message send instant* of messages sent by the rate-master node, time-keeping nodes can estimate the required rate correction. The time-difference capturing values depend on the length of the clock state synchronization interval and on the sending slot position of the rate-master node within the clock state synchronization interval. Thus, the time-keeping nodes

can only approximately estimate their rate deviation from the rate of the rate-master clock. Experimental results of the implementation of the mechanism presented above are given in [10] where it is shown that transient fail-silent failures of the rate-master node do not affect the precision of the cluster.

In this paper we go one step further and present a solution that tolerates arbitrary failure modes of rate-master nodes.

Furthermore, we analyze all possible failure modes (transient and permanent arbitrary failures) of the rate-master node within the given system model. As it will be discussed in the next section, a main part of failure modes is handled by the fault tolerance mechanisms of the system architecture (i.e. they need not to be handled by the rate-correction mechanism), whereas the remaining failure modes have to be handled by the fault-tolerant rate correction mechanism. The presented solution tolerates arbitrary FCR failures within the given time-triggered system model using atomic broadcast.

4. Rate-Master Failure Modes

We consider a clock synchronization algorithm in which the nodes do not explicitly send time information to other nodes. We assume a synchronous system that periodically transmits messages according to a static communication schedule that is *a-priori* known to all communication participants. Every node knows the expected arrival time of frames from other nodes. For a case study we use the Time-Triggered Architecture (TTA) [11]. In the TTA an incoming message is classified as correct if the syntax check is successfully passed (checksum, header structure, etc.) and if the message is received within the expected receive window. Node i knows *a-priori* the expected frame arrival time of the message sent by node j, say $Texp_i^j$. A message sent from node j that is received at node i (msg_i^j) will be classified as correct if the message is received within the receive window $[Texp_i^j - \Pi, Texp_i^j + \Pi]$, where Π is the precision of the global time base. Based on the difference between the expected frame arrival time and the actual frame arrival time, the nodes know the deviation of their own clocks from the clock of the sender node. We also assume that the minimum message transmission delays are known and constant between any two nodes. The jitter is bounded.

The TTA deploys a membership and a clique avoidance algorithm [3] that makes sure that all correct nodes reach a consensus about the operating state of all other nodes in the system. Moreover the TTA deploys either a star guardian (for a star topology) or a bus guardian (for a bus topology) that allows nodes to send messages only within their sending slot. Any message sent by a faulty node outside its sending slot will be blocked by the guardian [11].

In the following we consider all possible failure modes of the rate-master node and the time-keeping nodes with regard to the clock synchronization algorithm.

Time-keeping nodes are passive nodes regarding the clock rate correction. Thus, a failure of a time-keeping node cannot affect the proper operation of the rate-master clock.

Consistent detectable failures of the rate-master node occur when the rate master transmits a message which is syntactically incorrect. Message syntax failures are handled by the error detection mechanisms that check the structure and the content of the message headers and by comparing the checksum of the received data with the received checksum. All correct nodes consistently detect this failure of a rate-master node by means of the membership algorithm.

Fail-silent failures of the rate-master node occur when the rate-master node fails to transmit a message. This failure can occur either because of a fault in the timer unit of the controller, or in other functional units (e.g. in a faulty transmitter unit). This failure is consistently detected by all nodes.

Babbling idiot failures occur when a node tries to transmit a message outside its allocated sending slot. Such failures are avoided by the guardian mechanism [20]. In case that the guardian functionality is not provided, a babbling node can disturb the communication of the whole cluster. Therefore, we assume the guardian functionality in our system model. Because of the guardian, a faulty node that experiences babbling idiot failures causes that either no message from this node is received, or that other nodes receive invalid messages from this faulty node. This failure is consistently detected by all nodes.

Masquerading failures occur when a faulty node sends a message using the identity of any other node. Because of *a-priori* knowledge of the message schedule, each node expects a message from a specific node in a specific slot. A message that contains a node ID that does not match with its statically allocated sending slot will be detected as faulty. Another way for a node to try to masquerade within our system model is to send a wrong node ID (say n) in the sending slot of node n. This case is handled as a babbling idiot failure by the guardian. This failure is consistently detected by all nodes.

Drift rate changes of a rate-master node will change the drift rate of the whole cluster. Consider a scenario where a faulty rate-master node slowly changes its rate, such that the messages sent by the rate-master node are always received within the receive window of the other nodes. If this rate change is small enough, the other nodes will adjust their rate accordingly (the rates of the time-keeping nodes will follow the rate of the rate-master node). Eventually, the rates of all nodes will change slightly such that they slow-down or speed-up. Note that clock rate adjustment changes the macrotick (MT) duration as well. A continuous rate change into one direction will change the MT duration such that the operation of

the cluster with the new MT duration (granularity of the global time base) may eventually become impossible.

SOS (Slightly-Off-Specification) failures are failures where a message sent by a faulty node is received correctly by some nodes, and incorrectly by other nodes. We distinguish between SOS failures in the value domain, SOS failures in the time domain and SOS failures with respect to the start of frame transmission [1, 2]. The effect of all three SOS failure scenarios is the same in our system model. For example, in case of an SOS failure with respect to the start of frame transmission, the rate-master node sends a message that is received within the limit of the receive window of some nodes in the cluster, and slightly outside the receive window of other nodes in the cluster. An SOS failure causes an inconsistent view of the node states in the system by formation of cliques (nodes in one clique assume that the rate-master node is faulty, nodes in the other clique assume the rate-master node is correct). SOS failures are a subclass of Byzantine failures and, from the point of view of clock synchronization, they are the only possible Byzantine fault manifestation in our system model. Note that in our system model the rate-master node does not send any explicit time information. The measured time differences between the expected and actual frame arrival times of the messages sent by the rate-master node are used to perform master-slave clock-rate correction. SOS failures in the TTA are handled by the group membership algorithm [3].

The presented clock synchronization mechanism makes use of a voting algorithm explicitly for 2 rate-master nodes, denoted as *rate-master voting* (see Section 5). Using this voting mechanism, even systems that do not use a group membership algorithm will be able to have a consistent view of the state of the rate-master nodes. Thus, SOS failures of the rate-master node are consistently detected by all nodes.

To sum up, all arbitrary failure modes of the rate-master nodes with respect to the clock synchronization mechanisms within our system model can be classified into two groups:

- The rate-master node is consistently detected as faulty (it transmits an incorrect, an invalid or no message at all). From the point of view of the clock rate correction mechanism, all correct nodes will detect a faulty rate-master, and therefore will not perform clock-rate correction based on messages sent by the rate-master node.

- A slow change of the drift rate of the rate master will cause a change of the cluster drift rate (and a change of the granularity of the global time). From the point of view of the clock-rate correction mechanism (presented in [10]) the nodes will not detect a faulty rate master, and will continue to perform clock rate correction based on the messages sent by a faulty rate-master node.

5. Fault-Tolerant Rate Correction Algorithm

In this section we describe the new fault-tolerant rate correction algorithm for a single-cluster system. The fault hypothesis (Section 2.6) claims that the system is capable of tolerating a single arbitrary node failure at a time. Therefore, we replicate the rate-master node by adding a second rate-master node with a high-quality oscillator. In the following, we will denote the two rate-master nodes as *primary rate-master* (PRM) and *secondary rate-master* (SRM), respectively. At system startup, all nodes agree on the same PRM. The task of the SRM is to take over the role of the PRM once a majority of nodes in the cluster considers the PRM to be faulty. The decision which node is the rate-master is met by means of a *rate-master voting* mechanism.

All nodes periodically perform clock-state correction according to the clock state correction term delivered by the FTA algorithm. The time-keeping nodes (including the SRM) periodically perform clock rate correction according to the clock rate of the PRM determined from the time-difference capturing value obtained in the sending slot of the PRM. All nodes will perform rate correction by proper adjustment of the MMCF value as long as their MMCF value is within pre-defined bounds.

Rate-master voting. The PRM and the SRM node transmit at least one message per TDMA round. The time-keeping nodes and the SRM calculate a new MMCF value based on the time-difference capturing value from the PRM and apply the new MMCF value to their local clocks. If a node fails to receive a message from the PRM at the expected arrival time or if the new MMCF value is not within a pre-defined bound, the node considers the PRM to be *faulty*, otherwise to be *correct*. Each node maintains a two bit vector containing the value of the node that it has classified as the primary rate-master node. At its message send instant (once per TDMA round), each node adds its local view of who is the PRM to the message sent (the PRM always considers itself as *correct*). Upon reception of a message, each node increments a PRM_{ok} counter if the sender classifies the PRM to be *correct* (indicated by the rate-master voting information included in the message) and a PRM_{fail} counter otherwise. Both counters are set to 0 at system startup.

Rate-master agreement. At a pre-defined instant once per TDMA round (the *PRM membership point*), all nodes evaluate their local counters. If $PRM_{ok} \geq PRM_{fail}$, a majority of nodes considered the PRM to be *correct* during the last TDMA round and the current PRM is trusted until the next PRM membership point. If no majority for the current PRM is found, all nodes agree that the SRM becomes the new primary rate-master. The former PRM, which also evaluates its local counters, detects that it is no longer trusted by a majority of nodes and classifies itself

as SRM. All nodes clear the PRM_{ok} and PRM_{fail} counters and restart the rate-master voting algorithm. This rate-master voting algorithm is derived from the group membership algorithm of the TTP/C protocol [3]. A correctness proof of the TTP group membership algorithm can be found in [15].

Rate-master handover. After a new PRM has been voted, this new PRM slowly returns to its original (nominal) rate by periodically adding a pre-defined value to its current MMCF value until its MMCF value reaches its (offline defined) nominal value. Note that before the handover occurs, the current PRM was an SRM that has adjusted its clock rate to the rate of the former PRM. The new PRM considers itself *correct* and stepwise readjusts its MMCF to its nominal value to bring the clock rates of all nodes closer to the progression of realtime (a rate-master node has a more accurate oscillator than the time-keeping nodes).

The presented mechanism is capable of handling arbitrary rate-master node failures within our system model. As elaborated in Section 4, arbitrary failure modes within our system model can be classified into:

Consistent detectable rate-master failures. If the PRM fails to send a message in its sending slot, a majority of nodes (all correct nodes) will consistently agree on a new PRM according to the rate-master agreement algorithm[1].

Clock drift rate failures. If a clock reading from a PRM leads to a MMCF boundary violation at a majority of nodes, all nodes will consistently agree on a new PRM according to the rate-master agreement algorithm.

6. Tolerating Arbitrary Failures - Simulation Experiments

The rate-master voting algorithm has been validated by means of simulation experiments using SIDERA, a simulation model for time-triggered distributed systems [7]. In this section, we present the results of two typical validation experiments.

6.1. Experimental setup

Table 1 summarizes the system configuration used for the simulation experiments.

The 6 nodes are numbered from 0 to 5 and are assigned the following drift rates ρ_i from the clock drift rate window: $\rho_0 = -2 \times 10^{-5} s/s, \rho_1 = -1,2 \times 10^{-5} s/s$, $\rho_2 = -4 \times 10^{-6} s/s, \rho_3 = 4 \times 10^{-6} s/s$, $\rho_4 = 1,2 \times 10^{-5} s/s, \rho_5 = 2 \times 10^{-5} s/s$.

Node 2 and Node 3 are chosen as the PRM and the SRM, respectively (their clock drift rates are an order of magnitude better than those of the other clocks). The MMCF boundary is derived from the nominal MMCF and is equal for all nodes. This value for the MMCF boundary tolerates rate changes of the PRM of 10^{-4} s/s. The precision (the maximum deviation between any two clocks) of the cluster shall be guaranteed to be better than 10 microticks.

[1] In the TTA, SOS failures are resolved by the group membership and clique avoidance algorithm of the Time Triggered Protocol TTP [21].

Parameter	Value
Number of nodes	6
TDMA round length	12ms ($12 \times 10^{-3} s$)
Macrotick length	1 μsec ($10^{-6} s$)
nominal MMCF	20
Clock drift rate window	40ppm ($4 \times 10^{-5} s/s$)
MMCF boundary	0.2 ($[19,8; 20,2]$)
Primary Rate-Master	Node 2
Secondary Rate-Master	Node 3
Precision	10 microticks

Table 1. Cluster configuration

The figures used in the description of the simulation experiments consist of two windows each and show the following information: the x-axis denotes the progression of simulation time with the same granularity for all windows (i.e. events on the same x-coordinate in different windows happen at the same point in simulation time) and is divided into 12 columns of equal width (corresponding to a duration of $0,16s$) separated by dotted vertical lines. The columns are numbered from 1 to 12, starting at the leftmost column in each figure. The y-axis has different meanings in different windows: The upper window (window 1) shows the precision (i.e. the maximum deviation between any two clocks), whereas the lower window (window 2) shows the MMCF values of the different nodes.

6.2. Case I: Consistent detectable failure of the PRM

All nodes execute the fault-tolerant clock state and rate correction algorithm described in Section 5. After a few rate corrections at the time-keeping nodes and the SRM, cluster precision improves from 11 microticks to 4 microticks. Figure 3 shows the experimental results for Case I.

After about $0,4s$ simulation time (column 3), the PRM (node 2) suffers from a drift rate failure such that its local clock speeds up and changes its rate by $10^{-3} s/s$ within a duration of $0,08s$. The time-keeping nodes and the SRM modify their MMCF values to maintain agreement between their clock rates and the clock rate of the PRM (column 3, window 2). However, the rate calibration mechanism at the time-keeping nodes cannot prevent the PRM from drifting apart too far from cluster time: at $0,42s$ simulation time (column 3), the PRM detects a clock synchronization error and fails [2] (note the deterioration of precision in window 1). The time-keeping nodes and the SRM note that the PRM is out of service (all nodes fail to receive a message from the PRM in the following round) and agree on node 3 to become the new PRM according

[2] A node encounters a clock synchronization error if its current clock state correction term is bigger than half the granularity of the global timebase (which is 10 microticks in our configuration) [21].

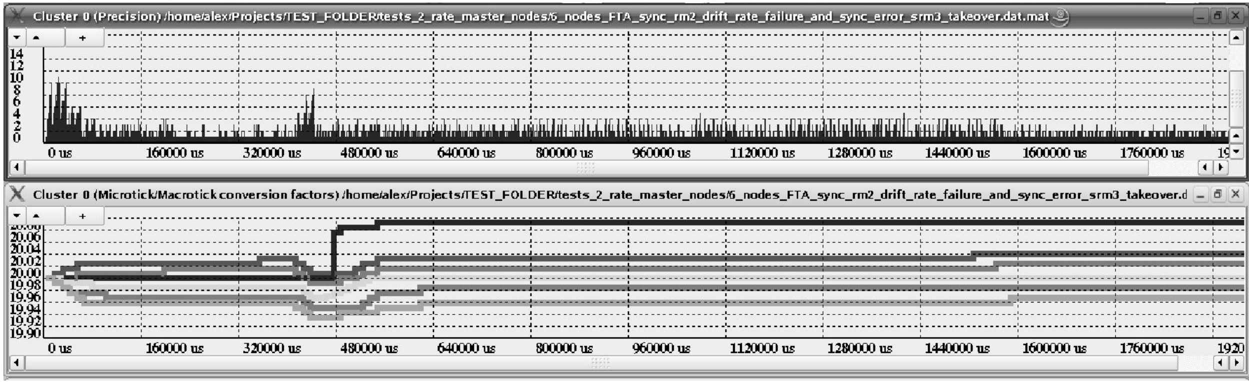


Figure 3. Consistent detectable failure of the PRM

to the rate-master voting algorithm (Section 5). The new PRM stepwise returns to its original rate by periodically adding a constant value of 0.005 to its MMCF fractional part (MMCF values rising in window 2, column 4) until its nominal MMCF value is attained at $0,64s$ simulation time (column 5). The time-keeping nodes also adjust their MMCF values according to this rate change of the new PRM. The former PRM reintegrates at $0,48s$ simulation time, starts with an initial MMCF value of $20,06$ (determined during a measurement phase before reintegration) and follows the clock rate of the PRM (node 3). At about $0,64s$ simulation time the clock rates of the time-keeping nodes and the former PRM (node 2) are calibrated to the clock rate of the new PRM (node 3) and do not change significantly till the end of simulation (window 2). Eventually, the cluster reaches and maintains a stable precision of 4 microticks.

In this experiment, the PRM detects a synchronization error and performs restart.

6.3. Case II: PRM clock drift rate failure

All nodes execute the fault-tolerant clock state and rate correction algorithm described in Section 5. After a few rate corrections at the time-keeping nodes and the SRM, cluster precision improves from 11 microticks to 4 microticks. Figure 4 shows the experimental results for Case II.

After about $0,4s$ simulation time (column 3), the PRM (node 2) suffers from a drift rate failure such that its local clock speeds up and changes its rate by $10^{-4}\, s/s$ within a duration of $0,08s$. The time-keeping nodes modify their MMCF values to maintain agreement between their clock rates and the clock rate of the PRM (column 3, window 2). Compared to the last experiment, the rate change of the PRM is one order of magnitude smaller, small enough to prevent the PRM to encounter a clock synchronization error. However, the PRM remains synchronized and slowly becomes faster and faster, drawing the clock rates of the time-keeping nodes towards its own clock rate (window 2, columns 4-7). As the clock rate change of the PRM is rather smooth, there is no significant deterioration in precision during this phase (window 1, columns 4-7). At about $0,96s$ simulation time, a majority of time-keeping nodes encounters a violation of their MMCF boundaries and agree on the SRM (node 3) to become the new PRM according to the rate-master voting algorithm (Section 5). The former PRM (node 2) detects that it is no longer trusted by a majority of nodes and fails (column 7). The new PRM stepwise returns to its original rate by periodically adding a constant value of 0.005 to its MMCF fractional part (MMCF values rising in window 2, column 7) until its nominal MMCF value is attained at about $1,6s$ simulation time (column 10). The time-keeping nodes also adjust their MMCF values according to this rate change of the new PRM. The former PRM reintegrates at $1,04s$ simulation time (column 7), starts with an initial MMCF value of $20,02$ (determined during a measurement phase before reintegration) and follows the clock rate of the PRM (node 3). At $1,6s$ simulation time the clock rates of the time-keeping nodes are calibrated to the clock rate of the new PRM (node 3) and do not change significantly till the end of simulation (window 2). Eventually, the cluster reaches and maintains a stable precision of 4 microticks.

In this experiment, the decision to stop operation was met by the majority of nodes based on the rate-master membership and not by the PRM itself.

7. Tolerating Arbitrary Failures - Hardware Experiments

The rate correction algorithm is evaluated using the Time-Triggered Architecture as a case study [11]. We have performed experiments with clusters with 6 TTA nodes. In the following, we present two representative scenarios of rate-master failures:

Case I: A fault causes a consistently detectable failure of a rate-master node.

Case II: The drift rate of a rate-master node slowly changes.

7.1. Case I

The target setup consists of a cluster of 6 nodes, using bus interconnection topology. Nodes are numbered

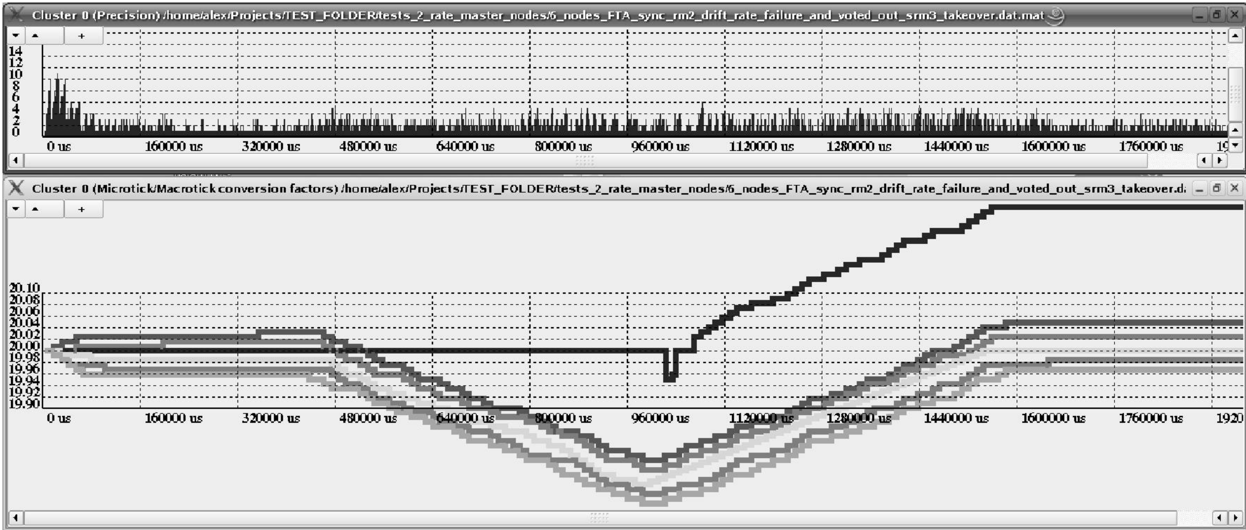


Figure 4. Clock drift rate failure of the PRM

from 0 to 5. One TDMA round consists of 6 slots and has a length of 15 milliseconds. The clock state synchronization interval is 2×TDMA = 30 milliseconds. Because the clock drift rates of the nodes used in the hardware experiments are in the range of 10^{-5} s/s, we have chosen a longer synchronization interval of two TDMA rounds to be able to observe a more significant deviation of the slowest and the fastest clock between two state synchronization instants (i.e. more than 10 microticks) to be able to illustrate the performance of the rate correction algorithm. The duration of the nominal microtick (μt) is 50 nanoseconds, and the initial value of the MMCF is 20, therefore the nominal duration of one macrotick (MT) is $1\mu s$. $Node$ 3 is chosen as the primary rate-master node, and *node* 0 is chosen as the secondary rate-master node. The behavior of the presented algorithm is investigated in the presence of faults. We have performed several fault injection experiments, in which the rate-master node fails in a fail-silent manner (consistently detected by all nodes). One of the experiments is shown in Figure 5.

In this experiment the clock-rates of the nodes are corrected based on the PRM (*node* 3). The horizontal axis in Figure 5 presents the elapsed time in terms of TDMA slots (2.5 ms). The upper part of Figure 5 presents the precision and the lower part of Figure 5 shows the difference of the calculated MMCF values from the nominal value of nodes 0,1 and 2.

It can be seen in Figure 5 that during the first 30 slots (i.e. before rate correction is performed) the precision of the cluster is 16 μt. After the start of clock rate correction the precision of the cluster improves to 6 μt.

The PRM node fails in a fail-silent manner in TDMA slot 500 (Figure 5). As the time-keeping nodes and the S-RM detect the failure of the rate-master, they select *node* 0 as PRM. As the new PRM has changed its rate according to the previous PRM (*node* 3, before becoming a PRM), it starts to stepwise change its rate until its nominal rate has been reached. Other nodes adjust their clock rates to the rate of the new PRM (*node* 0). $Node$ 3 reintegrates in TDMA slot 550. No changes in cluster precision are observed during the reintegration of the rate-master node (*node* 3).

It can be seen that the MMCF of *node* 0 changes back to its nominal rate. The difference of the MMCF to the nominal MMCF value is 1, because the chosen step size for the MMCF change is 2.

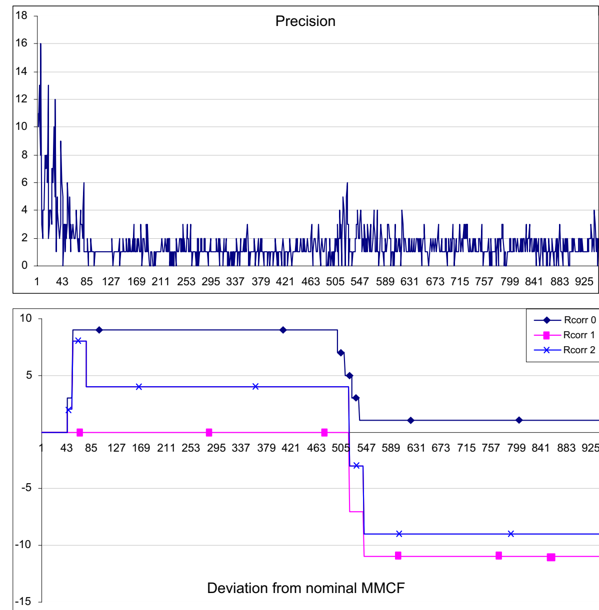


Figure 5. Fail-silent failure of PRM

7.2. Case II

The setup is the same as for *case* I. We injected faults into the PRM to cause drift rate changes of the PRM. Because of the injected faults the PRM (*node* 3) starts to change its rate slowly starting from TDMA slot 450 (Figure 6). The clock rates of the time-keeping nodes

and the SRM also change because they try to follow the rate changes of the PRM (*node* 3). The time-keeping nodes and the SRM node follow the PRM until their rate changes have reached the allowed limit. As soon as this limit is reached, the nodes vote for the secondary rate-master (*node* 0) to become the new PRM, but continue to adjust their rates to the current PRM (*node* 3) until *node* 0 gets a majority of votes. When the majority of votes is for the secondary rate-master (*node* 0), all nodes classify the secondary rate-master as new PRM and the previous PRM as the new SRM (handover). The new PRM (*node* 0) now slowly changes its current MMCF value back to the nominal MMCF value.

In our experiments we have considered the worst case with respect to the drift rates of the nodes: *node* 0 node is the fastest node in the cluster, whereas *node* 3 is the slowest node in the cluster. Usually both rate-master nodes have similar clock drift rates, as they are nodes with high-quality oscillators.

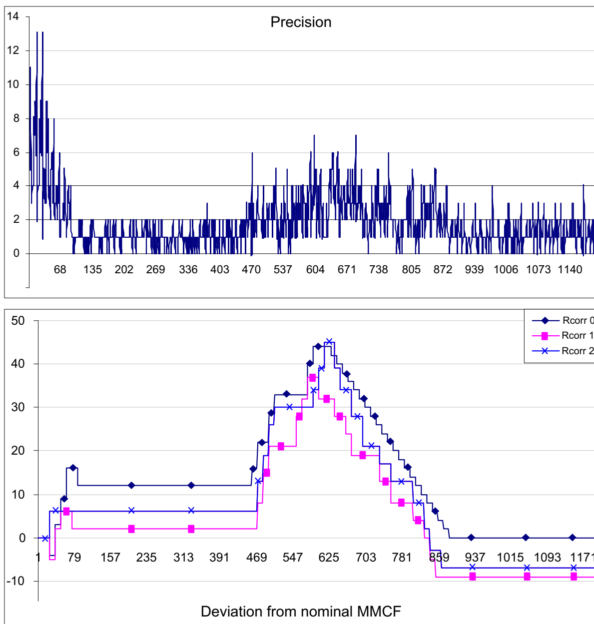


Figure 6. Clock drift rate failure of PRM

7.3. Discussion

Our system model reduces the arbitrary failure modes of the rate-master nodes to a restricted set of failure modes. Therefore, within our system model we have to consider only this reduced set of failure modes in order to be able to tolerate arbitrary failure modes of the rate-master nodes. By means of experiments we have shown that the proposed algorithm is capable of handling arbitrary rate-master node failures that can occur within our system model.

8. Related Work

Fetzer and Cristian in [5] present the CS algorithm for the integration of internal and external clock synchronization, where a set of reference time servers provide access to external reference time for non-reference time servers which are running virtual clocks.

A synchronization strategy for external synchronization of multi-cluster real-time systems based on clock state correction is presented in [12]. Experimental results of the implementation of a similar mechanism using a GPS receiver as a reference time server are presented in [4].

Verissimo et al. propose a synchronization mechanism designed for large-scale systems that are divided into sub-networks [22]. Each sub-network has at least one GPS node that has access to GPS time and that provides reference time to the other nodes within its network.

The network time protocol NTP [13] is a synchronization strategy for large, heterogenous systems like the internet. NTP is organized hierarchically, i.e. one or more primary servers synchronize directly to external reference time and secondary servers synchronize either to these primary servers or to other secondary servers.

Schmid in [17] presents a clock validation technique for establishing a highly accurate global time in fault-tolerant, large-scale distributed real-time systems that provides a precise system time that also relates to an external time standard with high accuracy.

The approach described in [19] presents an interval-based fault-tolerant algorithm for synchronizing both state and rate of clocks in a distributed system. The algorithm presented in [19] requires the exchange of synchronization messages among the nodes in the cluster.

The FlexRay protocol [6] uses a combination of clock state and clock rate correction to improve the precision of the global time base in distributed automotive applications. All nodes adjust their clock drift rate according to their deviation from the internally synchronized global time.

Schmuck and Cristian in [18] introduce the notion of clock amortization where the clock state correction term is not instantaneously applied to the local clocks, but spread over parts of or over the whole synchronization interval.

In our approach we are considering a system that uses a synchronous model of communication (using a TDMA scheme with *a-priori* knowledge of message send times). For the clock rate correction no additional messages or overhead in the message is introduced. Our solution is applicable for all synchronous systems that exchange messages using a TDMA scheme in broadcast mode (e.g TTP/C, FlexRay [6], TT Ethernet [9]).

A detailed discussion that compares our approach to the related work presented in this section is given in [10].

9. Conclusion

In this paper we presented an approach for maintaining a central clock rate correction algorithm in the case of ar-

bitrary failures of a rate-master clock. The presented system model restricts the manifestation of arbitrary failures such that a replication of the rate-master clock together with a rate-master membership algorithm is sufficient to tolerate arbitrary node failures.

As the implementation, debugging and monitoring of the special properties of the presented mechanisms in hardware is a very time-consuming task, we have first investigated the algorithm using a simulation model. After the algorithm has been developed and validated in the simulation environment, it was implemented in hardware to validate its applicability in a real environment with real oscillators.

By means of simulation and hardware experiments we have shown that using this approach it is possible to establish and maintain a global time base of high precision in the presence of arbitrary node failures in the presented system model.

Acknowledgments This work has been supported by the FWF project P16638 and the European IST project DECOS under project No. IST-511764.

References

[1] A. Ademaj. Slightly-Off-specification Failures in the Time-Triggered Architecture. In *Seventh Annual IEEE Workshop on High-Level Design Validation and Test (HLDVT02)*, pages 7–12, Cannes, France, Oct. 2002.

[2] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of Fault Handling of the Time-Triggered Architecture with Bus and Star Topology. In *IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, pages 123–132, San Francisco, Cal, USA, June 2003.

[3] G. Bauer and M. Paulitsch. An Investigation of Membership and Clique Avoidance in TTP/C. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 118–124, Nürnberg, Germany, Oct. 2000.

[4] G. Bauer and M. Paulitsch. External Clock Synchronization in the TTA. Research Report 3/2000, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2000.

[5] C. Fetzer and F. Christian. Integrating External and Internal Clock Synchronization. *Real-Time Systems*, 12(2):123–171, 1997.

[6] FlexRay-Group. FlexRay Communications System Protocol Specification Version 2.1. Technical report, FlexRay Consortium, 2005. Available at: http://www.flexray.com.

[7] A. Hanzlik. SIDERA - a Simulation Model for Time-Triggered Distributed Real-Time Systems. *International Review on Computers and Software (IRECOS)*, 1(3):181–193, Nov. 2006. Praise Worthy Prize.

[8] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9894-7.

[9] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet (TTE) Design. In *8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, Seattle, Washington, May 2005.

[10] H. Kopetz, A. Ademaj, and A. Hanzlik. Integration of Internal and External Clock Synchronization by the Combination of Clock-State and Clock-Rate Correction in Fault-Tolerant Distributed Systems. In *The 25th IEEE International Real-Time Systems Symposium, Lisbon, Portugal, Dec. 2004*, pages 415–425, Dec. 2004.

[11] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):112–126, Jan. 2003.

[12] H. Kopetz, A. Krüger, D. Millinger, and A. Schedl. A Synchronization Strategy for a Time-Triggered Multicluster Real-Time System. *14th IEEE Symposium on Reliable Distributed Systems*, Apr. 1995.

[13] D. L. Mills. Internet Time Synchronization: The Network Time Protocol. In *Zhonghua Yang and T. Anthony Marsland (Eds.), Global States and Time in Distributed Systems, IEEE Computer Society Press*. 1994.

[14] B. Pauli, A. Meyna, and P. Heitmann. Reliability of Electronic Components and Control Units in Motor Vehicle Applications. *Electronic Systems for Vehicles*, pages 1009–1024, 1998.

[15] H. Pfeifer. Formal Verification of the TTP Group Membership Algorithm. In *Formal Techniques for Distributed System Development, FORTE/PSTV 2000, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX), October 10-13, 2000, Pisa, Italy*, pages 3–18, 2000.

[16] H. Pfeifer, D. Schwier, and F. W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In *Dependable Computing for Critical Applications (DCCA-7)*, pages 207–226, San Jose, USA, Jan. 1999.

[17] U. Schmid. Synchronized Universal Time Coordinated for Distributed Real-Time Systems. *Control Engineering Practice*, 6(3):877–884, 1995.

[18] F. Schmuck and F. Cristian. Continuous Clock Amortization need not affect the Precision of a Clock Synchronization Algorithm. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 504–511, Quebec City, Quebec, Canada, 1990.

[19] K. Schossmaier and B. Weiss. An Algorithm for Fault-Tolerant Clock State and Rate Synchronization. In *Symposium on Reliable Distributed Systems, Lausanne, Switzerland*, pages 36–47, Oct. 1999.

[20] C. Temple. Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System. In *28th International Symposium on Fault-Tolerant Computing*, volume FTCS-28, pages 218–227, Munich, Germany, June 1998. IEEE Press.

[21] TTTech. Time-Triggered Protocol, High Level Specification Document. Vienna, Austia, D-032-S-10-28 Available at http://www.tttech.com, 2002.

[22] P. Verissimo, L. Rodrigues, and A. Casimiro. Cesium-Spray: a Precise and Accurate Global Time Service for Large-scale Systems. *In Special Issue on the Challenge of Global Time in Large-Scale Distributed Real-Time Systems. Journal of Real-Time Systems*, 12(3):243–294, 1997.

Exploiting Slack for Scheduling Dependent, Distributable Real-Time Threads in Unreliable Networks

Kai Han⋆, Binoy Ravindran⋆, and E. D. Jensen‡

⋆ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{khan05,binoy}@vt.edu

‡The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

Abstract

We consider scheduling distributable real-time threads with dependencies (e.g., due to synchronization) in unreliable networks, in the presence of node/link failures, message losses, and dynamic node joins and departures. We present a distributed real-time scheduling algorithm called RTG-DS. The algorithm uses a gossip-style protocol for discovering eligible nodes, node/link failures, and message losses. In scheduling local thread sections, it exploits thread slacks to optimize the time available for gossiping. We prove that RTG-DS probabilistically bounds distributed blocking times and distributed deadlock detection and notification times. Thereby, it probabilistically satisfies end-to-end thread time constraints. We also prove that RTG-DS probabilistically bounds failure-exception notification times for failed threads (so that their partially executed sections can be aborted). Our simulation results validate RTG-DS's effectiveness.

1. Introduction

Many distributed systems are most naturally structured as a multiplicity of causally-dependent, flows of execution within and among objects, asynchronously and concurrently. The causal flow of execution can be a *sequence* such as one that is caused by a series of nested, remote method invocations. It can also be caused by a series of chained, publication and subscription events, caused due to topical data dependencies—e.g., publication of topic A depends on subscription of topic B; B's publication, in turn, depends on subscription of topic C, and so on. Since partial failures are the common case rather than the exception in some distributed systems, those applications desire the sequential execution flow abstraction to exhibit application-specific, end-to-end integrity properties. Real-time distributed applications also require (application-specific) end-to-end timeliness properties for the abstraction, in addition to end-to-end integrity.

An abstraction for programming causal, multi-node sequential behaviors and for enforcing end-to-end properties on them is *distributable threads* [1, 9]. They first appeared in the Alpha OS [9], and now constitute the first-class programming and scheduling abstraction for multi-node sequential behaviors in Sun's emerging Distributed Real-Time Specification for Java [1]. In the rest of the paper, we will refer to distributable threads as *threads*, unless qualified.

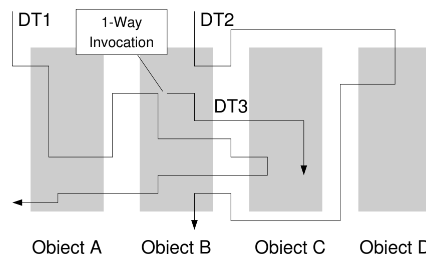


Figure 1. Distributable Threads

A thread is a single logically distinct (i.e., having a globally unique ID) locus of control flow movement that extends and retracts through local and (potentially) remote objects. A thread carries its execution context as it transits node boundaries, including its scheduling parameters (e.g., time constraints, execution time), identity, and security credentials. The propagated thread context is intended to be used by node schedulers for resolving all node-local resource contention among threads such as that for node's physical and logical resources (e.g., CPU, I/O, locks), according to a discipline that provides acceptably optimal system-wide timeliness. Thus, threads constitute the abstraction for concurrency and scheduling. Figure 1 shows the execution of three threads [10].

We consider threads as the programming and scheduling abstraction in *unreliable networks* (e.g., those without a fixed network infrastructure, including mobile, ad hoc and wireless networks), in the presence of application- and network-

induced uncertainties. The uncertainties include resource overloads (due to context-dependent thread execution times), arbitrary thread arrivals, arbitrary node failures, and transient and permanent link failures (causing varying packet drop rate behaviors). Despite the uncertainties, such applications desire strong assurances on end-to-end thread timeliness behavior. Probabilistic timing assurances are often appropriate.

When threads mutually-exclusively share non-CPU resources (e.g., disks, NICs) at a node using lock-based synchronizers, distributed dependencies can arise, causing distributed blockings and deadlocks. For example, a thread A may lock a resource on a node and may make a remote invocation, carrying the lock with it. Thread B may later request the same lock and will be blocked, until A unwinds back from its remote invocation and releases the lock. Unbounded blocking time can degrade system-wide timeliness optimality—e.g., B may have a greater urgency than A. Further, distributed deadlocks can occur when threads A and B block on each other for remotely held locks. Unbounded deadlock detection and resolution times can also degrade timeliness optimality.

When a thread encounters a node/link failure, partially executed thread sections may be blocked on nodes that are upstream and downstream of the failure point, waiting for the thread to unwind back from invocations that are further downstream to them. Such sections must be notified of the thread failure, so that they can respond with application-specific exception handling actions—e.g., releasing handlers for execution that abort the sections, after releasing and rolling-back resources held by them to safe states (under a termination model). Untimely failure notifications can degrade timeliness optimality—e.g., threads unaffected by a partial failure may become indefinitely blocked by sections of failed threads.

In this paper, we present an algorithm called *Real-Time Gossip algorithm for Dependent threads with Slack scheduling* (or RTG-DS) that provides assurances on thread time constraint satisfactions in the presence of distributed dependencies. We prove that thread blocking times and deadlock detection and notification times are probabilistically bounded under RTG-DS. Consequently, we prove that thread time constraint satisfactions' are probabilistically bounded. We also prove that RTG-DS probabilistically bounds failure-exception notification times for partially executed sections of failed threads. Our simulation studies verify the algorithm's effectiveness.

End-to-end real-time scheduling has been previously studied (e.g., [2, 15]), but these are limited to fixed infrastructure networks. Real-time assurances in unreliable networks have been studied (e.g., [6, 8]), but these exclude dependencies, which is precisely what RTG-DS targets.

Our work builds upon our prior work in [4] that presents the RTG-D algorithm. While RTG-DS uses a slack-based thread scheduling approach, RTG-D uses the Dependent Activity Scheduling Algorithm (DASA) in [3] for thread scheduling. We compare RTG-DS with RTG-D in this paper and illustrate RTG-DS's superiority. Further, RTG-D does not consider deadlock detection and notification, and failure-exception notification, while RTG-DS provides probabilistic assurances on such notification times. Thus, the paper's contribution is the RTG-DS that provides probabilistic end-to-end timing assurances (time constraint satisfactions and failure recovery times) in the presence of distributed dependencies.

The rest of the paper is organized as follows: In Section 2, we discuss the models of RTG-DS and state the algorithm objectives. Section 3 presents RTG-DS. We analyze RTG-DS in Section 4. In Section 5, we report our simulation studies. We conclude the paper and identify future work in Section 6.

2. Models and Algorithm Objectives

2.1. Task Model: Thread Abstraction

Distributable threads execute in local and remote objects by location-independent invocations and returns. A thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments.

A thread's initial segment is called its *root* and its most recent segment is called its *head*. The head of a thread is the only segment that is active. A thread can also be viewed as being composed of a sequence of *sections*, where a section is a maximal length sequence of contiguous thread segments on a node. A section's first segment results from an invocation from another node, and its last segment performs a remote invocation. More details on threads can be found in [1, 9, 10].

Execution time estimates of the sections of a thread are assumed to be known. The time estimate includes that of the section's normal code and its exception handler code, and can be violated at run-time (e.g., due to context dependence).

Each object transited by threads is uniquely hosted by a node. Threads may be created at arbitrary times at a node. Upon creation, the number of objects (and the object IDs) on which they will make subsequent invocations are known.

The identifier of the nodes hosting the objects, however, are unknown at thread creation time, as nodes may dynamically fail, or join, or leave the system. Thus, eligible nodes have to be dynamically discovered as thread execution progresses.

The sequence of remote invocations and returns made by a thread can be estimated by analyzing the thread code. The total number of sections of a thread is thus assumed to be known.

The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$.

2.2. Timeliness Model

Each thread's time constraint is specified using a time/utility function (or TUF) [5]. A TUF specifies the utility of completing a thread as a function of that thread's completion time. Figure 2 shows three example downward "step" shaped TUFs. A thread's TUF decouples its *importance* and *urgency*—i.e., urgency is measured as a deadline on the X-axis, and importance is denoted by utility on the Y-axis. This decoupling is significant, as a thread's urgency is sometimes orthogonal to its relative importance—e.g., the most urgent thread is the least important, and vice versa.

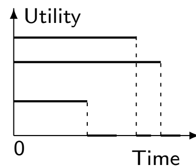


Figure 2. Step TUFs

A thread T_i's TUF is denoted as $U_i(t)$. Classical deadline is unit-valued—i.e., $U_i(t) = \{0, 1\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{0, \{n\}\}$. We focus on downward step TUFs, and denote the maximum, constant utility of a TUF $U_i()$, simply as U_i. Each TUF has an initial time I_i, which is the earliest time for which the TUF is defined, and a termination time X_i, which, for a downward step TUF, is its discontinuity point. Further, we assume that $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.

When thread time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued thread utility—e.g., maximizing the sum of the threads' attained utilities. Such criteria are called *utility accrual* (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. The criteria may also include other factors such as resource dependencies. Several UA algorithms such as DASA are presented in the literature. We derive RTG-DS's local thread section scheduling algorithm from DASA, and compare it with DASA.

2.3. Exceptions and Abortion Model

If a thread has not completed by its termination time, a failure-exception is raised, and exception handlers are immediately released and executed for aborting all partially executed thread sections. The handlers are assumed to perform the necessary compensations to avoid inconsistencies (e.g., rolling back resources held by the sections to safe states) and other actions that are required for the safety and stability of the external state.

Note that once a thread violates its termination time, the scheduler at that node will immediately raise the failure exception. At all other (upstream) nodes, a notification for the exception must be delivered. RTG-DS delivers those notifications.

We consider a similar abortion model for thread failures, for resolving deadlocks, and for resolving thread blocks when a blocking thread is aborted to obtain greater utility (similar to transactional abortions [14]). The scheduler at the node where these situations are detected will raise the failure exception. At all other (upstream) nodes, where the thread has partially executed, RTG-DS delivers the failure exception.

2.4. Resource Model

Thread sections can access non-CPU resources (e.g., disks, NICs) located at their nodes during their execution, which are serially reusable. Similar to fixed-priority resource access protocols [13] and that for TUF algorithms (e.g., [3]), we consider a single-unit resource model. Resources can be shared under mutual exclusion constraints. A thread may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped or disjoint. Threads explicitly release all granted resources before the end of their executions.

All resource request/release pairs are assumed to be confined within nodes. Thus, a thread cannot lock a resource on one node and release it on another node. Note that once a thread locks a resource on a node, it can make remote invocations (carrying the lock with it). Since request/release pairs are confined within nodes, the lock is released after the thread's head returns back to the node where the lock was acquired.

Threads are assumed to access resources arbitrarily—i.e., which resources will be needed by which threads, and in what order is not a-priori known. Consequently, we consider a deadlock detection and resolution strategy. A deadlock is resolved by aborting a thread involved in the deadlock, by executing the thread's handler.

2.5. System Model

The system consists of a set of processing components, generically referred to as *nodes*, denoted $N = \{n_1, n_2, n_3, \ldots\}$, communicating through bidirectional wireless links. A basic unicast routing protocol such as DSR is assumed for packet

transmission. MAC-layer packet scheduling is assumed to be done by a CSMA/CA-like protocol (e.g., IEEE 802.11). Node clocks are synchronized using an algorithm such as [12]. Nodes may dynamically join or leave the network. We assume that the network communication delay follows some non-negative probability distribution—e.g., the Gamma distribution. Nodes may fail by crashing, links may fail transiently or permanently, and messages may be lost, all arbitrarily.

2.6. Objectives

Our goal is to design an algorithm that can schedule threads with probabilistic termination-time satisfactions in the presence of (distributed) blockings and deadlocks. We also desire to maximize the total thread accrued utility. Moreover, the time needed to notify partially executed sections of a failed thread (so that handlers for aborting thread sections can be released) must also be probabilistically bounded.

Note that maximizing the total utility subsumes meeting all termination times as a special case. When all termination times are met (during underloads), the total accrued utility is the optimum possible. During overloads, the goal is to maximize the total utility as much as possible, thereby completing as many important threads as possible, irrespective of their urgency.

3. The RTG-DS Algorithm

We first overview RTG-DS. When a thread arrives at a node, RTG-DS decomposes the thread's end-to-end TUF into a set of local TUFs, one for each of the sections of the thread. The decomposition is done using the thread's scheduling parameters including its end-to-end TUF, number of sections, section execution time estimates that the thread presents to RTG-DS upon arrival. Local TUFs are used for thread scheduling on nodes.

When a thread completes its execution on a node, RTG-DS must determine the thread's next destination node. In order to be robust against node/link failures, message looses, and node joins/departures, RTG-DS uses a gossip-style protocol (e.g., [7]). The algorithm starts a series of synchronous gossip rounds. During each round, the node randomly selects a set of neighbor nodes and queries whether they can execute the thread's next section (as part of the thread's next invocation or return from its current invocation). The number of gossip rounds, their durations, and the number of neighbor nodes are derived from the local TUF's slack, as they directly affect the communication time incurred by gossip, and thereby affect following sections' available local slack.

When a node receives a gossip message, it checks whether it hosts the requested section, and can complete it satisfying its local TUF (propagated with the gossip message). If so, it replies back to the node where the gossip originated (referred to as the original node). If not, the node starts a series of gossip rounds and sends gossip messages (like the original node).

If the original node receives a reply from a node before the end of its gossip rounds, the thread is allowed to make an invocation on, or return to that node, and thread execution continues. If a reply is not received, the node regards that further thread execution is not possible (due to possible node/link failures or node departures), and releases the section's exception handler for execution. A series of gossip rounds is also immediately started to deliver the failure-exception notification to all upstream sections of the thread, so that handlers may be released on those nodes.

We now discuss the key aspects of RTG-DS.

3.1. Building Local Scheduling Parameters

RTG-DS decomposes a thread's end-to-end TUF based on the execution time estimates of the thread's sections and the TUF termination time. Let a thread T_i arrive at a node n_j at time t. Let T_i's total execution time of all the thread sections (including the local section on n_j) be Er_i, the total remaining slack time be Sr_i, the number of remaining thread sections (including the local section on n_j) be Nr_i, and the execution time of the local section be $Er_{i,j}$. RTG-DS computes a local slack time $LS_{i,j}$ for T_i as $LS_{i,j} = \frac{Sr_i}{Nr_i - 1}$, if $Nr_i > 1$; $LS_{i,j} = Sr_i$, if $0 \leqslant Nr_i \leqslant 1$.

RTG-DS determines the local slack for a thread in a way that allows the remaining thread sections to have a fair chance to complete their execution, given the current knowledge of section execution-time estimates, in the following way. When the execution of T_i's current section is completed at the node n_j, RTG-DS determines the next node for executing the thread's next section, through a set of gossip rounds. The network communication delay incurred by RTG-DS for the gossip rounds must be limited to at most the local slack time $LS_{i,j}$. The algorithm equally divides the total remaining slack time to give the remaining thread sections a fair chance to complete their execution.

The local slack is used to compute a local termination time for the thread section. The local termination time for a thread T_i is given by $LX_{i,j} = t + Er_{i,j} + LS_{i,j}$. The local termination time is used to test for schedule feasibility, while constructing local section schedules (we discuss this in Section 3.3).

3.2. Determining Next Destination Node

Once the execution of a section completes on a node, RTG-DS determines the node for executing the next section of the thread, through a set of gossip rounds during which the node randomly multicasts with other nodes in the network. RTG-DS determines the number of rounds for "gossiping" (i.e., sending messages to randomly selected nodes during a single gossip round) as follows. Let the execution of T_i's local section on node n_j complete at time t_c. T_i's remaining local slack time is given by $LSr_{i,j} = LX_{i,j} - t_c$.

Note that $LSr_{i,j}$ is not always equal to $LS_{i,j}$, due to the interference that the thread section suffers from other sections on the node. Thus, $LSr_{i,j} \leq LS_{i,j}$. With a gossip period Ψ, RTG-DS determines the number of gossip rounds before $LX_{i,j}$ as $round = LSr_{i,j}/\Psi$. RTG-DS also determines the number of messages that must be sent during each gossip round, called *fan out*, for determining the next node.

RTG-DS divides the system node members into: a) *head* nodes that execute thread sections, and b) *intermediate* nodes that propagate received gossip messages to other members.

3.3. Scheduling Local Sections

RTG-DS constructs local section schedules with the goals of (a) maximizing the total attained utility from all local sections, (b) maximizing the number of local sections meeting their local termination times, and (c) increasing the likelihood for threads to meet thread termination times, while respecting dependencies.

The algorithm's scheduling events include section arrivals and departures, and lock and unlock requests. When the algorithm is invoked, it first builds the dependency list of each section by following the chain of resource request and ownership. A section i is dependent upon a section j, if i needs a resource which is currently held by j. Dependencies can be local—i.e., the requested lock is locally held, or distributed—i.e., the requested lock is remotely held.

The algorithm then checks for deadlocks, which can be local or distributed (e.g., two threads are blocked on their respective nodes for locks which are remotely held by the other). Deadlocks are detected by the presence of a cycle in the resource graph (a necessary condition). Deadlocks are resolved by aborting that section in the cycle, which will likely contribute the least utility. That section is aborted by executing its handler, which will perform roll-backs/compensations.

Now, the algorithm examines sections in the order of non-increasing *potential utility densities* (or PUDs). A section's PUD is the total utility accrued by immediately executing it and its dependents divided by the aggregate execution time spent (i.e., the section's "return on investment").

The algorithm inserts each examined section and its dependents into a tentative schedule that is ordered by local slacks, least-slack-first (or LSF). The insertion also respects each section's dependency order. After insertion, the feasibility of the schedule is checked. If infeasible, the inserted section and its dependents are removed. The process is repeated until all sections are examined. Then, RTG-DS selects the least-slack section for execution. If the selected section is remote (because it holds a locally requested lock), the algorithm will speed up it's execution by adding all local dependents' utilities and propagating the aggregate value to it (by gossiping).

We now explain key steps of the algorithm.

3.3.1 Arranging Sections by PUD

The local scheduling algorithm examines sections in non-increasing PUD order to maximize the total accrued utility. Section i's PUD, $PUD_i = \frac{U_i+U(Dep(i))}{c_i+c(Dep(i))}$, where U_i is i's utility, c_i is i's execution time, and $Dep(i)$ is the set of sections on which i is directly or transitively dependent. Note that PUD_i can change over time, since c_i and $Dep(i)$ may change over time.

3.3.2 Determining Schedule Feasibility

RTG-DS determines a node's processor load ρ_R by considering that node's own processor bandwidth, and also by leaving a necessary gossip time interval for each thread section. Let t be the current time, and d_i be the local termination time of section i. ρ_R in time interval $[t, d_i]$ is given by:

$$\rho_{R_i}(t) = \frac{\sum_{d_k \leq d_i} c_k(t) + T_{comm}}{(d_i - t)}, \quad T_{comm} \geq LCD$$

where $c_k(t)$ is section k's remaining execution time with $d_k \leq d_i$, and LCD is the lower bound of network communication delay. Different from computing ρ on a single node, RTG-DS adds an additional communication time interval, T_{comm}, to each $c_k(t)$. If a section is the last one of its parent thread, there is no need to consider gossiping time and $T_{comm} = 0$. Without adding T_{comm}, a section may successfully complete, but may not have enough time to find the next destination node. Thus, not only that section's parent thread will be aborted in the end, but also will waste processor bandwidth, which could otherwise be used for other threads' sections.

Suppose there are n sections on a node, and let d_n be the longest local termination time. Then, the total load in $[t, d_n]$ is computed as: $\rho_R(t) = max \rho_{R_i}(t), \forall i$.

3.3.3 Least-Slack Section First (LSF)

RTG-DS selects local sections with the lesser (local) slack time earlier for execution. This ensures that greater remaining slack time is available for threads to find their next destination nodes.

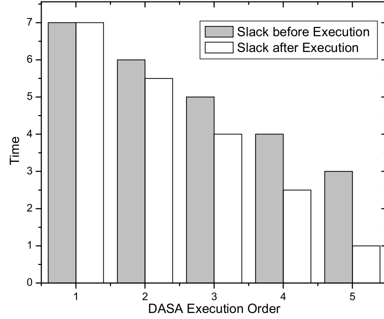


Figure 3. Slack Under DASA

For example, consider five sections with different local slack times. Figure 3 shows slacks of the sections before and after execution under DASA, on a single node. In the worst case, DASA will schedule sections along the decreasing order of slacks, as shown in Figure 3. Assuming that the lower bound of network communication time, LCD, is 0.5 time unit, section 5 has only 1 time unit left to gossip (its original local slack is 3 time units), which makes it more difficult to make a successful invocation on (or return to) another node.

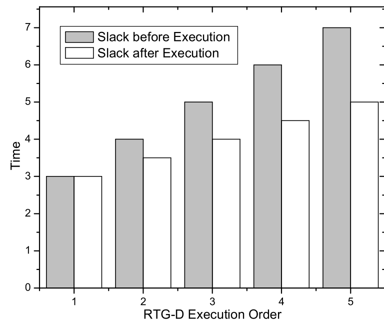


Figure 4. Slack Under RTG-DS

RTG-DS avoids this with the LSF order. In Figure 4, section 5's remaining local slack remains unchanged after execution, while section 1's slack decreases from 7 to 5 time units, which will slightly decrease its gossip time. Note that RTG-DS gains the same total slack time in these five sections as DASA does, but it allocates slack time more evenly, thereby seeks to give each section an equal chance to complete gossiping.

When checking feasibility, it is important to respect dependencies among sections. For example, section j may depend on section i, thus i must be executed before j to respect the dependency. However, under $LS_{ri} > LS_{rj}$, j will be arranged before i. To resolve this conflict without breaking the LSF order, RTG-DS "tightens" i's local slack time to the same as j's.

RTG-DS's local section scheduling algorithm is described in Algorithm 1.

Algorithm 1: RTG-DS's Local Section Scheduling Algorithm

```
1  Create an empty schedule φ;
2  for each section i in the local ready queue do
3      Compute Dep(i), detecting and resolving
       deadlocks if any;
4      Compute PUD_i;
5  Sort sections in ready queue according to PUDs;
6  for each section i in decreasing PUD order do
7      φ̂ = φ; /* get a copy for tentative changes */
8      if i ∉ φ then
9          CurrentLST = LST(i); /* LST(i) returns
           the local slack of i */
10         for each PrevS in Dep(i) do
11             if PrevS ∈ φ then
12                 if LST(PrevS) ≤ CurrentLST
                   then
13                     Continue;
14                 else
15                     LST(PrevS) = CurrentLST;
16                     Remove(PrevS, φ̂, LST); /*
                       Remove PrevS from φ̂ at
                       position LST */
17             Insert(PrevS, φ̂, CurrentLST);
18         if Feasible(φ̂) then
19             Insert(i, φ̂, CurrentLST);
20             if Feasible(φ̂) then
21                 φ = φ̂;
22 Select least-slack section from φ for execution;
```

3.3.4 Utility Propagation

Section i may depend on section j located on the same node or on a different node. For the latter case, RTG-DS propagates i's utility to j in order to speed up j's execution, and thus shorten i's time waiting for blocked resources. The utility is propagated by gossiping to all system members within a limited time interval, as it does in finding the next destination node.

When j's head node receives an utility-propagation message, it has to decide whether to continue executing j, or to immediately abort j and grant the lock to i. This decision is based on Global Utility Density (or GUD), which is defined as the ratio of the owner thread utility to the total remaining thread execution time. Thread PUDs are not used in this case, because this utility comparison involves multiple nodes.

Algorithm 2 describes this decision process. If the decision is to continue j's execution, the node

Algorithm 2: RTG-DS's Utility Propagation Algorithm

```
1  Upon receiving a UP gossip message msg:
2  COPY(gossip, msg) ;
3  if GUD_i > GUD_j then
4      if abt_j < er_j then
5          abort j;
6          gossip.lsr ← msg.lsr − abt_j;
7          /* give resource lock to i */
8      else
9          continue j's execution;
10         /* keep resource lock */
11         gossip.lsr ← msg.lsr;
12 else
13     gossip.lsr ← msg.lsr;
14 gossip.round ← gossip.lsr/Ψ ;
15 gossip.c ← FANOUT(gossip.round);
16 RTG_GOSSIP(gossip);
```

will add i's utility to j's current and previous head nodes, consequently speeding up j's execution (since the scheduler examines sections in the PUD order). If the decision is not to continue j's execution, the node will release j's abort handler, and will start gossiping to 1) release j's abort handler's on all previous head nodes of j and 2) grant lock to i. Note that i's utility is only propagated to j's execution nodes after the node from where i requested the lock, because j's other execution nodes do not contribute to this dependency.

3.3.5 Resolving Distributed Deadlocks

Detecting deadlocks between different nodes require all system members to uniformly identify each thread. Thus, when a thread is created, a global ID (or GID) is created for it. With GIDs, it is easier to determine the thread that must be aborted to resolve a distributed deadlock: If $GUD_i > GUD_j$, then i has a higher utility. Then, j is aborted to grant the lock to i. Otherwise, j keeps the lock and gossips an abortion message back to i. Algorithm 3 describes this procedure.

4. Algorithm Analysis

Let δ be the desired probability for delivering a message to its destination node within the gossip period Ψ. If the communication delay follows a Gamma distribution with a probability density function:

$$f(t) = \frac{(t - LCD)^{\alpha-1} e^{\frac{-(t-LCD)}{\beta}}}{\Gamma(\alpha)\beta^{\alpha}}, \quad t > LCD$$

where $\Gamma(\alpha) = \int_0^{\infty} x^{\alpha-1}e^{-x}dx$, $\alpha > 0$. Then, $\delta = \int_{LCD}^{t_b} f(t)\,dt$, $t > LCD$, where $t_b : D(t_b) = \delta$, and $D(t)$ is the distribution function. Note that LCD

Algorithm 3: RTG-DS's Distributed Deadlock Detection Algorithm

```
1  Upon j receiving i's UP gossip message msg:
2  COPY(gossip, msg) ;
3  if DETECT(msg) = true then
4      /* a distributed deadlock occurs */
5      if GUD_i > GUD_j then
6          abort j;
7          gossip.lsr ← msg.lsr − abt_j;
8          give resource lock to i;
9      else
10         continue j's execution;
11         keep resource lock;
12         gossip.lsr ← msg.lsr;
13     gossip.round ← gossip.lsr/Ψ ;
14     gossip.c ← FANOUT(gossip.round);
15     RTG_GOSSIP(gossip);
```

is the communication delay lower bound and $\Psi > t_b$.

We denote the message loss probability as $0 \leq \sigma < 1$, and the probability for a node to fail during thread execution as $0 \leq \omega < 1$. Let C denote the number of messages that a node sends during each gossip round (i.e., the fan out). We call a node *susceptible* if it has not received any gossip messages so far; otherwise it is called *infected*. The probability that a given susceptible node is infected by a given gossip message is:

$$p = \left(\frac{C}{n-1}\right)(1-\sigma)(1-\omega)\,\delta \qquad (1)$$

Thus, the probability that a given node is not infected by a given gossip message is $q = 1 - p$. Let $I(t)$ denote the number of infected nodes after t gossip rounds, and $U(t)$ denote the number of remaining susceptible nodes after t rounds. Given i infected nodes at the end of the current round, we can compute the probability for j infected nodes at the end of the next round (i.e., $j - i$ susceptible nodes are infected during the next round). The resulting Markov Chain is characterized by the following probability $p_{i,j}$ of transitioning from state i to state j:

$$p_{i,j} = P\left[I(t+1) = j | I(t) = i\right] = \begin{cases} \binom{n-i}{j-i} \left(1-q^i\right)^{j-i} q^{i(n-j)} & j \geqslant i \\ 0 & j < i \end{cases} \qquad (2)$$

The probability that the expected number of j nodes are infected after round $t+1$ is given by:

$$P\left[I(0) = j\right] = \begin{cases} 1 & j = 1 \\ 0 & j > 1 \end{cases} \qquad (3)$$

$$P\left[I(t+1) = j\right] = \sum_{i \leqslant j} P\left[I(t) = i\right] p_{i,j} \qquad (4)$$

Theorem 1. *RTG-DS probabilistically bounds thread time constraint satisfactions'.*

Proof. Let a thread will execute through m head nodes. The mistake probability p_{M_k} that a head node k cannot determine the thread's next destination head node after gossip completes at round t_{max} is given by:

$$p_{M_k} = \{1 - P[I(t_{\max}) = \eta]\} \times \frac{1}{U(t_{\max})}$$
$$= \left\{1 - \sum_{i \leqslant \eta} P[I(t_{\max-1}) = i]\, p_{i[\eta]}\right\} \times \frac{1}{U(t_{\max})} \quad (5)$$

where η is the expected number of infected nodes after $t_{\max}$. This p_{M_k} is achieved when all nodes are not overloaded (consequently, RTG-DS's LSF-order being locally optimal, will feasibly complete all local sections).

Let w_k be the waiting time before section k's execution. w_k includes the section interference time, gossip time, and blocking time (we bound blocking time in Theorem 4). Now, X_k and X_m can be defined as:

$$X_k = \begin{cases} 1 & \text{If } w_k \le LS_{rk} - LCD \\ 0 & \text{Otherwise} \end{cases} \quad (6)$$

$$X_m = \begin{cases} 1 & \text{If } w_k \le LS_{rm} \\ 0 & \text{Otherwise} \end{cases} \quad (7)$$

If $X_k = 1$, the relative section can not only finish its execution, but it can also make a successful invocation. X_m is for the last destination node, so it does not consider the communication delay LCD. Thus, the probability for a distributable thread d to successfully complete its execution P_{S_d}, and that for a thread set D to complete its execution, P_{S_D}, is given by:

$$P_{S_d} = X_m \prod_{k \le m-1} (1 - p_{M_k}) X_k \quad P_{S_D} = \prod_{d \in D} P_{S_d} \quad (8)$$

□

Theorem 2. *The number of rounds needed to infect n nodes, t_n, is given by:*

$$t_n = \log_{C+1} n + \frac{\log n}{C} + o(1) \quad (9)$$

Proof. We skip the proof, due to page constraints. The proof is similar to [11], but we conclude the theorem under the assumption that the fan out C exceeds 1. □

Lemma 3. *A head node will expect its gossip message to be replied in at most $2t_n$ rounds, with a high (computable) probability.*

Proof. Suppose the next destination node N is the last node getting infected by the gossip message from a head node A. Thus, node A will take t_n rounds to gossip to node N. Suppose A is the last node to be infected by N's reply message, and it will take another t_n rounds. Thus, the worst case to determine the next destination node is $2t_n$ rounds. The probability can be computed using Equations 3 and 5. Since the fan out number C can be adjusted, we can get a required probability by modifying C into a proper value. □

Theorem 4. *If a thread section is blocked by another thread section on a different node, then its blocking time under RTG-DS is probabilistically bounded.*

Proof. Suppose section i is blocked by section j whose head is now on a different node. According to Theorem 2, it will take section i at most t_{n_i} time rounds to gossip an utility propagation (UP) message to section j's head node.

After j's head node receives i's UP message, RTG-DS will compare i's GUD with j's. If $GUD_i > GUD_j$, then j must grant the lock to i as soon as possible. According to Algorithm 2, the handler will deal with j's head within $\texttt{min(}abt_j\texttt{,}er_j\texttt{)}$. According to Lemma 3, i's head will expect a reply from j after at most t_{n_i} time rounds. If $t_{n_i} - \texttt{min(}abt_j\texttt{,}er_j\texttt{)} \ge LCD$, then j can reply and grant lock to i at the same time. Thus, i's blocking time bound $b_{i,j} = 2t_{n_i}$. Otherwise, j must first reply to i. Since i's head needs at least LCD gossip time to continue execution, the blocking time is at most $LS_{ri} - LCD$. Thus, if $(LS_{ri} - LCD) - t_{n_i} - \texttt{min(}abt_j\texttt{,}er_j\texttt{)} \ge LCD$, $b_{i,j} = LS_{ri} - LCD$. If not, i has to be aborted because there is not enough time to grant the lock. Under this condition, RTG-DS aborts i, and $b_{i,j} = 2t_{ni}$, since j need not respond any more after the first reply to i. If $GUD_i \le GUD_j$, then j will not grant i the lock until it finishes necessary execution. Thus, $b_{i,j} = LS_{ri} - LCD$.

The probability of the blocking time bound is induced by RTG-DS's gossip process. It can be computed using (3) and (5), and a desired probability can be obtained by adjusting C. □

Theorem 5. *RTG-DS probabilistically bounds deadlock detection and notification times.*

Proof. As shown in Figure 5, there are two possible situations: 1) deadlock happens when section i requires resource R2, or 2) when section j's REQ R1 message arrives at Node 2.

Let $GUD_i > GUD_j$. Under the first condition, i will check the necessary time for deadlock solution, which is denoted as ds_{i2}. Let $LS_{r_i 2}$ be the remaining local slack time of section i on Node 2, $t_{n_i 2}$ be the time rounds needed by i to gossip to

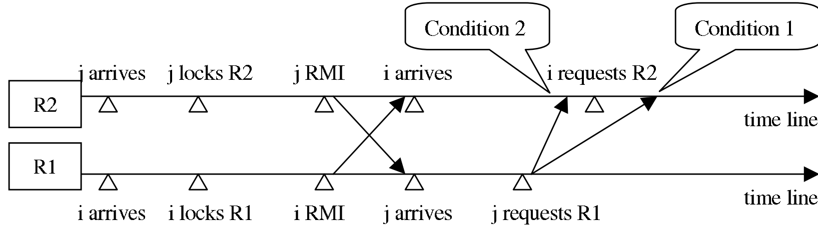


Figure 5. Example Distributed Deadlock

Node 1 in order to finish i on time, and abt_{j1}, abt_{j2} be the needed abortion time of section j on Node 1 and 2, respectively.

Then, $ds_{j2} = abt_{j2}$, if no LIFO-ordered abortion is necessary from node 1 to node 2; otherwise $ds_{j2} = abt_{j1} + abt_{j2} + 2t_{ni2}$. By LIFO-ordered abortion, the last executed section is the first one that is aborted.

Under the second condition, deadlock happens when j's REQ message arrives at Node 2. Now, $ds_{j2} = t_{nj1}$, if $t_{nj1} - abt_{j2} \geq LCD$, or if $t_{nj1} - (abt_{j1} + abt_{j2} + 2t_{ni2}) \geq LCD$. Otherwise, $ds_{j2} = \max(t_{nj1} + abt_{j1} + t_{ni1}, abt_{j2})$.

Thus, if $ds_{j2} \leq LS_{ri2} - LCD$, the scheduler will resume i. Otherwise, it will abort i since i won't have necessary remaining local slack time for gossiping.

The analysis is similar if $GUD_i > GUD_j$. The probabilistic blocking time bound is induced by RTG-DS's gossiping. It can be computed using (3), and a desired probability can be obtained by adjusting fan out C. □

Theorem 6. *RTG-DS probabilistically bounds failure-exception notification times for aborting partially executed sections of failed threads.*

Proof. From Lemma 6 in [4], we obtain the failure-exception notification time f_n as follows: $f_n = 3t_n$, if no LIFO-ordered abortion is necessary from node m to node n. Otherwise, $f_n = 3t_n + \sum_{i=m,\ldots,n} t_{ni}$. □

5. Simulation Studies

We evaluate RTG-DS's effectiveness by comparing it with "RTG-DS/DASA" — i.e., RTG-DS with DASA as the section scheduler — as a baseline. We do so because DASA exhibits very good performance among most UA scheduling algorithms. We use uniform distribution to describe section inter-arrival times, section execution times, and termination times of a set of distributable threads. All threads are generated to make invocations through the same set of nodes in the system. However, the relative arrival order of thread invocations at each node may change due to different section schedules on nodes. Thus, it is possible that a thread may miss its termination time because it arrives at a destination node late.

A fixed number of shared resources is used in the simulation study. The simulations featured four (one on each node) and eight (two on each node) shared resources, respectively. Each section probabilistically determines how many resources it needs. Each time a resource is acquired, a fraction (following uniform distribution) of the section's remaining execution time is assigned to it.

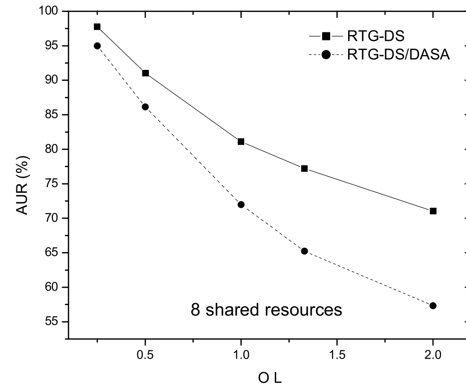


Figure 6. AUR in a 8-Resource-System Under RTG-DS and RTG-DS/DASA

We measure RTG-DS's performance using the metrics of Accrued Utility Ratio (AUR), Termination time Meet Ratio (TMR) and Offered Load (OL) in a 100-node system. AUR is the ratio of the total accrued utility to the maximum possible total utility, TMR is the ratio of the number of threads meeting their termination times to the total thread releases in the system, and OL is the ratio of a section's expected execution time to the expected section inter-arrival time. Thus, when OL < 1.0, a section will most possibly complete its execution before the next section arrives; when OL > 1.0, system will have long-term overloads.

Note that RTG-DS uses the novel techniques that we have presented here including $\rho_R(t)$, GUD and PUD, selecting LSF section, utility propagation, and distributed deadlock resolution. RTG-DS/DASA does not use any of these, but only follows RTG-DS in the gossip-based searching of next destination nodes.

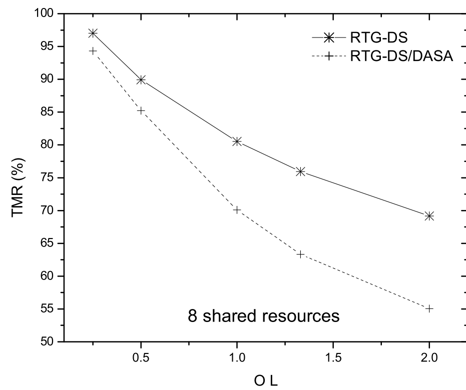


Figure 7. TMR in a 8-Resource-System Under RTG-DS and RTG-DS/DASA

Figures 6 and 7 show the results for the eight-resource system. From the figure, we observe that RTG-DS gives much better performance than RTG-D/DASA. Further, when OL is increased, both algorithms' AUR and TMR decrease. We observe consistent results for the four-resource case, but omit them here for brevity.

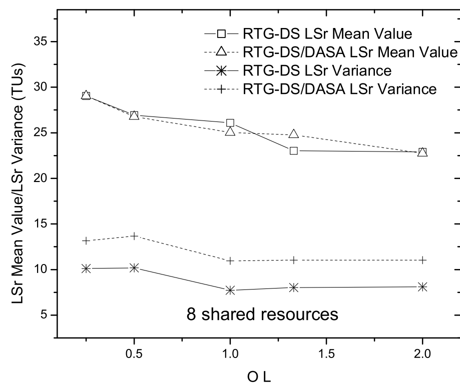


Figure 8. Remaining Local Slack Time Under RTG-DS and RTG-DS/DASA (Mean, Variance)

In Figure 8, as discussed in Section 3.3.3, we observe that under any OL, RTG-DS has a smaller variance of remaining local slack time than RTG-DS/DASA, because it first executes the least-slack section instead of the earliest local termination time section. By this way, though sections' mean value of remaining local slack time after execution is almost the same, RTG-DS gives sections with less local slack time more chances to finish their gossip process, and thus more chances to find their next destination nodes.

6. Conclusions and Future Work

We presented a gossip-based algorithm called RTG-DS, for scheduling distributable threads under dependencies in unreliable networks. We proved that RTG-DS probabilistically bounds thread blocking times and deadlock detection and notification times, thereby probabilistically bounding thread time constraint satisfactions'. We also showed that the algorithm probabilistically bounds failure-exception notification times for aborting failed threads.

Example directions for extending our work include allowing node anonymity, unknown number of thread sections, and non-step TUFs.

References

[1] J. Anderson and E. D. Jensen. The distributed real-time specification for java: Status report. In *JTRES*, 2006.

[2] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed Systems.* PhD thesis, UIUC, 1994.

[3] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, CMU, 1990. CMU-CS-90-155.

[4] K. Han et al. Probabilistic, real-time scheduling of distributable threads under dependencies in mobile, ad hoc networks. In *IEEE WCNC*, 2007.

[5] E. D. Jensen et al. A time-driven scheduling model for real-time systems. In *RTSS*, pages 112–122, 1985.

[6] E. D. Jensen and B. Ravindran. Guest editor's introduction to special section on asynchronous real-time distributed systems. *IEEE Transactions on Computers*, 51(8):881–882, August 2002.

[7] H. Li et al. Bar gossip. In *OSDI*, November 2006.

[8] B. S. Manoj et al. Real-time traffic support for ad hoc wireless networks. In *IEEE ICON*, pages 335 – 340, 2002.

[9] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel.* Academic Press, 1987.

[10] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, OMG, September 2001. Final Adopted Specification.

[11] B. Pittel. On spreading a rumor. *SIAM J. Appl. Math.*, 47(1), 1987.

[12] K. Romer. Time synchronization in ad hoc networks. In *MobiHoc*, pages 173–182, 2001.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[14] N. R. Soparkar, H. F. Korth, and A. Silberschatz. *Time-Constrained Transaction Management.* Kluwer Academic Publishers, 1996.

[15] J. Sun. *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems.* PhD thesis, UIUC, 1997.

Institut National Polytechnique de Lorraine
Impressions et Reliures :
INPL – Atelier de reprographie
2, avenue de la forêt de la Haye
B.P. 3. – F-54501 Vandoeuvre Cedex
Tel : 03.83.59.59.26 ou 03.83.59.59.27
Scientific editors : Isabelle PUAUT (IRISA),
Nicolas NAVET (LORIA), Françoise SIMONOT-LION (LORIA)
ISBN : 2-905267-53-4