International Conference

REAL-TIME AND NETWORK SYSTEMS

RTNS 2009

ECE, Paris, France
October 26-27, 2009

General Chair: Laurent George
Program Comittee Chairs: Maryline Chetto and Mikael Sjodin

Proceedings: HAL-INRIA system

Sponsors: IEEE-France, ECE, INRIA, OpticsValley, Astech

IEEE France Section · Opticsvalley · INRIA · ASTech Paris Region · ECE Paris Graduate School of Engineering

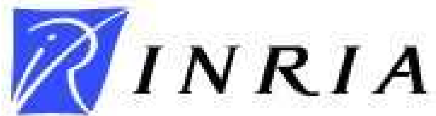
17th International Conference on Real-Time and Network Systems

PROCEEDINGS

October 26-27, 2009
ECE, Graduate school of Engineering
Paris, France

Sponsors

The 17th international conference on Real-Time Network and Systems is sponsored by:



Proceedings

RTNS 2009

Message form the General Chair **p5**
Message for the Co-Program Chairs **p6**
Conference Organization **p7**
Program Committee **p8**
List of Reviewers **p9**
Keynote Speaker **p10**

Technical Program

Uniprocessor Scheduling

François Dorin, Pascal Richard, Michaël Richard and Joël Goossens **p13**
Uniprocessor Schedulability and Sensitivity Analysis of Multiple Criticality Tasks with Fixed-Priorities

Robert Davis, Thomas Rothvoß, Sanjoy Baruah and Alan Burns **p23**
Quantifying the Sub-optimality of Uniprocessor Fixed Priority Pre-emptive Scheduling for Sporadic Tasksets with Arbitrary Deadlines

Timing Analysis

Michael Zolda, Sven Bünte and Raimund Kirner **p35**
Towards Adaptable Control Flow Segmentation for Measurement-Based Execution Time Analysis

Damien Hardy and Isabelle Puaut **p45**
Estimation of cache-related migration delays for multi-core processors with shared instruction caches

Haluk Ozaktas, Karine Heydemann, Christine Rochange and Hugues Cassé **p55**
Impact of Code Compression on Estimated Worst-Case Execution Times

Networks

David Espes and Zoubir Mammeri **p67**
QoS-aware Routing for Real-Time and Multimedia Applications in Mobile Ad Hoc Networks

Zheng Shi and Alan Burns **p75**
Improvement of Schedulability Analysis with a Priority Share Policy in On-Chip Networks

Saoucene Mahfoudh and Pascale Minet **p85**
Node activity scheduling in wireless sensor networks

Resource Management

Attila Zabos, Robert I. Davis, Alan Burns and Michael Gonzalez Harbour **p97**
Spare Capacity Distribution Using Exact Response-Time Analysis

Toufik Sarni, Audrey Queudet and Patrick Valduriez **p107**
Software Transactional Memory: Worst Case Execution Time Analysis

Alfons Crespo, Ismael Ripoll, Patricia Balbastre, Miguel Masmano and Alan Burns **p115**
Contract based management of the memory resource

Design Optimization

Nathan Fisher **p127**
An FPTAS for Interface Selection in the Periodic Resource Model

Caroline Lu, Jean-Charles Fabre and Marc-Olivier Killijian **p137**
An approach for improving Fault-Tolerance in Automotive Modular Embedded Software

Tanguy Le Berre, Philippe Mauran, Gérard Padiou and Philippe Quéinnec **p147**
A Data Oriented Approach for Real-Time Systems

Multiprocessor Scheduling

Shelby Funk and Vijaykant Nadadur **p159**
LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets

Vandy Berten and Joel Goossens **p169**
Multiprocessor Global Scheduling on Frame-Based DVFS Systems

Author Index **p179**

Message from the General Chair

Dear Participants, Dear Guests,

It is my great pleasure to welcome you to the 17th edition of the Real-Time and Network Systems International Conference (RTNS'2009) that is taking place this year in Paris, on the campus of ECE, a Graduate School of Engineering.

I would like to take the opportunity to thank our sponsors for their support:

- IEEE-France Section for his Technical Co-Sponsoring of RTNS'2009
- INRIA French National Research Institute that will publish our proceedings in the INRIA HAL archiving system.
- ECE, serving as host organization for our conference and his Director Pascal Brouaye that has offered administrative support from ECE for the organization of our event. ECE has also sponsored the cash price for the best student paper award.
- OpticsValley, an association in computer science willing to promote the relationship between small and medium size businesses and research laboratories, for their important role in the promotion of our event.
- Astech, a French competitiveness center, promoting space and aeronautics projects, for their financial support of our event.

I would like also to thank Georgio Butazzo, Editor-in-Chief of Real-Time Systems (RTS) journal for his support of a special issue of best papers of RTNS'2009 to be published in RTS journal.

I would like to thank the authors for their high quality papers, the program committee and all their associated reviewers for their important efforts to review the increasing number of submissions.

I hope you will appreciate the final program composed this year of sixteen presentations and one invited talk on "Real-Time Scheduling for Control Systems" given by Enrico Bini from Scuola Sant'Anna, Pisa, Italy.

This year, the Junior Researcher Workshop on Real-Time Computing (JRWRTC'2009) associated to our conference was very well chaired by Charlotte Seiner. Eleven papers will be briefly presented associated to a Poster. Don't miss this occasion to share ideas with junior researchers.

Finally, I would like to thank our Co-Program Chairs Maryline Chetto and Mikael Sjodin for their great investment in RTNS'2009, to ensure the scientific quality of the conference.

I hope you enjoy the conference, and have a nice stay in Paris.

Laurent George
University of Paris 12 / ECE

Message from the Program Co-Chairs

On behalf of the Technical Committee of the 17th Real-Time and Network Systems Conference, we are pleased to welcome you to attend RTNS'2009. The conference will be held on October 26-27, 2009, at ECE graduate school of engineering in Paris, France.

The scope of the Conference covers all aspects of real-time systems and the program features the following:

- 16 regular papers which are partitioned into six conference sessions (uniprocessor scheduling, timing analysis, networks, resource management, design optimization, multiprocessor scheduling).
- A junior researcher workshop JRWRTC'2009
- An invited talk on *Real-Time Scheduling for Control Systems* by Enrico Bini from Scuola Sant'Anna, Pisa, Italy
- A selection of best papers to be invited for publication in Real-Time Systems journal, Springer Editor

A total of 42 paper submissions were received in response to the call for papers and were anonymously examined by 55 reviewers. *Real-Time Scheduling* is a topic proved to be especially active by a good number of submissions. The EasyChair conference management system was used for the handling of electronic submissions, allocation of reviewing duties, filing of reviews, and calculation of scores to support the electronically conducted program committee meeting. The result of the reviewing process gives an acceptance rate of 38 %. Special thanks to Damien Masson from ESIEE, Paris, for the setting of the Easychair conference submission system.

We would like to thank all the reviewers who spent their precious time to read the submitted manuscripts and comment on them. Thanks to the Program Committee members who discussed the papers, and helped us to ensure the high quality of the program.

This conference could not take place without the great investment in time and energy of our General Chair, Laurent George from the University of Paris 12 / ECE. We thank him for his guidance and efforts in coordinating RTNS 2009.

We are also grateful to the volunteer staffs at ECE, especially to Sabrina Mayet and Anne-Marie Patard, for their roles in the local arrangement for the conference and for the organisation of the social event on the Eiffel Tower.

Last but not least, we would like to thank INRIA for the publication of the proceedings in HAL conference indexing system.

We hope you enjoy both the technical program of RTNS 2009 and the beautiful city of Paris!

Maryline Chetto, University of Nantes, France,
Mikael Sjodin , Malardalen University, Sweden,

Conference Organization

General Chair

Laurent George, University of Paris 12 / ECE, France

Co-Program Chairs

Maryline Chetto, University of Nantes, France
Mikael Sjodin, Malardalen University, Sweden

Steering Committee

P. Minet, INRIA-Hipercom, Rocquencourt, France
N. Navet, INRIA-Loria, Nancy, France
F. Simonot-Lion, LORIA-INPL, Nancy, France
I. Puaut, University of Rennes1 / IRISA, France
G. Juanole, LAAS, Toulouse, France
P. Richard, LISI / Poitiers, France
J. Goossens, ULB, Brussels, Belgium

Local Organisation Co-Chairs

Laurent George, University of Paris 12 / ECE, France
Damien Masson, ESIEE, France
Ikbal Benakila, ECE, France

Publicity Chair

Serge Midonnet, University of Marne La Vallée, France

Program Committee

S. Baruah, University of North Carolina, USA
E. Bini, Scuola Superiore Sant'Anna, Pisa, Italy
M. Chetto, IRCCyN, Nantes, France
A. Crespo, Polytechnic University of Valencia, Spain
J-D. Decotignie, CSEM, Neuchatel, Switzerland
T. Facchinetti, University of Pavia, Italy
N. Fisher, Wayne State University, US
J. A. Fonseca, University of Aveiro, Portugal
L. George, University of Paris 12 / ECE, France
J. Goossens, ULB, Brussels,Belgium
G. Juanole, LAAS, Toulouse, France
J. Kaiser, University of Magdeburg, Germany
R. Kirner, TU Vienna, Austria
T-W. Kuo, National Taiwan University, Taiwan
L. Lo Bello, University of Catania, Italy
Z. Mammeri, IRIT/UPS Toulouse, France
P. Marquet, INRIA/LIFL, Lille, France
S. Midonnet, University of Paris-Est, Marne la Vallée, France
P. Minet, INRIA-Rocquencourt, France
D. Mosse, University of Pittsburgh, USA
N. Navet, INRIA-Loria, Nancy, France
N. Nissanke, London South Bank University, UK
M. Sjoden, Mlardalen University, Sweden
M. Pouzet, Université Paris Sud-LRI, France
I. Puaut, University of Rennes/IRISA, France
P. Richard, LISI, Poitiers, France
C. Rochange, IRIT Toulouse, France
G. Rodriguez-Navas, University of Balearic Islands, Palma de Mallorca, Spain
B. Sadeg, LITIS - University of Le Havre, France
D. Simon, INRIA-Rhônes Alpes, France
F. Simonot-Lion, LORIA-INPL, Nancy, France
Y. Sorel, INRIA-Rocquencourt, France
E. Tovar, Polytechnic Institute of Porto, Portugal
Y. Trinquet, IRCCyN, Nantes, France
F. Vasques, University of Porto, Portugal
F. Vernadat, LAAS, Toulouse, France

List of Reviewers

Slim Abdellatif
Björn Andersson
Sanjoy Baruah
Jean-Luc Béchennec
Mohammed Benakila
Vandy Berten
Enrico Bini
Konstantinos Bletsas
Mario Calha
Maryline Chetto
Alfons Crespo
Silvano Dal Zilio
Jean-Dominique Decotignie
Arvind Easwaran
Andreas Ermedahl
Tullio Facchinetti
Hua-Wei Fang
Joaquim Ferreira
Mamoun Filali-Amine
Nathan W. Fisher
Jose Fonseca
Laurent George
Emmanuel Grolleau
Joel Goossens
Jean-Francois Hermant
Pierre-Emmanuel Hladik
Joerg Kaiser
Raimund Kirner
Didier Le Botlan
Yung-Feng Lu
Zoubir Mammeri
Patrick Meumeu
Serge Midonnet
Pascale Minet
Daniel Mosse
Nicolas Navet
Nimal Nissanke
Patrick Pons
Marc Pouzet
Isabelle Puaut
Ihsan Qazi
Khaled Refaat
Pascal Richard
Christine Rochange
Guillermo Rodriguez-Navas
Bruno Sadeg
Daniel Simon
Francoise Simonot
Mikael Sjodin
YeQiong Song
Yves Sorel
Eduardo Tovar
Yvon Trinquet
John Yackovich
Chuan-Yue Yang

Keynote Speaker

Enrico Bini, Scuola Sant'Anna, Pisa, Italy

Real-Time Scheduling for Control Systems

Schedulability analysis consists of performing a guarantee test to verify whether a given scheduling algorithm is able to execute a set of real-time tasks within their deadlines, assuming their values are known and given in advance. However, at a design stage, it is not always clear how task deadlines should be assigned to best meet the system requirements.

Also, in Fixed Priority scheduling the process of assigning priorities is often driven by the relative "importance" of the tasks in the application. However, there may be very important tasks that could run with lower priority, as well as less important tasks that are sensitive to delay that would require to run at higher priority.

The problem of assigning performance parameters (such as priorities, deadlines or periods) is that their effect is difficult to measure quantitatively in terms of application requirements.

Control systems represent a case in which measuring the performance is possible and there are techniques that relate stability, speed of convergence, and sampling error to performance requirements.

This talk presents an overview of the techniques that can be used to design control systems taking performance requirements into account since a design stage. Extending such methods to other application fields is also discussed.

Uniprocessor Scheduling

Uniprocessor Schedulability and Sensitivity Analysis of Multiple Criticality Tasks with Fixed-Priorities

François DORIN, Pascal RICHARD, Michaël RICHARD
LISI
ENSMA - Université de Poitiers
1 rue Clément Ader, BP 40109,
86961 Chasseneuil du Poitou Cedex, France
{dorinfr, richardp, richardm}@ensma.fr

Joël GOOSSENS
Computer Science Department
Université Libre de Bruxelles
Boulevard du Triomphe - C.P. 212
1050 Bruxelles, Belgium
joel.goossens@ulb.ac.be

Abstract

Safety-critical real-time standards define several criticality levels for the tasks (e.g., DO-178B - Software Considerations in Airborne Systems and Equipment Certification). Classical models do not take into account these levels. Vestal introduced a new multiple criticality model, to model more precisely existing real-time systems, and algorithms to schedule such systems. Such task model represents a potentially very significant advance in the modeling of safety-critical real-time systems. Baruah and Vestal continues this investigation, with a new algorithm under fixed and dynamic priority policies.

In this paper, we provide some results about the optimality of Vestal's algorithm and analyze an interesting property of this algorithm. We also adapt sensitivity analysis developed by Bini et al. for multiple criticality systems.

1. Introduction

Execution times of a recurring task are different from one execution to another. Schedulability analysis of real-time systems is based on the worst-case execution time (WCET). The execution time of a task never exceeds its WCET otherwise it is impossible to guarantee the system schedulability. Determining an exact WCET value for every task occurrence is a very difficult problem. So in practice, the used WCET is an upper bound of execution requirements.

Since computing WCET is a complex problem, two different approaches can be considered:

- The first one is to allow some WCET exceedance (for instance, due to a optimistic approximation of the WCETs). Some models allow to take into account this kind of problem. For example, Bougueroua, in [7], introduced the notion of allowance to achieve this aim.
- The second one is to consider several levels of confidence for WCET. A high required confidence task have to never miss a deadline whereas a low required confidence task can miss some deadline sometimes without great consequences on the safety of the whole system. In such cases, the WCET of high required confidence tasks have to be evaluated with the maximum possible precision because an underestimated value can cause the task to miss a deadline, which can be very critical for the system, and an overestimated value can lead a feasibility test to conclude that a task is not feasible whereas no deadline miss can occur at run-time. So, the idea is to perform tight evaluation of the WCET for tasks having a high confidence level and to allow more approximate (i.e., average) evaluation for tasks with low confidence levels.

Some software standards define several criticality levels which define several levels of required confidence. For example, the RTCA DO-178B software standard [3] defines 5 levels of criticity, denoted from A to E. A failure of a A-criticality task can have catastrophic results (i.e., crash of an airplane) whereas a failure of a E-criticality task has no effect on the safety of the airplane. The failure conditions, reported in Table 1, are categorized by their effects on the aircraft, crew, and passengers.

A way to take into account these different levels of confidence is to perform a time partitioning between the different software applications which allows to enforce temporal isolation of tasks like described, for example, in the ARINC 653 standard [1]. ARINC 653 is an Application Programming Interface that provides time partitioning among applications having different required Design Assurance Levels. The timeline is defined as a set of time partitions. Each partition has a fixed predetermined amount of time. Each task (or a set of dependent tasks) is attached to a partition and a classical scheduling algorithm is executed on each partition. Since each partition has a fixed predetermined amount of allocated time, a partition cannot interfere with another one. In other words, a

Level	Failure condition	Description
A	Catastrophic	Failure may cause a crash.
B	Hazardous	Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the plane due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
C	Major	Failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries).
D	Minor	Failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine flight plan change)
E	No Effect	Failure has no impact on safety, aircraft operation, or crew workload.

Table 1. The required Design Assurance Level in the DO-178B.

task, which belongs to a partition A, cannot interfere with a task which belongs to a partition B. Moreover, by affecting task with the same required level of confidence on the same partition, it is possible to ensure temporal isolation between tasks requiring different levels of confidence.

Another way to take into account different levels of confidence was discussed by Vestal in a recent paper [13]. Vestal introduced a new formal model for representing real-time task sets. This model, based on the consideration of several WCETs instead of a single one, allows to require more or less confidence depending on the criticality of the tasks. Baruah and Vestal gave this definition in [4]: 'the more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to be in practice'.

In [13], Vestal provided two fixed priority algorithms in order to schedule such systems: one based on period transformation [12] and another based on the Audsley's algorithm [2]. In [4], these works were completed by Vestal and Baruah. They established a link between classical sporadic task systems and multiple criticality task systems. The corresponding sporadic task system is defined as the initial multiple criticality task set in which only the WCET corresponding to its critical confidence level is considered for every task. They proved an interesting property for the feasibility analysis: a multiple criticality sporadic task system is feasible if and only if the corresponding traditional sporadic task system is feasible (i.e., schedulable when temporal isolation of task executions is enforced by the operating system).

On-line scheduling algorithms can be classified into three different categories: fixed-task-priority (FTP, all occurences of a given task have the same priority as for Rate Monotonic (RM) or Deadline Monotonic (DM) priority assignment policies); fixed-job-priority (FJP, every job has a fixed priority, but subsequent jobs of a given task can have different priorities - the Earliest Deadline First (EDF) is such an algorithm); and lastly, Dynamic Priority (DP, the most general class of scheduling algorithms). For Liu and Layland's task systems, a classical result is that FTP scheduling algorithms are dominated by EDF [11]. That is to say, if a task system is schedulable by an FTP scheduling algorithm, then it is schedulable by EDF. This result does not hold for multiple criticality task system since Baruah and Vestal gave a counter-example of a task system which can be scheduled by FTP algorithm and cannot be scheduled by EDF. In other words, FTP scheduling algorithms and EDF cannot be compared.

To overcome the fact that EDF and FTP algorithms are not comparable, Vestal and Baruah proposed an hybrid-priority scheduling algorithm able to schedule any task system schedulable by Vestal's algorithm and/or by EDF, that is to say by any FTP algorithm or by EDF, since Vestal's algorithm is optimal for the FTP algorithm class. This hybrid-priority scheduling belongs to the class of the fixed job-priority (FJP) scheduling. A last result provided in [4] is that this hybrid-priority scheduling is not optimal in the FJP algorithm class.

This Research. In this paper, we give a modest step in the study of multiple criticality task systems. Precisely, we provide a complete proof that the original Audsley's algorithm already is optimal for this kind of problem. We then analyse the sensitivity of system parameters from processor speed and task execution requirements:

- What is the required processor speed so that a multiple criticality task set is schedulable under Vestal's algorithm. Precisely, we show that Vestal's algorithm can be easily adapted to compute such a processor speed.
- What is the the allowed variations of WCETs of a task so that it is still schedulable. For that purpose, we adapt the sensitivity analysis introduced by Bini in [6] for analyzing multi-criticality task systems scheduled under a FTP scheduling policy.

Organization. The paper is organized as follow: Section 2 introduces the multiple criticality model as well as some known results we will discuss later. We prove, in Section 3, the optimality of the original Audsley's algorithm [2] for the kind of independent task systems with constrained-deadlines under fixed priority policy. Section 4 deals with Vestal's algorithm, and the fact the returned schedule has the highest critical scaling factor among all the possible schedules. In Section 5, we performed sensitivity analysis on multiple criticality based systems followed by an example.

2. Task Model and known results

2.1. Task Model

The model developed by Vestal in [13] is based on the classical Liu and Layland's one [11]. Let τ denote a task system composed of n tasks. Each task τ_i for $i = 1, \dots, n$ is composed of:

- a worst-case execution time C_i, which corresponds to the required processor time per instance of the task,
- a period T_i, which is the minimum inter-arrival separation time between two consecutive instances of the task τ_i,
- a relative deadline D_i, which corresponds to the maximum autorized amount of time between the activation and the end of an instance of the task τ_i.

For the multiple criticality model, Vestal introduced the following parameters:

- A WCET function $C_i : \mathbb{N}^+ \rightarrow \mathbb{R}^+$, which specifies the WCET for different criticality levels. We can notice that C_i is no more a constant for a given task but a function. Thus, the WCET for the criticality level ℓ is denoted by $C_i(\ell)$.
- A criticality level L_i, $L_i \in \mathbb{N}^+$, which specifies the required confidence for the task τ_i. By convention, it is assumed that the level 1 is the lowest critical level.

In addition to these parameters, we introduce the priority π_i of a task τ_i, which allows to determine which task have to be executed at a given time: the task with the highest priority is executed first. By convention, a high numerical value for π_i denotes a low priority task. Thus, the task having a priority equal to 0 has the highest priority.

In this paper, we assume that tasks have constrained-deadlines (i.e., $D_i \leq T_i$ for each task). $u_i(\ell) \stackrel{\text{def}}{=} C_i(\ell)/T_i$ denotes the processor utilization factor of task τ_i and the system utilization factor is the sum of task utilization factors. Any task set having a utilization factor greater that 1 is said overloaded and it is well known that such system cannot be scheduled by any DP scheduling algorithm. Moreover, it is supposed that tightness of WCET increases according to critical levels. Thus, for all task τ_i and for all criticality level l, the following relation is verified [13]:

$$C_i(\ell) \leq C_i(\ell + 1) \tag{1}$$

From a multi-criticality task τ set can be defined the corresponding sporadic task system τ' as follows: to every multi-critical task τ_i is defined a corresponding sporadic task $\tau_i'(C_i(L_i), D_i, T_i)$. The key assumption to enforce (i.e., Theorem 1 [4]) that a multiple criticality sporadic task system is schedulable if and only if the corresponding traditional sporadic task system is feasible, it must be enforced that:

$$\forall i \in [1, n], \forall j \in [1, L_i], C_i(j) = C_i(L_i) \tag{2}$$

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
1	5	5	1	1	2
2	5	5	2	2	5

Table 2. Example of violation

Let us consider the task set presented in Table 2. The classical schedulability analysis of the corresponding task set concludes that it is unschedulable, since no DP algorithm can schedule a task set having the utilization factor is greater than 1:

$$\frac{C_1(L_1)}{T_1} + \frac{C_2(L_2)}{T_2} = 1.2 > 1 \tag{3}$$

However, from a multiple criticality schedulability analysis, this task system is schedulable when assigning the highest priority to the task τ_2 and the lowest priority to the task τ_1. The corresponding worst-case response time of τ_1 and τ_2 are:

$$Tr_1 = C_1(L_1) = 5 \leq D_1 \tag{4}$$

$$Tr_2 = C_1(L_2) + C_2(L_2) = 3 \leq D_2 \tag{5}$$

Thus, all deadlines seem to be met which is obviously impossible in any overloaded system. Remember that task execution requirements must satisfy the Equation (2) for every multiple criticality task.

2.2. Known results

Scheduling algorithm In [13], Vestal introduced a modified version of the Audsley's algorithm [2]. The Vestal's algorithm is optimal in the category of the fixed priority algorithms for independent task systems with constrained-deadlines [4].

The Audsley's algorithm is based on the following observation: the response time of a task depends only of the set of the higher priority tasks, and it is unnecessary to know the exact priority assignment. So, the principle of the Audsley's algorithm is to enumerate each priority level from the lowest to the highest. At each priority level is assigned the first task which is schedulable at this priority level (ties are broken arbitrarily). If there is at least one priority level with no task which can be assigned to it, then the task system is unschedulable using a fixed-priority algorithm.

Vestal modified this algorithm in the following way: instead of taking the first task which can be scheduled at a given priority level, Vestal's algorithm assigns the task with the highest critical scaling factor i.e., factor which corresponds to the maximum factor by which we can multiply the WCET of the task without the task τ_i missed a deadline [9]. We recall the previse definition of the critical scaling factor of a system because it will be reused hereafter:

$$\Delta^* \stackrel{\text{def}}{=} \left[\max_{1 \leq i \leq n} \min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right]^{-1} \tag{6}$$

where S_i is the set of scheduling points as defined in [10]:

$$S_i \overset{\text{def}}{=} \left\{ kT_j | j = 1, \ldots, i; k = 1, \ldots, \left\lfloor \frac{D_i}{T_j} \right\rfloor \right\} \qquad (7)$$

This critical scaling factor corresponds to the maximum factor by which we can multiply all C_i of the tasks without a deadline failure. If we consider tasks separately, the critical factor of a task can be defined as follows:

$$\Delta_i \overset{\text{def}}{=} \left[\min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right]^{-1} \qquad (8)$$

In [4], Baruah and Vestal claimed that this algorithm is optimal for scheduling independent task sets with constrained-deadlines under a fixed priority policy without providing a complete proof.

We show next that the original Audsley's algorithm already is optimal for this kind of problem (i.e., without considering the critical scaling factors as a tie braking rule) and, as a consequence, that Vestal's algorithm also is optimal. We also show that Vestal's algorithm returns a schedule having the highest possible critical scaling factor.

We give an example of Vestal's assignment algorithm in Figure 1. The upper table summarizes the task characteristics. The bottom table is a trace of Vestal's algorithm. For example, when we are looking for a task to assign at the priority level 3, we compute the critical scaling factor of each task, and we choose the one having the highest critical scaling factor which is, in this case, τ_3. So, we continue this process at the priority level 2 without forget to remove task τ_3. The task with the highest critical scaling factor at this level is τ_0, so τ_0 is assigned at the priority level 2. And so on.

The critical scaling factor of the system is given by the minimum of the critical scaling factor of each task when all tasks are assigned of a priority. In this case, the critical scaling factor of the system is determined by the critical scaling factor of τ_3.

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
0	164	104	1	7	17
1	89	44	2	4	4
2	191	80	1	12	16
3	283	283	2	85	85

Priority	Trace
3	$\Delta_{\tau_0} = 0.928571$, $\Delta_{\tau_1} = 0.360656$, $\Delta_{\tau_2} = 0.740741$, $\Delta_{\tau_3} = 1.69461$ $\Rightarrow \pi_3 = 3$
2	$\Delta_{\tau_0} = 3.86957$, $\Delta_{\tau_1} = 1.18919$, $\Delta_{\tau_2} = 3.47826$ $\Rightarrow \pi_0 = 2$
1	$\Delta_{\tau_1} = 2.2$, $\Delta_{\tau_2} = 5$ $\Rightarrow \pi_2 = 1$
0	$\Delta_{\tau_1} = 11$ $\Rightarrow \pi_1 = 0$

$\Delta = min_{\Delta_i} = 1.69461$

Figure 1. Vestal's priority assignment trace

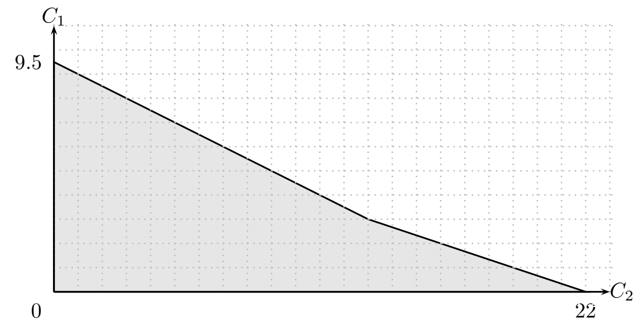


Figure 2. Example of the representation of the $\mathbb{C}$-space of a system composed of 2 tasks, with $T_1 = D_1 = 9.5$ and $T_2 = D_2 = 22$

Schedulability analysis. In [9], the critical scaling factor is a basic sensitivity analysis on independent task systems under fixed priority policy.

In [6], Bini et al. performed a sensitivity analysis which extends the Lehoczky's one. Two methods are described: one to perform schedulability in the $\mathbb{C}$-space (i.e., studying the modification of the execution time C_i of the tasks), and an other in the f-space (i.e., studying the modification of the period T_i of the tasks).

This method allows to represent graphically these spaces (i.e., Figure 2 for an example of a $\mathbb{C}$-space graphically represented).

In the following, we focus on schedulability in $\mathbb{C}$-space since we are interested by the impact of using a model with several WCETs per task instead a single one. So, readers interested by schedulability in f-space can report themself to the original paper from Bini et al. [6].

The method to perform sensitivity analysis on the $\mathbb{C}$-space allows to choose the direction in which we want to perform the analysis, that is to say to choose which subset of tasks we want to study, and the weighting for each task.

The starting point of the method is the fact that a task system is schedulable if, and only if:

$$\max_{1 \le i \le n} \min_{t \in S_i} \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \le t \qquad (9)$$

or, in a vectorial form:

$$\max_{1 \le i \le n} \min_{t \in S_i} \mathbb{C}_i n_i(t) \le t \qquad (10)$$

where $\mathbb{C}_i$ is a vector of the i highest priority task $\mathbb{C}_i = (C_1, C_2, \ldots, C_i)$, and $n_i(t) = \left(\left\lceil \frac{t}{T_1} \right\rceil, \left\lceil \frac{t}{T_2} \right\rceil, \ldots, \left\lceil \frac{t}{T_{i-1}} \right\rceil, 1 \right)$.

By replacing $\mathbb{C}_i$ by $\mathbb{C}_i + \lambda d_i$ in the Equation 10, we

obtain (the complete proof can be found in [6]):

$$\lambda = \min_{i=1,\dots,n} \max_{t \in sched(P_i)} \frac{t - n_i(t)\mathbb{C}_i}{n_i(t)d_i} \tag{11}$$

where λ is a scaling factor and $sched(P_i)$ is a subset of S_i.

The vector d_i correspond to the studied direction. If we want to perform schedulability analysis on τ_k only, then d_i is equal to $(\underbrace{(0,\dots,0,\overbrace{1}^{k^{th}\text{ element}},0,\dots,0}_{i\text{ elements}})$.

If we want to perform a sensitivity analysis on the whole system, then d_i must be equal to $\mathbb{C}_i$. The corresponding analysis leads to define the critical scaling factor of the system.

The schedulability in the $\mathbb{C}$-space is a generalization of the schedulability analysis introduced by Lehoczky in [9] in the sense that the computation of a critical factor for a single task or for the whole tasks system are particular cases of the Bini's method. Indeed, Bini's method allows to choose the direction on which the sensitivity analysis is performed. Thus, it is possible to study only one task, the whole task system or any subset of tasks of the system.

In this paper, one of our contributions is to adapt this algorithm to multiple criticality task systems (see Section 5) in the case of sensitivity analysis on the $\mathbb{C}$-space.

3. Optimality of Audsley's algorithm

Our first contribution corresponds to the following result:

Theorem 1. *The Audsley's algorithm is optimal for scheduling multiple criticality independent task systems with constrained-deadlines under a fixed-priority policy.*

To prove this theorem, we will use the lemmas described next:

Lemma 1. *When studying a specific task τ_i, we can consider corresponding task system instead of a multiple criticality task system, with the WCETs corresponding to the ones on critical level L_i, the criticality level of the studied task τ_i.*

Proof. This lemma can be deduced from the definition of a multiple criticality task system. When we compute the worst-case response time (WCRT) of the task τ_i, we consider only the WCET of the criticality level of τ_i as we can see in the following equation, which is the modified version of the Joseph and Pandya's equation [8] introduced by Vestal in [13] to compute the WCRT for multiple criticality systems:

$$Tr_i = \sum_{j=1}^{i} \left\lceil \frac{Tr_i}{T_j} \right\rceil C_j(L_i) \tag{12}$$

Thus, when we are studying the task τ_i, we can consider only a classical task system with the WCETs corresponding to the WCET of the criticality level of τ_i, that is to say L_i. □

If we have a look to the task system given in Figure 1, we can see that the critical scaling factor of task τ_1 when assigned at the priority 2 is greater than the critical scaling factor of the task τ_1 when assigned at the priority 3 (i.e., at a lower priority level). This intuitive result is summarized in the following lemma:

Lemma 2. *Let τ_i to be a task which has a critical factor of $\Delta_{i,j}$ when assigned of the priority j. If τ_i is assigned of the priority $j-1$ then the critical factor of τ_i for this priority verifies $\Delta_{i,j} < \Delta_{i,j-1}$*

Proof. For the following proof, we will consider a task τ_i which can be assigned at the level priority j or $j-1$. It is important to notice that the only difference between these two assignments is that the set of higher priority tasks, when τ_i is assigned at the priority level j contains one additional task than the set of higher priority tasks when τ_i is assigned at the priority level $j-1$. By convenience, we suppose the additional task to be τ_j, but since the task set of higher priority tasks are not ordered, it can be any higher priority task.

By definition, from [9]

$$\Delta_{i,j} \stackrel{\text{def}}{=} \left[\min_{t \in S_{i,j}} \frac{1}{t} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \right]^{-1} \tag{13}$$

$$\Delta_{i,j-1} \stackrel{\text{def}}{=} \left[\min_{t \in S_{i,j-1}} \frac{1}{t} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \right]^{-1} \tag{14}$$

These definitions were just adapted to multiple criticality task systems, replacing classical WCET C_k by multiple criticality task WCET at level L_i which is equal to $C_k(L_i)$.

$S_{i,j}$ denotes the set of scheduling points for the task τ_i when assigned of the priority j. This set is defined by the following equation:

$$S_{i,j} \stackrel{\text{def}}{=} \left\{ kT_m | m = 1,\dots,j; k = 1,\dots,\left\lfloor \frac{D_i}{T_m} \right\rfloor \right\} \cup \{D_i\} \tag{15}$$

We were aware that Bini et al. introduced in [5] a sufficient subset of scheduling points, but for our proof, we need to consider the set of all scheduling points.

So, according to Equations 13 and 14, there exists t_j and t_{j-1} such as

$$\Delta_{i,j} = \left[\frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \right]^{-1} \tag{16}$$

$$\Delta_{i,j-1} = \left[\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil \right]^{-1} \tag{17}$$

One can remark than $S_{i,j-1} \subset S_{i,j}$. So, we have two cases to take into account: $t_j \in S_{i,j-1}$ and $t_j \notin S_{i,j-1}$:

- If $t_j \in S_{i,j-1}$. It is obvious that:

$$\forall t, \frac{1}{t}\sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t}{T_k}\right\rceil > \frac{1}{t}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t}{T_k}\right\rceil \quad (18)$$

So, if $t = t_j$ then:

$$\frac{1}{t_j}\sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil > \frac{1}{t_j}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (19)$$

Since $t_j \in S_{i,j-1}$ and t_{j-1} minimize $\frac{1}{t}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t}{T_k}\right\rceil$ (see definition of t_{j-1}, Equation 17), we have:

$$\frac{1}{t_{j-1}}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_{j-1}}{T_k}\right\rceil \leq \frac{1}{t_j}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (20)$$

Equations 19 and 20 give:

$$\frac{1}{t_{j-1}}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_{j-1}}{T_k}\right\rceil < \frac{1}{t_j}\sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (21)$$

That is to say:

$$\Delta_{i,j-1} > \Delta_{i,j} \quad (22)$$

- Now, we consider the case when $t_j \notin S_{i,j-1}$.

By definition, we have to notice that $D_i = \max(S_{i,j})$ and $D_i = max(S_{i,j-1})$. Since $t_j \notin S_{i,j-1}$, we have $t_j \neq D_i$. So,

$$\exists t_k \in S_{i,j-1}, t_j < t_k \quad (23)$$

We can notice than $\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t}{T_k}\right\rceil$ is a piece-wise function and t_j is not a point of discontinuity since $t_j \notin S_{i,j-1}$, so:

$$\exists t_k \in S_{i,j-1}, \begin{cases} t_k > t_j \\ \sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil = \sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_k}{T_k}\right\rceil \end{cases} \quad (24)$$

Moreover,

$$\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil < \sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (25)$$

So, Equations 24 and 25 lead to:

$$\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_k}{T_k}\right\rceil < \sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (26)$$

Since $t_k > t_j$, we have $\frac{1}{t_k} < \frac{1}{t_j}$. And, if we use Equation 26, we have:

$$\frac{1}{t_k}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_k}{T_k}\right\rceil < \frac{1}{t_j}\sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (27)$$

By definition of t_{j-1} (i.e., Equation 17), we have

$$\frac{1}{t_{j-1}}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_{j-1}}{T_k}\right\rceil = \min_{t \in S_{i,j-1}} \frac{1}{t}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t}{T_k}\right\rceil \quad (28)$$

And then, since $t_k \in S_{i,j-1}$:

$$\frac{1}{t_{j-1}}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_{j-1}}{T_k}\right\rceil \leq \frac{1}{t_k}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_k}{T_k}\right\rceil \quad (29)$$

If we combine Equations 27 and 29, we obtain:

$$\frac{1}{t_{j-1}}\sum_{k=1}^{j-1} C_k(L_i)\left\lceil \frac{t_{j-1}}{T_k}\right\rceil < \frac{1}{t_j}\sum_{k=1}^{j} C_k(L_i)\left\lceil \frac{t_j}{T_k}\right\rceil \quad (30)$$

That is to say,

$$\Delta_{i,j-1} > \Delta_{i,j} \quad (31)$$

We proved that in both cases ($t_j \in S_{i,j-1}$ and $t_j \notin S_{i,j-1}$), $\Delta_{i,j-1} > \Delta_{i,j}$. This prove the lemma. □

Now, we have the material to prove Theorem 1.

Proof of Theorem 1. Using Lemma 1, studying the schedulability of a multiple criticality task can be done by studying the schedulability of the equivalent task system on the criticality level of the studied task. And taking into account Lemma 2, the critical scaling factor of a task can only increase when we assign the task to a higher priority level. In other word, the interference due to higher priority tasks can only decrease.

Thus, if a task is schedulable at a priority level j, then it is schedulable when assigned of a higher priority. Since the hypothesis of the classical task model are also respected in the case of the multiple criticality task model, we can deduce that the Audsley's algorithm is also optimal for multiple criticality task systems. □

And having the previous theorem, we can easily state the following theorem:

Theorem 2. *The Vestal's algorithm is optimal to schedule a set of independent tasks with constrained-deadlines under a fixed priority scheduling policy.*

Proof. Since Vestal's algorithm is a particular case of Audsley's algorithm (i.e., task critical scaling factors are used for braking ties), and since Audsley's algorithm is optimal due to Theorem 1, we can conclude that Vestal's algorithm is also optimal to schedule independent task systems with constrained-deadlines under fixed priority policy. □

4. Processor speed

For multiple criticality task systems, Audsley's algorithm is optimal. But, if the system is not schedulable, then computing the minimum amount of supplementary processor speed so that the system becomes schedulable under a FP assignment is an important issue for system designers.

Clearly, for sporadic tasks with constrained-deadlines, priority assignment (i.e., DM) and speed up factor computation are independent problems. We prove next that such a result is also valid for multiple criticality task system and furthermore that both problem can be solved simultaneously (i.e., the speed up factor can be computed in a greedy manner while performing the priority assignment).

Algorithm 1 Processor speed modulation and priority assignment

```
Require: τ* = set of tasks to schedule
Ensure: Δ* = maximum scaling factor
Ensure: τ̃ = scheduled task system
  τ ⇐ τ*
  τ̃ ⇐ ∅
  for j from n to 1 do
    τ_Vestal = ∅
    for τ_A ∈ τ do
      if τ_Vestal = ∅ then
        τ_Vestal ⇐ τ_A
        Δ* = Δ(τ_A, τ)
      else
        if Δ(τ_Vestal, τ) < Δ(τ_A, τ) then
          τ_Vestal ⇐ τ_A
        end if
      end if
    end for
    π(τ_Vestal) ⇐ j
    τ ⇐ τ − {τ_Vestal}
    τ̃ ⇐ τ̃ ∪ {τ_Vestal}
    if Δ(τ_Vestal, τ) < Δ* then
      Δ* = Δ(τ_Vestal, τ)
    end if
  end for
```

The Algorithm 1 presents an implementation of our algorithm in pseudo-code. It computes a priority assignment and a critical scaling factor Δ^*. The function $\Delta(\tau_i, \tau)$ computes the critical scaling factor of the task τ_i when the higher or equal priority task set is equal to τ.

If the critical scaling factor Δ^* is greater than 1, then it corresponds to the maximum factor by which we can divide the processor speed without having deadline failure. If $\Delta^* < 1$, then the initial task set is not schedulable and Δ^* corresponds to the minimum factor by which the processor speed must be accelerated to lead to a schedulable task system.

The main result (i.e., Theorem 3) will be based on the following property:

Lemma 3. *Let τ denote a task system and τ_i and τ_j be two tasks with τ_i having a higher priority than τ_j. If the critical scaling factor of the task τ_i at the priority level of τ_j is greater than the critical factor of the task τ_j at the same level, then inserting the task τ_i at the priority level of the task τ_j can only increase the critical factor Δ of the task system.*

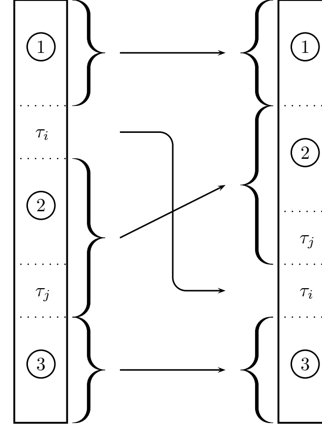


Figure 3. Scheme of the transformation

Proof. We shall use an interchange argument to prove the result. The Figure 3 represents the basis of the transformation. Each zone corresponds to the following:

- Zone 1 is composed of tasks with higher priority than task τ_i,
- Zone 2 is composed of tasks with intermediate priority, that is to say with lower priority than τ_i but higher priority than τ_j,
- Zone 3 is composed of tasks with lower priority than the task τ_j.

If we study the evolution of the critical scaling factor of each task when performing the transformation, we can observe that:

- The critical factor of tasks in Zone 1 are not modified by the priority modifications of tasks with lower priority,
- The critical factor of tasks in Zone 3 are not modified by the modifications of the priority order of tasks of higher priority,
- The critical factor of tasks in Zone 2 can only increase due to Lemma 2.

And if we perform the transformation, it is, by hypothesis because task τ_i has a higher critical scaling factor at priority level of τ_j than τ_j.

In other words, in all the cases, the critical scaling factor of each task can be either unchanged or increased, except for task τ_i. But by assumption the new critical scaling factor of task τ_i is greater than the old critical scaling factor of task τ_j. The result follows. □

Now, using this lemma, it is easy to prove the following theorem.

Theorem 3. *Vestal's algorithm returns a priority assignment with the greatest critical scaling factor of tasks (i.e., minimum speed up factor if the system is not schedulable under a unit-speed processor).*

Proof. Let τ denote the task system. This task system is composed of n tasks, $\tau_1, \ldots, \tau_n$, and each task is assigned to a priority. To prove the result, we build-up Vestal's schedule from τ using Lemma 3. The method is straightforward: we are looking for the task having the highest critical scaling factor at the priority level n among the tasks having a priority higher or equal to n. Then, we insert this task to this level. Due to Theorem 2, the critical scaling factor of the new task system τ' is greater or equal to the critical scaling factor of Δ. We repeat this operation, replacing τ by τ' and looking for the task to insert at the level priority $n-1$, and so on until the studied priority task level is equal to 1.

By this way, we construct a new schedule from the initial one, which is the same than this one produced by Vestal's algorithm because in both cases, the same task selection is performed. Since the transformation used can only increase the critical scaling factor of the initial task set τ and since the initial task set τ can represent any task set, we can conclude that the task set resulting of Vestal's algorithm has the highest possible critical scaling factor for fixed priority policy. This proves the Theorem 3. □

So, Vestal's algorithm, by providing a schedule with the highest possible critical scaling factor, has a great interest since it offers a simple way to define the minimum processor requirement so that a multiple criticality task set is schedulable.

5. Sensitivity analysis on WCET

We next adapt the Bini et al. sensitivity analysis (i.e., initially developed for classical real-time task systems [6]) to multiple criticality task systems. We only focus to the sensitivity analysis in the $\mathbb{C}$-space, since the multiple criticality task model distinguishes from classical sporadic task systems by considering a set of WCETs for every task).

5.1. Sensitivity analysis in the $\mathbb{C}$-space

We extend the sensitivity analysis in the $\mathbb{C}$-space by analyzing tasks at the same critical level. Instead of having one λ in the studied direction d, we define one λ_ℓ per criticity level ℓ.

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	86	86
τ_3	248	168	1	32	160

Table 3. Example of a multiple criticality tasks system

$$\lambda_\ell \stackrel{\text{def}}{=} \min_{\substack{i=1,\ldots,n \\ L_i=\ell}} \max_{t\in\text{sched}(\text{P}_\text{i})} \frac{t - n_i(t)\mathbb{C}_i(\ell)}{n_i(t)d_i} \quad (32)$$

A particular attention must be focused on the modified C_i. Indeed, the modifications can break a basic assumption of multiple criticality system expressed by Equation 1 (a complete example is detailed in the next section). In practice, such a problem can be easily solved by setting Equation 2 as a constraint in Bini et al. sensitivity analysis method. Precisely, it is necessary to normalize execution requirements of every task so that the assumption on execution time stated in the task model is respected (i.e., Equation 1).

For that purpose every time that Equation 1 is not satisfied:

$$\exists \ell, C_i(\ell) > C_i(\ell+1) \quad (33)$$

then, we assign the value of C_i at criticality level $\ell+1$ to the C_i at criticality level ℓ

$$C_i(\ell) \leftarrow C_i(\ell+1) \quad (34)$$

5.2. Example

After this simple normalization step, Bini et al. sensitivity analysis can be easily performed. Let study the example of multiple criticality task system where characteristics are given Table 3.

And let focus on the task τ_2 on which we will perform the sensitivity analysis. Bini et al. showed in [6] that when the schedulability analysis is performed only on a single task, Equation 32[1] can be rewritten in:

$$\delta C_k^{\max} = \min_{i=k,\ldots,n} \max_{t\in\text{sched}(\text{P}_\text{i})} \frac{t - n_i(t)\mathbb{C}_i}{\left\lceil \frac{t}{T_k} \right\rceil} \quad (35)$$

To apply the sensitivity analysis, scheduling points must be computed. In [5], Bini uses these recursive definition to find them:

$$\begin{cases} \text{sched}(\text{P}_\text{i}) \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i) \\ \mathcal{P}_0(t) \stackrel{\text{def}}{=} \{t\} \\ \mathcal{P}_i(t) \stackrel{\text{def}}{=} \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \bigcup \mathcal{P}_{i-1}(t) \end{cases} \quad (36)$$

[1] We do not use the Bini's notation ΔC_k^{max} to avoid possible confusion with the critical scaling factor Δ. We use δC_k^{max} instead.

τ_2		τ_3	
t	δC_2	t	δC_3
137	22	137	10
139	-5	168	32

Table 4. Trace of the δC_i

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	118	108
τ_3	248	168	1	32	160

Table 5. Sensitivity analysis before normalization step

Applying Equation 36 to τ_2 and τ_3 to have their scheduling points give us the following sets:

$$\text{sched}(\text{P}_2) = \{T_1, D_2\} \quad (37)$$
$$\text{sched}(\text{P}_3) = \{T_1, D_3\} \quad (38)$$

So, we can now compute the critical scheduling factor for task τ_2 and τ_3 (a trace of the computations can be found in Table 4):

$$\delta C_2 = \max(22, -5) = 22 \quad (39)$$
$$\delta C_3 = \max(10, 32) = 32 \quad (40)$$

Having these δC_i, we can now compute the critical scaling factor per criticity level:

$$\delta C_2^{\max}(1) = \min_{i=1,\ldots,n \wedge L_i=1}(\delta C_i) = \min(\{\delta C_3\}) \quad (41)$$
$$\delta C_2^{\max}(2) = \min_{i=1,\ldots,n \wedge L_i=2}(\delta C_i) = \min(\{\delta C_2\}) \quad (42)$$

If we apply the modification to the task system, we obtain the system shows in Table 5. We can easily see that the basic hypothesis of multiple criticality task system (Equation 1) is not satisfied for task τ_2 since $C_2(1) > C_2(2)$. So, we have to perform a normalization step, as describe in the previous section.

After normalization, we obtain the task system describes in Table 6. Figure 4 shows the multiple criticality $\mathbb{C}$-space for the task τ_2, that is to say the possible value for $C_2(1)$ and $C_2(2)$ in order to satisfy Equation 2.

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	108	108
τ_3	248	168	1	32	160

Table 6. Sensitivity analysis after normalization step

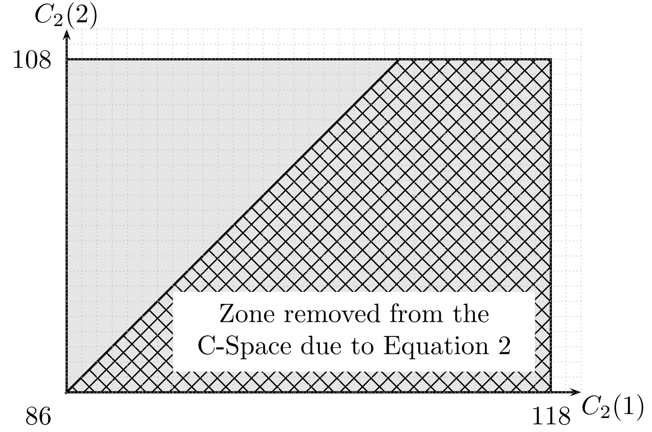


Figure 4. Multiple criticality $\mathbb{C}$-space for task τ_2

6. Conclusion and future work

In this article, we investigate the multiple criticality task scheduling model introduced in [13] and [4]. Such task model represents a potentially very significant advance in the modeling of safety-critical real-time systems. We first formally proved the original Audsley's algorithm is already optimal in the class of fixed-priority algorithm for scheduling independent task systems with constrained-deadlines, and as a consequence that the tie braking rule used in Vestal's algorithm is not useful for assigning fixed-priority to multi criticality tasks.

Moreover, we performed two kind of sensitivity analysis: we first showed that Vestal's algorithm can be extended to compute the minimum processor speed so that a multiple criticality task set is schedulable. For that purpose, Lehoczky's critical scaling factor is used as a tie breaker at each task priority level.

We also show how to adapt the sensitivity analysis in the $\mathbb{C}$-space originally developed by Bini in [6] for the case of multiple criticality task systems. Such an extension allows to analyse a subset of tasks. From a practical point of view, it is particularly useful to analyse all tasks belonging to the specific critical level.

Future work Lehoczky, in [9] performed a sensitivity analysis for a single task and the whole task system. Bini et al., in [6] extends this method to allow a task sensitivity analysis according to a given direction. Future works may concern the sensitivity analysis of a task system to draw the $\mathbb{C}$-space without considering any particular direction.

References

[1] ARINC. Avionics application software standard interface. ARINC Spec, 653, 1997.

[2] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Real-Time Systems, 1991.

[3] F. Authority. Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc: EUROCAE, 1992.

[4] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems, pages 147–155, Washington, DC, USA, 2008. IEEE Computer Society.

[5] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. Computers, IEEE Transactions on, 53(11):1462–1473, Nov. 2004.

[6] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. Real-Time Systems, 39(1-3):5 – 30, 2008.

[7] L. Bougueroua, L. George, and S. Midonnet. Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled fp and edf. In The Second International Conference on Systems (ICONS'07), April 2007.

[8] M. Joseph and P. Pandya. Finding response times in a real-time system. The Computer Journal, 29(5):390–395, 1986.

[9] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. Proceedings of the 11th Real-Time Systems Symposium, pages 201–209, Dec 1990.

[10] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm-exact characterization and average case behavior. In in Proc. ZEEE Real-Time Svst. - SvmD, 1989.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 20(1):46–61, 1973.

[12] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In Proceedings IEEE Real-Time Systems Symposium, pages 181–191. IEEE Computer Society Press, 1986.

[13] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

Quantifying the Sub-optimality of Uniprocessor Fixed Priority Pre-emptive Scheduling for Sporadic Tasksets with Arbitrary Deadlines

Robert I. Davis (✉)
Real-Time Systems Research Group,
Department of Computer Science,
University of York, York, UK.
rob.davis@cs.york.ac.uk

Thomas Rothvoß
Ecole Polytechnique Federale de Lausanne,
Institute of Mathematics, Station 8 - Bâtiment
MA, CH-1015 Lausanne, Switzerland.
thomas.rothvoss@epfl.ch

Sanjoy K. Baruah
Department of Computer Science,
University of North Carolina,
Chapel Hill, NC 27599-317,
Carolina, USA.
baruah@cs.unc.edu

Alan Burns
Real-Time Systems Research Group,
Department of Computer Science,
University of York, York, UK.
alan.burns@cs.york.ac.uk

Abstract

This paper examines the relative effectiveness of fixed priority pre-emptive scheduling in a uniprocessor system, compared to an optimal algorithm such as Earliest Deadline First (EDF). The quantitative metric used in this comparison is the processor speedup factor, defined as the factor by which processor speed needs to increase to ensure that any taskset that is schedulable according to an optimal scheduling algorithm can be scheduled using fixed priority pre-emptive scheduling. For implicit-deadline tasksets, the speedup factor is $1/\ln(2) \approx 1.44270$. *For constrained-deadline tasksets, the speedup factor is* $1/\Omega \approx 1.76322$. *In this paper, we show that for arbitrary-deadline tasksets, the speedup factor is lower bounded by* $1/\Omega \approx 1.76322$ *and upper bounded by 2. Further, when deadline monotonic priority assignment is used, we show that the speedup factor is exactly 2.*

1. Introduction

In this paper, we are interested in determining the largest factor by which the processing speed of a uniprocessor would need to be increased, such that any feasible taskset (that was previously schedulable according to an optimal scheduling algorithm) could be guaranteed to be schedulable according to fixed priority pre-emptive scheduling. We refer to this resource augmentation factor as the *processor speedup factor* [14].

In 1973, Liu and Layland [18] considered fixed priority pre-emptive scheduling of synchronous[1] tasksets comprising independent periodic tasks, with bounded execution times, and deadlines equal to their periods. We refer to such tasksets as *implicit-deadline* tasksets. Liu and Layland showed that *rate monotonic* priority ordering (RMPO) is the optimal fixed priority assignment policy for implicit-deadline tasksets, and that using rate monotonic priority ordering, fixed priority pre-emptive scheduling can schedule any implicit-deadline taskset with a total utilisation $U \le \ln(2) \approx 0.693$.

Liu and Layland also showed that Earliest Deadline First (EDF) is an optimal dynamic priority scheduling algorithm for implicit-deadline tasksets, and that EDF can schedule any such taskset with a total utilisation $U \le 1$.

In 1974, Dertouzos [11] showed that EDF is in fact an optimal pre-emptive uniprocessor scheduling algorithm, in the sense that if a valid schedule exists for a taskset, then the schedule produced by EDF will also meet all deadlines.

Combining the result of Dertouzos [11] with the results of Liu and Layland [18] for both EDF and fixed priority pre-emptive scheduling, we can see that the processor speedup factor required to guarantee that fixed priority pre-emptive scheduling can schedule any feasible implicit-deadline taskset is $1/\ln(2) \approx 1.44270$.

Research into real-time scheduling during the 1980's and early 1990's focussed on lifting many of the restrictions of the Liu and Layland task model. Task arrivals were permitted to be sporadic, with known

[1] A taskset is synchronous if all of its tasks share a common release time.

minimal inter-arrival times, (still referred to as periods), and task deadlines were permitted to be less than or equal to their periods (so called *constrained deadlines*) or less than, equal to, or greater than their periods (so called *arbitrary deadlines*).

In 1982, Leung and Whitehead [15] showed that *deadline monotonic*[2] priority ordering (DMPO) is the optimal fixed priority ordering for constrained-deadline tasksets. Exact fixed priority schedulability tests for constrained-deadline tasksets were introduced by Joseph and Pandya in 1986 [13], Lehoczky et al. in 1989 [17], and Audsley et al. in 1993 [1].

In 1990, Lehoczky [16] showed that deadline monotonic priority ordering is not optimal for tasksets with arbitrary deadlines; however, an optimal priority ordering for such tasksets can be determined, in at most $n(n+1)/2$ task schedulability tests, using Audsley's optimal priority assignment algorithm[3] [1].

Exact schedulability tests for tasksets with arbitrary deadlines were developed by Lehoczky [16] in 1990 and Tindell et al. in 1994 [20].

Exact EDF schedulability tests for both constrained and arbitrary-deadline tasksets were introduced by Baruah et al. in 1990 [6], [7].

In 2008, Baruah and Burns [5] showed that the processor speedup factor for constrained-deadline tasksets is lower bounded by 1.5 and upper bounded by 2. In 2009, Davis et al. [10] derived the exact speedup factor for constrained-deadline tasksets; $1/\Omega \approx 1.76322$ (where Ω is the mathematical constant defined by the transcendental equation $\ln(1/\Omega) = \Omega$, hence, $\Omega \approx 0.567143$).

In this paper, we derive the speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets. We are able to give an exact speedup factor when deadline monotonic priority assignment is used, and upper and lower bounds assuming an optimal priority assignment.

It is known that an exact condition for the schedulability of a constrained or arbitrary-deadline taskset under an optimal pre-emptive uniprocessor scheduling algorithm, such as EDF [11], is that a quantity referred to as the processor LOAD (see Section 2.3) does not exceed the capacity of the processor (i.e. $\text{LOAD} \le 1$) [6], [7].

The processor speedup factor derived in this paper shows that every arbitrary-deadline taskset with $\text{LOAD} \le 0.5$ is guaranteed to be schedulable according to fixed priority pre-emptive scheduling using either deadline-monotonic priority assignment or an optimal priority assignment algorithm.

This result complements the earlier results of Davis et al. [10] that every constrained-deadline taskset with $\text{LOAD} \le \Omega \approx 0.567143$ is guaranteed to be schedulable according to fixed priority pre-emptive scheduling using deadline-monotonic priority assignment; and the seminal result of Liu and Layland [18] ($U \le \ln(2) \approx 0.693$), that applies to implicit-deadline tasksets.

While the results presented in this paper are mainly theoretical, they also have practical utility in enabling system designers to quantify the maximum penalty for using fixed priority pre-emptive scheduling in terms of the additional processing capacity required. This performance penalty can then be weighed against other factors such as implementation overheads when considering which scheduling algorithm to use.

1.1. Related work on average case sub-optimality

This paper examines the sub-optimality of fixed priority pre-emptive scheduling in the worst-case, other research has examined its behaviour in the average-case.

In 1989, Lehoczky et al. [17] introduced the *breakdown utilisation* metric: A taskset is randomly generated, and then all task execution times are scaled until a deadline is just missed. The utilisation of the scaled taskset gives the breakdown utilisation. Lehoczky et al. showed that the average breakdown utilisation, for implicit-deadline tasksets of large cardinality under fixed priority pre-emptive scheduling is approximately 88%, corresponding to a penalty of approximately 12% of processing capacity with respect to an optimal algorithm such as EDF.

In 2005, Bini and Buttazzo [8] showed that breakdown utilisation suffers from a bias which tends to penalise fixed priority scheduling by favouring tasksets where the utilisation of individual tasks is similar. Bini and Buttazzo introduced the *optimality degree* metric, defined as the number of tasksets in a given domain that are schedulable according to some algorithm *A*. divided by the number that are schedulable according to an optimal algorithm. Using this metric, they showed that the penalty for using fixed priority-pre-emptive scheduling for implicit-deadline tasksets is typically significantly lower than that assumed by determining the average breakdown utilisation.

1.2. Organisation

The remainder of this paper is organised as follows. Section 2 describes the system model and notation used, and recapitulates exact schedulability analysis for both fixed priority and EDF scheduling. Section 3 illustrates the processor speedup factor via a simple example. Section 4

---

[2] *Deadline monotonic* priority ordering assigns priorities in order of task deadlines, such that the task with the shortest deadline is given the highest priority.

[3] This algorithm is optimal in the sense that it finds a schedulable priority ordering whenever such an ordering exists.

derives the processor speedup factor required for arbitrary-deadline tasksets under fixed priority pre-emptive scheduling. Section 5 concludes with a summary of the results.

2. Scheduling model and schedulability analysis

In this section, we outline the scheduling model, notation and terminology used in the rest of the paper. We then recapitulate the exact schedulability analysis for both fixed priority pre-emptive scheduling and EDF scheduling.

2.1. Scheduling model, terminology and notation

In this paper, we consider the pre-emptive scheduling of a set of tasks (or *taskset*) on a uniprocessor.

Each taskset comprises a static set of *n* tasks ($\tau_1..\tau_n$), where *n* is a positive integer. We assume that the index *i* of task τ_i also represents the task priority used in fixed priority pre-emptive scheduling, hence τ_1 has the highest fixed-priority, and τ_n the lowest.

Each task τ_i is characterised by its bounded worst-case execution time C_i, minimum inter-arrival time or period T_i, and relative deadline D_i. Each task τ_i therefore gives rise to a potentially infinite sequence of invocations, each of which has an execution time upper bounded by C_i, an arrival time at least T_i after the arrival of its previous invocation, and an absolute deadline D_i time units after its arrival.

In an *implicit-deadline* taskset, all tasks have $D_i = T_i$. In a *constrained-deadline* taskset, all tasks have $D_i \le T_i$, while in an *arbitrary-deadline* taskset, task deadlines are independent of their periods, thus each task may have a deadline that is less than, equal to, or greater than, its period. The set of arbitrary-deadline tasksets is therefore a superset of the set of constrained-deadline tasksets, which is itself a superset of the set of implicit deadline tasksets.

The *utilisation* U_i, of a task is given by its execution time divided by its period ($U_i = C_i / T_i$). The total utilisation *U*, of a taskset is the sum of the utilisations of all of its tasks:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad (1)$$

The following assumptions are made about the behaviour of the tasks:

- The arrival times of the tasks are independent and hence the tasks may share a common release time.
- Each task is released (i.e. becomes ready to execute) as soon as it arrives.
- The tasks are independent and so cannot block each other from executing by accessing mutually exclusive shared resources, with the exception of the processor.
- The tasks do not voluntarily suspend themselves.

A task is said to be *ready* if it has outstanding computation and so is awaiting execution by the processor.

A taskset is said to be *schedulable* with respect to some scheduling algorithm and some system, if all possible sequences of task invocations (or jobs) that may be generated by the taskset can be scheduled on the system by the scheduling algorithm without any deadlines being missed.

Under Earliest Deadline First (EDF) scheduling, at any given time, the ready task invocation with the earliest absolute deadline is executed by the processor. In contrast, under fixed priority pre-emptive scheduling, at any given time, the highest priority ready task is executed by the processor.

When a taskset is scheduled according to fixed priorities, task priorities need to be assigned according to some algorithm. Optimal priority assignment algorithms are known for implicit-deadline [18], constrained-deadline [15], and arbitrary-deadline [1] tasksets.

A priority assignment policy *P* is said to be *optimal* with respect to some class of tasksets if there are no tasksets in the class that are schedulable according to fixed priority pre-emptive scheduling using any other priority ordering policy that are not also schedulable using the priority assignment determined by policy *P*.

A taskset is said to be *feasible* with respect to a given system model if there exists some scheduling algorithm that can schedule all possible sequences of task activations that may be generated by the taskset on that system without missing any deadlines. Note, in this paper, we are primarily interested in a reference system model that consists of a pre-emptive uniprocessor with unit processing speed.

A scheduling algorithm is said to be *optimal* with respect to a system model and a tasking model if it can schedule all of the tasksets that comply with the tasking model and are feasible on the system.

We note that EDF is known to be an optimal pre-emptive uniprocessor scheduling algorithm for tasksets compliant with the tasking model described in this section [11]. Least Laxity First is another such optimal algorithm [19].

A schedulability test is termed *sufficient*, with respect to a scheduling algorithm and system model, if all of the tasksets that are deemed schedulable according to the test are in fact schedulable on the system under the scheduling algorithm. Similarly, a schedulability test is termed *necessary*, if all of the tasksets that are deemed unschedulable according to the test are in fact unschedulable on the system under the scheduling

algorithm. A schedulability test that is both sufficient and necessary is referred to as *exact*.

2.2. Schedulability analysis for fixed priority pre-emptive scheduling

In this section, we give a brief summary of Response Time Analysis [2] used to provide an exact schedulability test for fixed priority pre-emptive scheduling of constrained-deadline tasksets. We then recapitulate on response time analysis for arbitrary-deadline tasksets.

First, we introduce the concepts of worst-case response time, synchronous arrival sequence, and busy periods, which are fundamental to response time analysis.

For a given taskset scheduled under fixed priority pre-emptive scheduling, the *worst-case response time* R_i of task τ_i is given by the longest possible time from release of the task until it completes execution. Thus task τ_i is schedulable if and only if $R_i \le D_i$, and the taskset is schedulable if and only if $\forall i \quad R_i \le D_i$.

A *synchronous arrival sequence* refers to a pattern of arrival such that all tasks arrive simultaneously, and then subsequently as early as possible given the constraints on minimum inter-arrival times.

The term *priority level-i busy period* refers to a period of time $[t_1, t_2)$ during which the processor is busy executing computation at priority i or higher, that was released at the start of the busy period at t_1, or during the busy period but strictly before its end at t_2.

The synchronous arrival sequence generates the longest possible priority level-*i* busy period. For constrained-deadline tasksets, the length w_i of this busy period corresponds directly to the worst-case response time of task τ_i. In the remainder of this paper, when we refer to a priority level-*i* busy period, we mean the longest such busy period. Further, when it is clear which priority level is referred to we use the more concise term, busy period.

The busy period comprises two components, the execution time of the task itself, and so called *interference*, equal to the time for which task τ_i is prevented from executing by higher priority tasks.

For constrained-deadline tasksets, the length of the busy period w_i, can be computed using the following fixed point iteration [2], with the summation term giving the interference due to the set of higher priority tasks *hp*(*i*).

$$w_i^{m+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \tag{2}$$

Iteration starts with an initial value w_i^0, typically $w_i^0 = C_i$, and ends when either $w_i^{m+1} = w_i^m$ in which case the worst-case response time R_i, is given by w_i^{m+1}, or when $w_i^{m+1} > D_i$ in which case the task is unschedulable. The fixed point iteration is guaranteed to converge provided that the overall taskset utilisation is less than or equal to 1.

Equation (2) gives an exact schedulability test for the fixed priority pre-emptive scheduling of constrained-deadline tasksets with any fixed priority ordering.

For arbitrary-deadline tasksets, execution of one invocation of a task may not necessarily be complete before the next invocation is released. Hence a number of invocations of task τ_i may be present within the longest priority level-*i* busy period, with earlier invocations delaying the execution of later ones. In general it is therefore necessary to compute the response times of all invocations within the busy period in order to determine the worst-case response time [20].

The length of the busy period $w_i(q)$, starting at the simultaneous arrival of all tasks and extending until the completion of the *q*th invocation of τ_i (where $q = 0$ is the first invocation) is given by the fixed point iteration:

$$w_i^{n+1}(q) = (q+1)C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{3}$$

Iteration starts with an initial value $w_i^0(q)$, typically $w_i^0(q) = (q+1)C_i$, and ends when either $w_i^{n+1}(q) = w_i^n(q)$ in which case the worst-case response time $R_i(q)$, of invocation *q*, is given by $w_i^{n+1}(q) - qT_i$ or when $w_i^{n+1}(q) - qT_i > D_i$ in which case invocation q is unschedulable.

Invocation *q* can only impinge upon the execution of subsequent invocations if its completion occurs after their release. Hence, response times need to be calculated for invocations *q*=0,1,2,3… until an invocation *q* is found that completes at or before the earliest possible release of the next invocation *q*+1, i.e. where: $w_i(q) \le (q+1)T_i$. The worst-case response time of task τ_i is then given by:

$$R_i = \max_{\forall q} (w_i(q) - qT_i) \tag{4}$$

Again, the task is schedulable provided that $R_i \le D_i$.

Equations (3) and (4) give an exact schedulability test for the fixed priority pre-emptive scheduling of arbitrary-deadline tasksets with any fixed priority ordering.

The exact schedulability test given by Equations (3) and (4) potentially requires the examination of a large number of invocations of the task of interest.

A simpler sufficient schedulability test for a task τ_i in an arbitrary-deadline taskset can be derived by considering the maximum amount of task execution at priority *i* and higher released within an interval of length D_i starting with simultaneous arrival of all tasks. If all of this execution can be completed by D_i, then this indicates that the length of the longest priority level-*i* busy period is at most D_i, and hence that all invocations of τ_i released in that busy period meet their deadlines, and so τ_i is schedulable. This sufficient schedulability test is given by

Equation (5):

$$\sum_{\forall j=hep(i)} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \le D_i \tag{5}$$

Where *hep*(i) is the set of tasks with priorities higher than or equal to i.

2.3. Exact schedulability analysis for EDF

The schedulability of an arbitrary-deadline taskset under EDF can be determined via the processor demand bound function $h(t)$ given below:

$$h(t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \tag{6}$$

Baruah et al [6], [7] showed that a taskset is schedulable under EDF if and only if a quantity referred to as the processor LOAD is ≤ 1 where the processor LOAD is given by:

$$\text{LOAD} = \max_{\forall t}\left(\frac{h(t)}{t}\right) \tag{7}$$

Further, they showed that the maximum value of $h(t)/t$ occurs for some value of t in the interval $(0, L]$, where L is defined as follows, thus limiting the number of values of t that need to be checked to determine schedulability.

$$L = \max\left(D_1, D_2, ...D_n, \max_{\forall i}\left((T_i - D_i)\frac{U}{1-U}\right)\right) \tag{8}$$

The only values of t that need to be checked in the interval $(0, L]$ are those where the processor LOAD can change, i.e. $\forall i \quad t = kT_i + D_i$ for integer values of k.

Significant developments have been made, extending the scope of the schedulability tests for both fixed priority pre-emptive scheduling and EDF; however, these basic forms are sufficient for the purposes of this paper.

2.4. Definitions

Definition 1: Let Ψ be a taskset that is feasible (i.e. schedulable according to an optimal scheduling algorithm) on a processor of speed 1. Now assume that $f(\Psi)$ is the lowest speed of any similar processor that will schedule taskset Ψ using scheduling algorithm A. The *processor speedup factor* f^A for scheduling algorithm A is given by the maximum processor speed required to schedule any such taskset Ψ.

$$f^A = \max_{\forall \Psi}(f(\Psi))$$

For any scheduling algorithm A, we have $f^A \ge 1$, with smaller values of f^A indicative of a more effective scheduling algorithm, and $f^A = 1$ implying that A is an optimal algorithm.

In the remainder of the paper, unless otherwise stated, when we refer to the processor speedup factor, we mean the processor speedup factor for fixed priority pre-emptive scheduling using an optimal priority assignment policy.

Definition 2: A taskset is said to be *speedup-optimal* if it requires the processor to be speeded up by the processor speedup factor in order to be schedulable under fixed priority pre-emptive scheduling. Hence for a speedup-optimal taskset Ψ, $f(\Psi) = f^A$.

3. Example

The concept of processor speedup factor defined in the previous section can be illustrated by means of an example.

Consider the arbitrary-deadline taskset S comprising the two tasks defined in Table 1. The parameters of these tasks appear to have some unusual values; however, this is because they have been chosen so that the taskset is just schedulable according to EDF, yet requires a speedup factor of 1.8 in order to be schedulable according to fixed priority pre-emptive scheduling, with priorities ordered via deadline monotonic priority assignment.

Table 1

Task	C_i	T_i	D_i
τ_1	1.8	2	16
τ_2	14.4	∞	17

We now show that taskset S is schedulable according to EDF

Under EDF scheduling, the processor demand bound function $h(t)$ for taskset S is the sum of the processor demand bound functions $h(t,\tau_1)$ and $h(t,\tau_2)$ for tasks τ_1 and τ_2 respectively, where $h(t,\tau_i)$ is the processor demand bound at time t for a single task τ_i, given below:

$$h(t,\tau_i) = \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \tag{9}$$

Thus:

$$h(t,\tau_1) = \begin{Bmatrix} 0 & t < 16 \\ \left\lfloor \frac{t-16}{2} \right\rfloor 1.8 + 1.8 & t \ge 16 \end{Bmatrix} \tag{10}$$

as $\lfloor x/y \rfloor \le x/y$, we have:

$$h(t,\tau_1) \le \begin{Bmatrix} 0 & t < 16 \\ \frac{1.8(t-16)}{2} + 1.8 & t \ge 16 \end{Bmatrix} \tag{11}$$

Similarly, the processor demand bound function for task τ_2 is:

$$h(t,\tau_2) = \begin{Bmatrix} 0 & t < 17 \\ 14.4 & t \ge 17 \end{Bmatrix} \quad (12)$$

Recall that any arbitrary-deadline taskset is schedulable according to EDF, provided that:

$$\text{LOAD} = \max_{\forall t}\left(\frac{h(t)}{t}\right) \le 1 \quad (13)$$

Now, given the following:

(i) The value of $h(t)/t$ at times $t = 16$, $t = 17$, and $t = 18$ are 1.8, 16.2 and 18 respectively.

(ii) From Equations (11) and (12), an upper bound on the value of $h(t)/t$ at time $t = 18$ is 18.

(iii) From Equation (11), the rate of increase of the upper bound on $h(t)/t$ for $t \ge 18$ is 0.9.

Hence, the maximum value of $h(t)/t$ occurs at time $t = 18$. The processor LOAD of taskset S is therefore 1, indicating that the taskset is just schedulable according to EDF.

We now consider the schedulability of taskset S when scheduled according to fixed priority pre-emptive scheduling, using deadline monotonic priority assignment, on a processor that has been speeded up by a factor of 1.8. The parameters of the taskset on this faster processor are given in Table 2. We refer to this taskset as V.

Table 2

Task	C_i	T_i	D_i
τ_1	1	2	16
τ_2	8	∞	17

Figure 1 illustrates the execution of taskset V under fixed priority pre-emptive scheduling, assuming a synchronous arrival sequence.

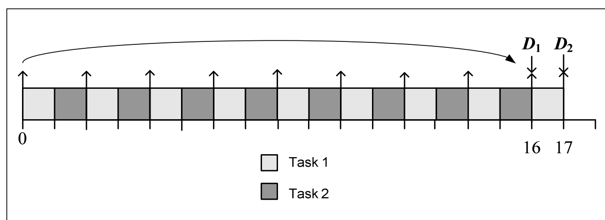


Figure 1

We note that the worst-case response time of task τ_1 is 1 and that of task τ_2 is 16. Taskset V is only just schedulable under fixed priority pre-emptive scheduling, using deadline monotonic priority assignment. Any reduction in processor speed would result in the taskset being unschedulable. The processor speedup factor required is therefore 1.8.

4. Processor speedup factor for arbitrary-deadline tasksets

In this section, we derive the exact processor speedup factor required for the (non-optimal) case where deadline monotonic priority ordering is used in conjunction with arbitrary-deadline tasksets. Further, we provide upper and lower bounds on the processor speedup factor required for the general case where an optimal priority assignment algorithm [1] is used to determine task priorities.

4.1. Arbitrary-deadline tasksets with deadline Monotonic priority ordering

Initially, we consider the case of arbitrary-deadline tasksets where task priorities are assigned in deadline monotonic priority order (DMPO). Recall that DMPO is not optimal in this case [16]; nevertheless, fixed priority pre-emptive scheduling using DMPO is a simple combination of scheduling algorithm and priority assignment policy that is used in many real-time systems. We now derive an exact processor speedup factor for this combination.

Lemma 1: An upper bound on the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using deadline monotonic priority assignment is 2.

Proof: Let S be any taskset that is schedulable on a processor of unit speed according to an optimal scheduling policy such as EDF.

For each task τ_k, in S, consider the processor demand bound during an interval of length $2D_k$. As taskset S is schedulable according to EDF, it follows that:

$$s\sum_{i=1}^{n}\max\left(0,\left\lfloor\frac{2D_k - D_i}{T_i}\right\rfloor + 1\right)C_i \le 2D_k \quad (14)$$

Where $s = 1$ is the speed of the processor.

Next, consider taskset S scheduled according to fixed priority pre-emptive scheduling on a processor of speed $s = 2$ using deadline monotonic priority assignment. DMPO implies that $\forall i \le k \quad D_i \le D_k$.

From Equation (14) above, assuming speed $s = 2$, and discarding the contribution from all tasks of lower priority than k we have:

$$\sum_{i=1}^{k}\max\left(0,\left\lfloor\frac{2D_k - D_i}{T_i}\right\rfloor + 1\right)C_i \le D_k \quad (15)$$

As $\lfloor x \rfloor + 1 \ge \lceil x \rceil$ and $\forall i \le k \quad D_i \le D_k$ then:

$$\sum_{i=1}^{k}\left\lceil\frac{D_k}{T_i}\right\rceil C_i \le D_k \quad (16)$$

Equation (16) is recognisable as the sufficient schedulability test for task τ_k in an arbitrary-deadline taskset *S*, scheduled under fixed priority pre-emptive scheduling (see Equation (4) in Section 2.2). Repeating the above argument for each task τ_k in *S* proves that the taskset is schedulable on a processor of speed 2 under fixed priority pre-emptive scheduling using deadline monotonic priority assignment □

Theorem 1: An exact bound on the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using deadline monotonic priority ordering is 2.

Proof: Consider taskset *V* with the following parameters on a processor of speed f :

τ_1 : $C_1 = 1/2k$, $T_1 = 1/k$, $D_1 = 1$

τ_2 : $C_2 = 1/2$, $T_2 = \infty$, $D_2 = 1 + 1/2k$

where *k* is an integer, and task τ_1 has a higher priority than task τ_2 i.e. deadline monotonic priority ordering. The execution of taskset *V* under fixed priority pre-emptive scheduling is illustrated in Figure 2. (Note the similarity to the taskset used as an example in Section 3).

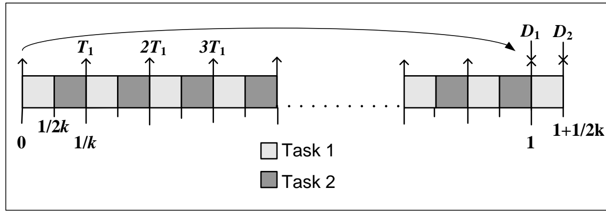


Figure 2

We observe that with fixed priority pre-emptive scheduling, any increase in the execution time of either task will cause task τ_2 to miss its first deadline following simultaneous release of the two tasks.

We now consider the execution of taskset *V* under EDF on a processor of unit speed. Let taskset *S* be formed from taskset *V* by increasing the execution times of tasks τ_1 and τ_2 by a scaling factor f to form tasks τ_1' and τ_2', thus accounting for the reduction in processor speed.

We observe that $f = 2$ is an upper bound on the maximum scaling factor that could possibly result in a schedulable taskset under EDF as this scaling factor results in task τ_1' having a utilisation of 100%.

Under EDF scheduling, the processor demand bound function $h(t)$ for taskset *S* is the sum of the processor demand bound functions $h(t, \tau_1')$ and $h(t, \tau_2')$ for tasks τ_1' and τ_2' respectively.

$$h(t, \tau_1') = \left\{ \begin{matrix} 0 & t < 1 \\ \left\lfloor \frac{t - 1 + (1/k)}{(1/k)} \right\rfloor \frac{f}{2k} & t \ge 1 \end{matrix} \right\} \quad (17)$$

as $\lfloor x/y \rfloor \le x/y$, we have the following upper bound:

$$h(t, \tau_1') \le \left\{ \begin{matrix} 0 & t < 1 \\ \frac{f(t-1)}{2} + \frac{f}{2k} & t \ge 1 \end{matrix} \right\} \quad (18)$$

Similarly, the processor demand bound function for task τ_2' is:

$$h(t, \tau_2') = \left\{ \begin{matrix} 0 & t < 1 + (1/2k) \\ f/2 & t \ge 1 + (1/2k) \end{matrix} \right\} \quad (19)$$

Recall that any arbitrary-deadline taskset is schedulable according to EDF, provided that:

$$\text{LOAD} = \max_{\forall t} \left(\frac{h(t)}{t} \right) \le 1 \quad (20)$$

Now, given the following:

(i) The value of $h(t)/t$ at time $t = 1$ is $f/2k$.

(ii) An upper bound, from Equations (18) and (19), on the value of $h(t)/t$ at time $t = 1 + (1/2k)$ is:

$$\frac{h(1+(1/2k))}{(1+(1/2k))} \le \frac{(f/2) + ((1+(1/2k)) - 1)(f/2) + (f/2k)}{(1+(1/2k))}$$
$$= \frac{f(k + (3/2))}{2(k + (1/2))} \quad (21)$$

(iii) The rate of increase of the upper bound on $h(t)/t$ for $t > 1 + (1/2k)$ is $f/2$ (from Equation (18)).

Then for values of $f \le 2$, the maximum value of the upper bound on $h(t)/t$ occurs at time $t = 1 + 1/2k$, therefore:

$$\max_{\forall t} \left(\frac{h(t)}{t} \right) = \frac{f(k + (3/2))}{2(k + (1/2))} \overset{\lim_{k \to \infty}}{=} \frac{f}{2} \quad (22)$$

From Equation (22), the minimum value for the processor LOAD is achieved in the limit as $k \to \infty$, and this value is $f/2$. From Equation (22), for $k = \infty$, taskset *V* is schedulable according to EDF when its task execution times are scaled up by a factor of $f = 2$ to form taskset *S*. Hence taskset *S* requires a processor speedup factor of 2 in order to be schedulable under fixed priority pre-emptive scheduling with deadline monotonic priority ordering. As the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using deadline monotonic priority ordering is also upper bounded by 2 (Lemma 1), the exact processor speedup factor is 2 □

Corollary 1: Taskset *S* defined in the proof of Theorem 1 (with $k = \infty$), is a speedup-optimal taskset for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using deadline monotonic priority ordering.

It is interesting to note that the speedup-optimal taskset (requiring the largest speedup factor), includes a task τ_1 , with a deadline much larger than its infinitesimal period,

and a task τ_2, with a deadline much smaller than its infinite period.

Theorem 2: An upper bound on the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using an optimal priority assignment algorithm is 2.

Proof: Follows directly from the fact that using an optimal priority assignment algorithm, fixed priority pre-emptive scheduling can schedule any taskset that is schedulable using deadline monotonic priority ordering. Hence the processor speedup factor required can be no greater with optimal priority assignment than the exact processor speedup factor given by Theorem 1 for deadline monotonic priority ordering □

Theorem 3: A lower bound on the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets using an optimal priority assignment algorithm is $1/\Omega$ = 1.76322.

Proof: Follows directly from the fact that the set of arbitrary-deadline tasksets is a superset of the set of constrained-deadline tasksets, and the proof given by Davis et al. [10] that the exact speedup factor required for constrained-deadline tasksets is $1/\Omega$ □

5. Summary and conclusions

In this paper, we have examined the relative effectiveness of fixed priority pre-emptive scheduling for tasksets with arbitrary deadlines. Our metric for measuring the effectiveness of this scheduling algorithm is a resource augmentation factor known as the processor speedup factor.

The processor speedup factor is defined as the minimum amount by which the processor needs to be speeded up so that any taskset that is feasible (i.e. schedulable by an optimal algorithm such as EDF) can be guaranteed to be schedulable under fixed priority pre-emptive scheduling.

Table 3 shows the processor speedup factor needed for fixed priority pre-emptive scheduling given the different taskset classifications (implicit-, constrained-, and arbitrary-deadline) and different priority assignment policies. In Table 3, when a single value is shown for both the upper and lower bounds, this implies that the bounds are the same and the value is exact. (Note the results shown are for tasksets of arbitrary cardinality).

Table 3: Fixed priority pre-emptive scheduling processor speedup factors

Taskset constraints [Priority ordering]	Lower Bound	Upper Bound
Implicit-deadline [Optimal (RMPO)]	$1/\ln(2) =$ **1.44269**	
Constrained-deadline [Optimal (DMPO)]	$1/\Omega =$ **1.76322**	
Arbitrary-deadline [Not optimal (DMPO)]	**2**	
Arbitrary-deadline [Optimal algorithm]	$1/\Omega =$ **1.76322**	**2**

In conclusion, the major contributions of this paper are as follows:

- Proving that the exact processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets with priorities assigned according to deadline monotonic priority assignment is 2.
- Proving that the processor speedup factor for fixed priority pre-emptive scheduling of arbitrary-deadline tasksets with priorities assigned according to Audsley's optimal priority assignment algorithm, is upper bounded by 2 and lower bounded by $1/\Omega$ = 1.76322.

The seminal work of Liu and Layland [18] characterises the maximum performance penalty incurred when an implicit-deadline taskset is scheduled using rate-monotonic, fixed priority pre-emptive scheduling instead of an optimal algorithm such as EDF.

The research in this paper provides an analogous characterisation of the maximum performance penalty incurred when arbitrary-deadline tasksets are scheduled using fixed priority pre-emptive scheduling instead of an optimal algorithm such as EDF. Table 4 summarises the maximum extent of these performance penalties, when deadline monotonic priority assignment is used.

Table 4: Sub-optimality of fixed priority pre-emptive scheduling using deadline monotonic priority assignment

	Optimal (e.g. EDF)	Fixed Priority (DMPO)	Speedup factor
Implicit-deadline	$U \le 1$	$U \le \ln(2)$ ≈ 0.693147	$1/\ln(2)$ ≈ 1.44270
Constrained-deadline	$\text{LOAD} \le 1$	$\text{LOAD} \le \Omega$ ≈ 0.567143	$1/\Omega$ ≈ 1.76323
Arbitrary-deadline	$\text{LOAD} \le 1$	$\text{LOAD} \le 0.5$	2

Note that although in this paper, we have made numerous references to EDF as an example of an optimal pre-emptive uniprocessor scheduling algorithm, and made use of results about EDF in our proofs, our results are valid with respect to *any* optimal pre-emptive uniprocessor scheduling algorithm, for example Least Laxity First [19]. This is because all such optimal algorithms can by definition schedule exactly the same set of tasksets: all those that are feasible.

In conclusion, this paper provides for the first time, bounds on the sub-optimality of fixed priority pre-emptive scheduling for uniprocessor systems with arbitrary-deadlines

Future work

Although this paper provides upper and lower bounds, the exact sub-optimality of fixed priority pre-emptive scheduling with respect to *arbitrary-deadline* tasksets assuming optimal priority assignment remains an open question. To the best of our knowledge, no research has yet been done to characterise the average-case sub-optimality of fixed priority pre-emptive scheduling for arbitrary-deadline tasksets. This is also an interesting area for future research.

Acknowledgements

This work was funded in part by the EU FP7 projects Jeopard (project number 216682) and eMuCo (project number 216378).

References

[1] Audsley N.C., "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times", *Technical Report YCS 164*, Dept. Computer Science, University of York, UK, 1991.

[2] Audsley N.C., Burns A., Richardson M., Wellings A.J., “Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling”. *Software Engineering Journal*, 8(5), pages 284-292, 1993.

[3] Baker T.P., “Stack-based Scheduling of Real-Time Processes.” *Real-Time Systems Journal* (3)1, pages 67-100. 1991.

[4] Baruah S., Burns A. “Sustainable Scheduling Analysis”. *In Proceedings of the IEEE Real-Time Systems Symposium*, pages 159-168, 2006.

[5] Baruah S., Burns A., “Quantifying the sub-optimality of uniprocessor fixed priority scheduling.” *In Proceedings of the IEEE International conference on Real-Time and Network Systems*, pages 89-95, 2008.

[6] Baruah S.K., Mok A.K., Rosier L.E., “Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor”. *In Proceedings of the IEEE Real-Time System Symposium*, pages182-190, 1990.

[7] Baruah S.K., Rosier L.E., Howell R.R., “Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on one Processor”. *Real-Time Systems*, 2(4), pages 301-324, 1990.

[8] Bini E., Buttazzo G.C., “Measuring the Performance of Schedulability Tests”, *Real-Time Systems* 30 (1-2), pages 129-154, 2005.

[9] Bini E., Buttazzo G.C., Buttazzo G.M., “Rate Monotonic Scheduling: The Hyperbolic Bound”. *IEEE Transactions on Computers*, 52(7), pages 933–942, 2003.

[10] Davis R.I., Rothvoß T., Baruah S.K., Burns A., “Exact Quantification of the Sub-optimality of Uniprocessor Fixed Priority Pre-emptive Scheduling.” *Real-Time Systems to appear* 2009.

[11] Dertouzos M.L., “Control Robotics: The Procedural Control of Physical Processes”. *In Proceedings of the IFIP congress*, pages 807-813, 1974.

[12] Fineberg M.S., Serlin O., “Multiprogramming for hybrid computation”. *In Proceedings of AFIPS Fall Joint Computing Conference*, pages 1-13, 1967.

[13] Joseph M., Pandya P.K., “Finding Response Times in a Real-time System”. *The Computer Journal*, 29(5), pages 390–395, 1986.

[14] Kalyanasundaram B., Pruhs K., “Speed is as powerful as clairvoyance”. *In Proceedings of the 36th Symposium on Foundations of Computer Science*, pages 214-221, 1995.

[15] Leung J.Y.-T., Whitehead J., "On the complexity of fixed-priority scheduling of periodic real-time tasks". *Performance Evaluation*, 2(4), pages 237-250, 1982.

[16] Lehoczky J., “Fixed priority scheduling of periodic task sets with arbitrary deadlines”. In *Proceedings 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[17] Lehoczky J.P., Sha L., Ding Y., “The rate monotonic scheduling algorithm: Exact characterization and average case behaviour”. *In Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[18] Liu C.L., Layland J.W., "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of the ACM*, 20(1) pages 46-61, 1973.

[19] Mok A.K., “Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment,” *Ph.D. Thesis*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983.

[20] Tindell K.W., Burns A., Wellings A.J., “An extendible approach for analyzing fixed priority hard real-time tasks”. *Real-Time Systems.* Volume 6, Number 2, pages 133-151, 1994.

[21] Zuhily A., Burns A., “Optimality of (D-J)-monotonic Priority Assignment”. *Information Processing Letters*. Number 103, pages 247-250, 2007.

Timing Analysis

Towards Adaptable Control Flow Segmentation for Measurement-Based Execution Time Analysis *

Michael Zolda Sven Bünte Raimund Kirner

Real Time Systems Group
Vienna University of Technology, Austria
E-mail: {michaelz,sven,raimund}@vmars.tuwien.ac.at

Abstract

During the design of embedded real-time systems, engineers have to consider the temporal behavior of software running on a particular hardware platform. Measurement-based timing analysis is a technique that combines elements from static code analysis with execution time measurements on real physical hardware. Because performing exhaustive measurement is generally not tractable, some kind of abstraction must be used to deal with the combinatoric complexity of real software. We propose an adaptable measurement-based analysis approach that uses the novel flexible abstraction of a segment graph *to model control flow at varying levels of detail. We also present preliminary experimental results produced by a prototype implementation.*

1 Introduction

In real-time systems the term correctness does not only refer to the functional behavior of calculations. Compliance with temporal requirements is an essential part in the design process. If transient violations of timing constraints are tolerated we speak of soft real-time systems. Think of a mobile phone for instance where short communication delays are acceptable. On the other hand, safety-critical hard real-time systems include at least one temporal requirement the violation of which would potentially lead to a catastrophe. An airbag not releasing in time or a non-reacting aircraft control unit for instance can lead to a fatal disaster.

Consequently, there is an inherent demand for verification techniques that focus on the temporal behavior of real-time systems. Usually, a design is composed out of tasks to handle complexity. A schedule ensures that functional as well as temporal dependencies are adhered to. Most of the common schedulability analysis techniques demand the knowledge of a safe upper bound of the *worst-case execution time (WCET)* for each single task. For hard real-time systems those deadlines are strict. However, soft real-time systems can tolerate violations to some degree.

A comprehensive overview of WCET analysis techniques is given in [22]. In summary, determining the WCET is hard due to the inherent complexity and the partly complementary requirements to the analysis:

- **Safety** is the property that the obtained WCET estimate may not underestimate the real WCET
- **Precision** is an indicator of the deviation between the obtained WCET estimate and the real WCET
- **Performance** of the WCET analysis denotes the amount of computational resources needed to perform the analysis
- **Accessibility** of the WCET analysis covers aspects like available granularity of WCET results, the back-annotation of WCET results to the source code, and the necessary effort to perform the WCET analysis on a new target hardware

There are three categories in which the various WCET analysis techniques can be classified. In **end-to-end black box testing** the program is simply executed on a set of input data. The advantage is that the test environment is easy to set up. However, there is no way to state anything about precision or safety. In contrast, **static analysis** examines a software/hardware model of the system under investigation without executing the program. This approach allows for deriving safe and sufficiently precise execution time bounds, which makes it suitable for verifying safety-critical real-time systems [9]. Still, modeling and analyzing the system adequately takes much effort. It can even be impossible if the system behavior is too complex or partly unknown (e.g. the documentation provided by the

*The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project Formal Timing Analysis Suite of Real-Time Systems (FORTASRT) under contract P19230-N13 and the research project 'Sustaining Entire Code-Coverage on Code Optimization (SECCO) under contract P20944-N13.

processor manufacturer may be incomplete w.r.t. temporal aspects). A third category encompasses **measurement-based techniques**, i.e. all approaches that combine execution time measurements and static program analysis. Measurement-based techniques is usually designed with the explicit goal to provide a trade-off between safety, precision on the one hand and performance, accessibility on the other hand.

This article illustrates the overall architecture and segmentation technique for a flexible and easily accessible measurement-based approach that is supposed to give a WCET estimate where the precision depends on how much effort the developer is willing to invest. The overall goal is to make it applicable in several development stages of the system under investigation. We focus on soft real-time systems because we cannot in general avoid underestimation of the actual WCET. However, the method is still appropriate for hard real-time systems in an early stage of development when preliminary results are needed.

The basic concepts of measurement-based timing analysis are discussed in Section 2, which is followed by a discussion of related work on measurement-based WCET analysis. Program segmentation, as described in Section 4, is the key strategy to provide an adjustable coverage metric for the systematic execution time measurements. The details of the algorithm for program segmentation are given in Section 5. In Section 6 we illustrate the setup of experiments on a prototype implementation of the *Formal Timing Analysis Suite (*FORTAS*)*[1], which yields preliminary results regarding the applicability and consequences of adaptable program segmentation for WCET analysis.

2 Measurement-based Timing Analysis

Measurement-based Execution Time Analysis (MBTA) is a hybrid WCET analysis technique, i.e., it combines static program analysis techniques and execution time measurements. As shown in Figure 1, *measurement-based timing analysis* typically consists of the following three phases:

Analysis and Decomposition:
For WCET analysis, the maximal end-to-end execution time of the software is of interest. In general, to obtain a perfectly accurate timing model, we would have to consider the execution time of all possible operation sequences that can be performed by the computer while executing the given computer program. Measuring all these sequences is in general intractable, as there are simply too many of them. Therefore, reducing the number of execution time measurements is crucial for MBTA. One way to do this is to decompose the program behavior into subsets and to ensure coverage on each subset.

The *control flow graph (CFG)* is a common graph-based program representation in compiler construction, where nodes represent the operations of the software and where edges represent possible successive executions[2]. We have chosen to operate on the CFG as a good basis for MBTA, because the execution time depends largely on which specific instructions are executed.

We use the technique of *segmentation* to decompose the CFG of ANSI C programs into smaller subgraphs. Depending on how we decompose the program into segments, we can adjust the trade-off between safety, precision on the one hand, and performance, accessibility on the on the other hand.

Execution Time Measurement:
Once the program is decomposed, the execution time is determined for each *segment*.

We measure the execution time on the real hardware, allowing us to take hardware characteristics into account without modeling them in full detail.

Timing Composition:
Having performed systematic execution time measurements for each segment, the timing results from all segments have to be composed to obtain a WCET estimate.

As said above, due to hardware features like pipelines, caches, or out-of-order executions, without additional precautions our MBTA approach will not provide sufficient state coverage to guarantee safe WCET bounds.

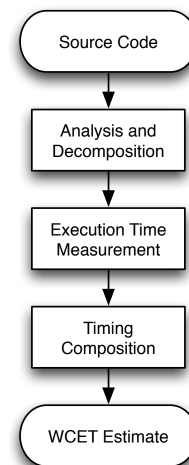


Figure 1: Measurement-based Timing Analysis.

Our design choice of decomposing C programs based on their CFG and learning a hardware model closely related to the CFG has two major ramifications:

Accessibility: the derived timing information can be related to the source code. At least at the granularity of our segmentations it is directly possible to assign timing values to source code regions representing a segment. It is also possible to relate the timing of individual paths within a segment back to the source

[1] http://www.fortastic.net

[2] The traditional definition of a CFG does not allow for linear sequences of operations, i.e., the nodes must constitute so-called *basic blocks*. We relax this requirement and allow for general graphs, as the restriction the basic blocks is neither strictly necessary, nor particularly useful in our context.

code. However, due to compiler optimizations, this can sometimes lead to timing distributions that may not be obvious at the source code level. Here the user would have to investigate the generated code to fully understand the timing results. Overall, our approach provides the software developer with a convenient representation of the timing information.

Furthermore, since we do not make use of a hardware model, any systems can be analyzed as long as the target platform offers the necessary means for measurement.

Safety and Precision: the measurement-based timing analysis framework is generally used to provide WCET estimates of reasonable precision instead of safe WCET bounds. The WCET estimate can be potentially unsafe due to the following reasons:

- *Compiler optimizations* can introduce new control flow paths, which may not be covered by our test data generation based on the program source code. This is a given fact with today's compilers which can only be avoided by deactivating code optimizations. However, we are also working on a more intelligent approach where the goal is to let the compiler activate only those code optimizations that do not threaten the preservation of a selected code coverage [13]. The advantage of this approach is that it will be relatively easy to integrate it into existing compilers.
- *State coverage of hardware components* is usually very hard to achieve by measurement-based timing analysis methods [14]. Thus, on hardware where the instruction timing depends on the current state of the processor, the WCET estimate provided by our method may miss the worst-case initial hardware state for an execution-time measurement. A workaround to this problem would be the explicit enforcement of a predictable state at well-known program points [20]. Measurement-based WCET analysis can potentially outperform static WCET analysis in precision. However, due to the statistical operation of measurement-based WCET analysis, this cannot be guaranteed.

The discussions so far leads to the following main requirement for our measurement-based WCET analysis:

The degree of precision and safety of the analysis has to be ***adaptable*** *to the resources (e.g. analysis time) the developer is willing to invest. All involved means have to be conveniently accessible and capable of being integrated smoothly into a design process at multiple stages.*

One way to achieve this goal is to make use of techniques where the level of abstraction is adaptable. We use *program segmentation* for splitting the CFG into overlapping subgraphs called segments. Each segment is small enough to be measured exhaustively w.r.t. path coverage (also referred to as predicate coverage [16]).

The implicit premise of path coverage is that there is only a finite number of paths that need to be considered. Practically, this amounts to ruling out infinite loops and infinite recursion, which is a reasonable assumption for the kind software components we consider. Concretely we assume each task to be analyzed to be a so-called transformative system, i.e., a subsystem that takes its input data and transforms them into output data [2].

Following common practice, we assume the availability of iteration bounds for all cycles in the CFG. In many cases, such bounds can be derived automatically via static analysis techniques [7, 6]. Otherwise they must be provided by a human expert.

The input data for the measurements is produced by FSHELL [10, 11], a database engine dispatching queries about a C program to program analysis tools. The version at hand utilizes the bounded model checker CBMC [5], which supports full ANSI-C, including function-pointers, bit-operations, and floating-point arithmetic. As a result, FSHELL is able to cope with full ANSI-C, but—due to the nature of bounded model checking—requires loop bounds to be given for all loops or recursive calls with non-constant bounds.

For deriving a global WCET estimate we use the *Implicit Path Enumeration Technique (IPET)* [17, 15], for which the segment graph forms the input, i.e., the linear equations model the flow between segments and the cost for a segment is the worst case observed execution time thereof.

As we will show, the segment size inherently affects analysis complexity and precision and must therefore be adaptable to satisfy our main requirement of having the precision and safety adaptable. Moreover, we will see that segments can be formed out of any graph-like structure such that hardware effects can potentially be incorporated. This enables our measurement-based WCET analysis to increase the level of precision and safety.

3 Related Work

A means for control flow segmentation is discussed in [12] where the program is decomposed into a hierarchical tree of *Regions*. Each region has a single entry and a single exit (SESE) like our segments. We do not make use of any hierarchical structure, though. Furthermore, the presented algorithm to form the regions does not specifically target the reduction of possible control flow paths.

Bernat et al. [3] and Ernst et al. [8] use program segmentation explicitly to target WCET analysis. Because they do not address the problem of systematic generation of input data and the implicit goal of reducing control flow

paths, both the structures and the segmentation algorithms differ from ours.

Our work is largely motivated by Wenzel et al. [21]. The idea of CFG partitioning to reduce the amount of local paths for exhaustive measurements and the successive compositions of timing information for WCET calculation is first discussed in their work. We extend the segmentation to deal with loops and unstructured code. Furthermore, our approach is more flexible in the sense that we extend the degree of freedom for decomposition.

The idea for an adaptable abstraction by using segments is discussed in [1]. However, while we decompose the CFG, their segmentation splits the IPET equation system for reducing complexity.

4 Segmentation

To obtain a perfectly accurate timing model, we generally have to consider the execution time of all possible operation sequences that can be performed by the computer while executing the given computer program.

In compiler construction, the traditional program representation that makes all statically possible operation sequences explicit is the *control flow graph (CFG)*, a graph where nodes represent the operations of the software, and where edges represent possible successive execution.

There are richer graph-like *system representations* for the set of possible operation sequences, like, e.g., the *kripke structures* used in formal methods, which can encode detailed information on the system state (the CFG merely distinguishes different code locations). Also, it is possible to enrich a CFG with additional state information, e.g., by using preconditions. In this paper we will only consider plain CFGs, but it should be noted that the concepts presented here can be adapted to other graph-like representations on different levels of abstraction.

A CFG does not include information about the dynamics of the software. It therefore overapproximates the *(dynamically) feasible* operation sequences, a subset of all *statically possible* operation sequences.

More precisely, each path through the CFG (from a distinguished start node to a distinguished end node) represents a (statically) possible operation sequence. By considering all these paths, we can conclude about the timing behavior of the complete program, from the timing behavior of the individual paths. For example, if we know the Worst Case Execution Time (WCET) of each path, we could, in principle, derive the WCET of the complete program.

Consider the C source code in Listing 1. We can see that the *false* branch of the second conditional statement cannot be taken, if the *false* branch of the first conditional statement has been taken before. As a consequence, only three of the four statically possible paths through the corresponding CFG (Figure 2) are feasible.

The infeasible path e_7, e_6, e_3, e_2 does not contribute to the timing behavior of the program, because it can never be executed. It should therefore be excluded from timing analysis.

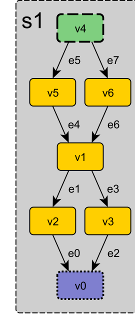


Figure 2: *Maxseg* segment graph induced by the program in Listing 1, an equivalent of the programs CFG. There are four statically possible paths. Assuming that edges 5 and 2 correspond to a successful test of the conditions `x != 0` and `x % 2 == 0`, respectively, the path e_7, e_6, e_3, e_2 is dynamically infeasible.

```
if (x != 0)
  flags = 1;
else
  flags = 0;

if (x % 2 == 0)
  flags = flags | 2;
else
  flags = flags | 4;
```

Listing 1: Source code of a program with two consecutive tests, where the second test can only fail after the first test has succeeded.

The CFG alone cannot represent this information. What we would like to have is a representation that is expressive enough to represent individual paths. However, considering the prohibitively large number of paths in most real software, the representation must also be capable of representing collections of paths concisely. Lastly, our representation should be similar to a CFG, so that timing analysis methods like IPET, which operate on a CFG, can be used with minimal adaption.

Definition 1 (Segment Graph) A *segment graph* Σ *of a CFG* G is a tuple

$$\langle G, S, I, nodes, edges, entry, exit \rangle,$$

where

$$G = \langle N, E, init, final \rangle$$

is a CFG with nodes N, edges $E \subseteq N \times N$, an unique initial node $init \in N$, and an unique final node $final \in N$. Moreover, S is a set of *segment* names, $I \subseteq S \times S$ is a set of *inter edges* (edges between segments), $nodes : S \rightarrow \mathcal{P}(N)$ is a function designating the nodes in each segment, $edges : S \rightarrow \mathcal{P}(E)$ is a function designating the edges in each segment, $entry : S \rightarrow N$ is function designating

the *entry node* of each segment, and $exit : S \rightarrow N$ is a function designating the *exit node* of each segment.

For any inter edge $\langle s, t \rangle$, we require that $\langle exit(s), entry(t) \rangle \in E$.

An *intra edge in a segment* s is an edge $\langle v, w \rangle$ with $\langle v, w \rangle \in edges(s)$.

Each node and each intra edge must be in at least one segment, i.e.,

$$\bigcup_{s \in S} nodes(s) = N \text{ and } \bigcup_{s \in S} edges(s) = E.$$

Furthermore, entry and exit nodes must be in their corresponding segments, i.e.,

$$\{entry(s), exit(s)\} \subseteq nodes(s).$$

Moreover, the source and target nodes of all intra edges must also be in the corresponding segment, i.e.,

$$\langle v, w \rangle \in edges(s) \Rightarrow v \in nodes(s) \wedge w \in nodes(s).$$

An *initial segment* is a segment s with $init \in nodes(s)$. Likewise, a *final segment* is a segment s with $final \in nodes(s)$.

A *segment path* $\pi(s)$ *through a segment* s is a sequence

$$\pi(s) = \langle v_1, v_2 \rangle \langle v_2, v_3 \rangle \ldots \langle v_{n-2}, v_{n-1} \rangle \langle v_{n-1}, v_n \rangle$$

of intra edges $\langle v_i, v_{i+1} \rangle$ that are all in the segment s, i.e., $\langle v_i, v_{i+1} \rangle \in edges(s)$, for some $s \in S$.

Moreover, the path must start in the segment's entry node and end in the segment's exit node, i.e., $v_1 = entry(s)$ and $v_n = exit(s)$.

Figure 3 visualizes a segmentation of the CFG from Figure 2 with three segments. We can seen that segments s_1 and s_3 are initial segments, because they contain the CFG's initial node, whereas segments s_2 and s_3 are final segments, because they contain the CFG's final node. Entry and exit of each segment indicated by dashed or dotted borders, respectively. We can see that nodes and intra edges can be shared between segments. Although not shown in this figure, it is also possible that an inter edge can at the same time be an intra edge for some segments.

Semantically, a segment graph of a CFG G is a description of a subset of the paths in G. A segment graph can therefore be seen as a restriction of a CFG to a certain set of paths.

Definition 2 (Paths in a Segment Graph) Let Σ be a segment graph

$$\langle G, S, nodes, edges, entry, exit \rangle.$$

The *set of paths in* Σ is the set of all CFG paths

$$\pi = \pi_1(s_1)e_1\pi_2(s_2)e_2 \ldots e_{n-1}\pi_n(s_n),$$

where the $\pi_i s_i$ are segment paths that constitute dynamically feasible subpaths in G, and where $e_i = \langle exit(s_i), entry(s_{i+1}) \rangle$, i.e., the segment paths are connected via inter edges.

It can be shown that the set of paths in a segment graph of a CFG G is a subset of the paths in G.

There are two interesting special cases of segment graphs:

minseg: The *minseg* segment graph is the segment graph where each node is contained in its own segment, and where no segment contains any edge, i.e., $nodes(s_v) = \{v\}$, for any $v \in N$, $edges(s_v) = \emptyset$, $entry(s_v) = v$, and $exit(s_v) = v$. The paths in a minseg segment graph of a CFG G are the statically possible paths described by G.

maxseg: There is a single segment s, which contains all nodes and edges of the CFG, i.e., $nodes(s) = N$, $edges(s) = E$, $entry(s) = init$, and $exit(s) = final$. The paths in a *maxseg* segment graph of a CFG G are the dynamically feasible paths of G.

Applying the semantic definitions on the segment graph visualized in Figure 3, we obtain the following segment paths:

$$s1 : \{e_5\}$$
$$s2 : \{e_1e_0,\ e_3e_2\}$$
$$s3 : \{e_7e_6e_1e_0,\ e_7e_6e_3e_2\}.$$

Because the path $e_7e_6e_3e_2$ is a dynamically infeasible path, the set of paths in the segment graph is

$$\{e_5e_1e_0,\ e_5e_3e_2,\ e_7e_6e_1e_0\}.$$

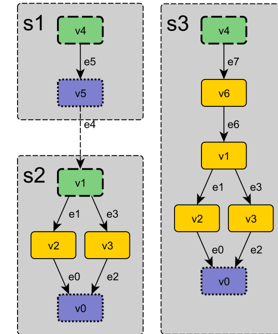


Figure 3: A segment graph that was obtain from the one in Figure 2 by splitting at edge e_4. In this example the segment split into three smaller segments. Segment s_1 now holds all paths that went from the original entry node of the prevision segment to the source node of the split edge, without passing the split edge itself. Segment s_2 contains all paths that went from the target node of the split edge to the exit node, without passing the split edge itself. Segment s_3 contains all paths that went from the entry node to the exit node, without passing the split edge. Edge e_4 has become an inter edge.

The segment graph framework as presented here does not include any special construct for handling function

calls. However, function calls are easily supported via inlining the body of called functions at their respective call sites. This approach allows for unrestricted segments across calling borders. On the other hand, function calls that have not been inlined are handled transparently, as atomic operations. This black-box view can be particularly useful in the case of closed-source third-party code.

5 Segmentation Algorithm

For a given CFG, many different segment graphs are possible, so which one should we choose for our purpose of measurement-based timing analysis? The two corner cases are *minseg* and *maxseg*. The minseg segment graph is only interesting for comparison purposes, as it describes the same set of paths as the plain CFG. On the other hand, performing an analysis on a *maxseg* segment graph would mean that all statically possible paths have to be checked for feasibility and, in case they are found feasible, be subject to measuring and analysis.

In this paper, we consider a segmentation algorithm that is based on the following idea: in order to exclude from the analysis as many infeasible paths as possible, we would like to have segments that are as large as possible in terms of the total number of segment paths. However, the total number of segment paths must not become too large, because we can only check, measure, and analyze a limited number of paths.

Our algorithm starts out with a *maxseg* segment graph and iteratively splits segments into smaller segments until the number of segment paths[3] is small enough in each segment.

Because we have ruled out infinite loops, the number of paths in a segment is always finite and can be calculated as exact or approximate solution of a combinatorial problem that incorporates the given iteration bounds.

Segments are always split at some intra edge and are thereby reduced to up to four smaller segments–details follow below. The new segments are put into a priority queue that is ordered by the number of segment paths. Multiple copies of the same segment are merged into a single segment, as soon as they occur. The algorithm keeps on splitting the largest segment (unless it is already small enough) until the queue is empty. As split edge, the algorithm chooses an edge with a maximal edge betweenness centrality measure.

Edge betweenness is a centrality measure for graphs that indicates the relative importance of an edge as a passageway for shortest paths. It is defined as

$$\Sigma_{v,w \in N} \frac{\sigma_{v,w}(e)}{\sigma_{v,w}}, \qquad (1)$$

where $\sigma_{v,w}$ designates the number of different shortest paths[4] from node v to node w, and where $\sigma_{v,w}(e)$ designates the number of shortest paths from node v to node w that pass through edge e.

The rationale for choosing an edge with maximal edge betweenness for splitting is that cutting such an edge will produce new segments of considerably smaller size than the original segment, i.e., the algorithm will converge to a solution quickly. Moreover, the solution will feature relatively few, but large segments, which can be advantageous during further analysis, e.g., to keep the constraint system in an IPET analysis small.

Edge betweenness can be computed very efficiently. Brandes [4] presents a method for computing betweenness and related shortest-path based centrality measures. The algorithm has an asymptotic worst-case time complexity of $\mathcal{O}(n \cdot m)$, and an asymptotic worst-case space complexity of $\mathcal{O}(n+m)$, where n and m are the number of nodes and edges in the graph, respectively. Our current implementation of segmentation makes use of the BGL [18, 19] implementation of Brandes' algorithm.

Algorithm 1 illustrates our implementation.

Algorithm 1 Pseudo code of the maximum betweenness segmentation algorithm.

```
 1: procedure SEGMENTATE_MAXBET(cfg, limit)
 2:     g ← maxseg segment graph of cfg
 3:     s ← the segment of g
 4:     insert s into priority queue q
 5:     while q is not empty do
 6:         pop segmentation s from q
 7:         if number of paths in s > limit then
 8:             e ← intra edge of s w/max. betweenness
 9:             new_segments ← split s at edge e
10:             replace s with new_segments in g
11:             insert new_segments into q
12:             merge equivalent segments in q
13:         end if
14:     end while
15:     return g
16: end procedure
```

Splitting

Splitting a segment s at an intra edge (v, w) means removing (v, w) from s and turning it into one or more inter edges $e_1, \ldots, e_n$ that connect the segments in the segment graph in such a way that, semantically, no dynamically feasible path is lost.

When the split edge is removed, the previous segment breaks into up to four new segments:

tosplit segment: A segment capturing all paths in the previous segment s from node $entry(s)$ to node v that do not pass through edge (v, w).

fromsplit segment: A segment capturing all paths in the previous segment s from node w to node $exit(s)$ that do not pass through edge (v, w).

[3] An alternative measure is the total number of paths over all segments.

[4] I.e., shortest statically possible CFG paths, in our case.

bypass segment: A segment capturing all paths in the previous segment s from node $entry(s)$ to node $exit(s)$ that do not pass through edge (v, w).

loop segment: A segment capturing all paths in the previous segment s from node w to node v that do not pass through edge (v, w).

Figure 4 shows how these segments are connected by inter edges. Figures 5 and 6 show a concrete example of splitting at a back edge.

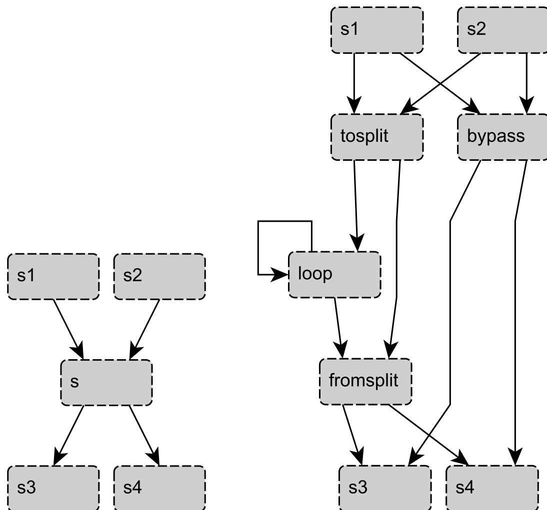


Figure 4: Connection scheme of new segments after splitting. The left hand side shows segment s together with two predecessor segments, s_1 and s_2, and two successor segments, s_3 and s_4. On the right hand side s has been replaced by the new segments $tosplit$, $fromsplit$, $bypass$, and $loop$. These segments have been connected among each other as well as to their environment.

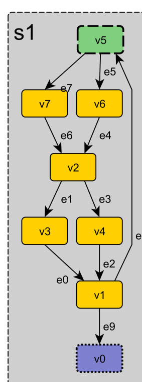


Figure 5: A *maxseg* segment graph of a CFG that contains a loop.

Complexity Considerations

In the course of repeated splitting, it may happen that some of the produced segments are very similar. In particular, our experimental evaluation showed that the plain maximum betweenness segmentation algorithm, as presented above, can produce a large number of identical segments. Mostly, this happens when overlapping segments with a common entry or exit node are found to have their maximum betweenness in a shared edge. Our implementation of the maximum betweenness segmentation algorithm can optionally be configured to merge identical segments after each splitting step, which can reduce the size of the intermediate and final segment graphs significantly.

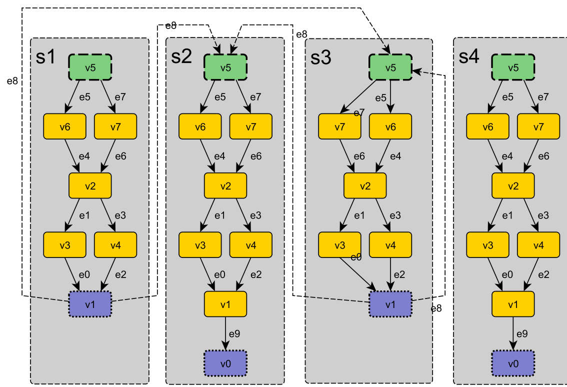


Figure 6: A segment graph that was obtained from the one in Figure 5 by splitting at the back edge e_8. Because the previous segment contained a loop, *loop segment* s_3 was produced. This segment can be reached directly from the *tosplit segment*, and directly reaches the *fromsplit segment*, as well as itself, via a self loop.

During our experimental evaluation, our optimized segmentation algorithm was seen to work fine in practice. For a formal worst case complexity analysis of the algorithm, one would have to consider the combination of two diametrically opposed tendencies. On the one hand, each splitting step will replace a segment with up to four smaller segments. Even though many of these segments are immediately collapsed by the subsequent merging step, this can lead to exponential space complexity in the number of splitting steps. On the other hand, however, the *bypass* segment yielded by splitting is linearly smaller (in terms of paths) than the original segment. Moreover, the size of the *to split*, *from split*, and *loop* segments yielded by splitting at the edge with maximum betweenness is typically a fractional power of the size of the original segment. We have not performed a formal analysis of the overall space and time complexity of our algorithm.

6 Experiments

To highlight the adaptable character of the segmentation techniques we will compare two segmentations of the same input program. They represent two extreme cases where the first includes segments as large as possible (at most 100 paths per segment, i.e. test case generation is barely feasible for the number of contained paths) and the second one forms a single segment for each CFG node (i.e. one path per segment), respectively. This way we illustrate the dependency between analysis complexity and

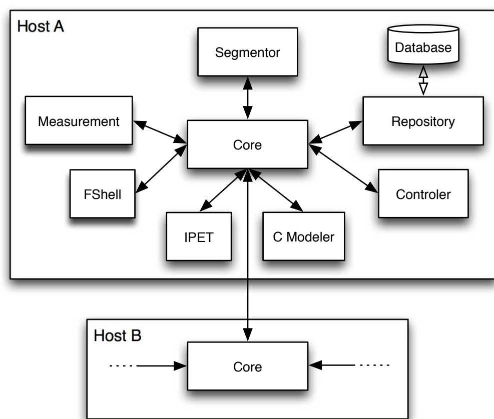


Figure 7: The FORTAS architecture.

the WCET estimate's precision.

We utilize the FORTAS framework to perform our experiments by which means we have access to all basic functions we need. Figure 7 illustrates the FORTAS architecture that combines a collection of modules/plug-ins. The *core* manages the communication between these plug-ins by distributing XML-RPC protocol messages. It potentially allows for running plug-ins on different hosts for instance to parallelize the measurement and the test case generation processes. The presented algorithm is implemented in the *segmentor*. It gets the CFG from *C modeler*, an extension of LLVM and the Clang frontend. The derived segment graphs are added to the *repository* plug-in which provides a consistent and persistent way to access both intermediate and final analysis results. For each segment, we automatically derive a query for FSHELL [10, 11] which in turn generates a test data set yielding path coverage for the according segment. A query to FSHELL is formed by a sequence of program locations such that the generated input data result in an execution sequence including those locations. The input language also includes negations to exclude code locations for a test case and commands to target coverage metrics. By these means, each path in a segment can be expressed by the sequence of its CFG nodes. Query generation is therefore straightforward and convenient.

Once all test data are generated, the consecutive *measurement* process takes the input data set and performs execution time measurements on the target platform. In a next step the longest observed execution time for each segment is filtered out from all measurements. The *IPET* plug-in assembles those values and the segment graph to apply the implicit path enumeration technique, yielding a global WCET estimate. The so far unmentioned *controller* implements the demonstrated work flow as well as means of logging, monitoring and verification. All plug-ins and the core run on a 2.66 GHz Intel Core2 Quad host with 8 GB of RAM.

Target Platform and Measurement

We perform measurements on an Infineon TriCore TC1796 microcontroller. It includes an instruction cache and a processor pipeline which leads to potential underestimations of the global WCET since we do not incorporate execution histories at segment entries on the one hand. We also cannot capture all data-dependent execution time jitter on the other hand, as the CFG is a too coarse abstraction. However, for less complex hardware, e.g. the HCS12 microcontroller, the introduced timing analysis produces a safe WCET bound. We have chosen the TriCore for our measurements, because we plan to tackle the shortcomings of the approach w.r.t. complex hardware in the near future. Furthermore, the TC1796 includes *On-Chip Debug Support (OCDS)* level 2, providing means for cycle-accurate execution tracing. We utilize the Lauterbach LA-7690 Powertrace device to document both timing and flow of control, rendering code instrumentation obsolete: not only measurements have a higher resolution, also the source code can remain unchanged. A measurement starts with test data injection right before the call of the *main* function where all relevant registers (e.g. function arguments) and global variables are set. Note, that the hardware state (cache, pipeline, etc.) is unknown at this point. However, due to a previous initialization script this state is identical for every measurement. The measured execution (or *trace*) then includes not only the execution of *main* but also of all its children in the function call graph. In a post-processing phase the resulting trace is related to the source code, its CFG and segments by the Measurement plug-in using debug information from the binary.

OCDS Level 2 provides traces of temporally high resolution, i.e. every executed machine instruction gets a time stamp. Consequently, the duration of a measured segment path can be derived precisely as the machine instructions can be related to all CFG nodes in the corresponding segment. If measurements are too coarse, not every CFG node of a path through a segment will get a time stamp. This happens for instance when software instrumentation is used to raise hardware signals at certain program points to externally assign timestamps. In this case we choose the splitting edges during segmentation such that they are near to CFG nodes that can be mapped to a time stamp. Consequently, the level of freedom in choosing segment borders is an important feature for guaranteeing portability of our approach.

One problem that comes along with OCDS is that the trace buffer might overflow if too many control flow changing instructions follow in succession. A lack of timing information in the trace influences measurement precision if it occurs at segment borders in which case the first/last available time stamp after/before the gap is taken as a reference for calculating the duration of a segment path. Although this potentially introduces a source of pessimism, we did not observe any trace gaps so far for any benchmark.

Benchmark

The input program on which we carried out the analysis is an engine control unit implemented in ANSI C. The reason for choosing the benchmark is manifold: (a) it rep-

resents a practical application from the automotive industry (provided by Magna Steyer Fahrzeugtechnik), (b) the code is generated by Matlab/Simulink and demonstrates that the analysis can potentially be integrated into a modern design process, (c) with 2952 source code lines and a size of 201430 bytes, it is considerably large and (d) it involves a complex control flow structure (1632 CFG nodes, 2164 transitions) with more than 10^{45} statically possible paths. The target function has one subfunction which is called at most three times per execution. The benchmark includes 230 integer variables that potentially affect control flow. Unfortunately, we cannot make the benchmark publicly available due to a non-disclosure agreement.

1	Paths per Segment	1	≤ 100
2	Number of segments	1287	73
3	Sum of statically possible segment paths	1287	2139
4	Sum of feasible segment paths	1201	1403
5	Analyzed and/or measured paths	387	2139
6	Segmentation time [s]	166	81
7	Test case generation time [s]	6025	177464
8	Measurement [s]	4447	10706
9	IPET time [s]	1423	0.005
10	Overall analysis time [s]	12061	188251
11	Analysis time / path [s]	20	83
12	WCET estimate [μs]	20789	1975
13	WCOET [μs]	728	728
14	Pessimism [%]	2756	171

Table 1: Summarized results.

Preliminary Results

The relation between maximal number of paths per segment, analysis complexity and precision is illustrated in Table 1. We see the results for two segmentations with a maximum of 1 and 100 paths per segment, respectively. The most important effects of these parameters, i.e. analysis complexity and precision are emphasized in rows 10 and 14: the more time is spent the less pessimistic the WCET estimate gets. Here, pessimism is defined as the difference between WCET estimate and the *worst-case observed execution time (WCOET)*, divided by the WCET estimate. There were no manual efforts to maximize the WCOET, i.e. the WCOET is the execution time of maximal observed length. Note, that this metric is only an approximation for this target hardware. However, comparing WCOET and WCET estimate is the best metric available.

The overall analysis time comprises applying the segmentation algorithm (6), test case generation via FSHELL (7), measuring the feasible segment paths (8) and timing composition via IPET (9) to get a WCET estimate. Test case generation uses up most of the analysis time: it has to generate input data or prove infeasibility for each statically possible segment path in each segment.

The difference in analysis time per path is due to an optimization technique. In the experiment with one path per segment, we instructed FSHELL to generate input data yielding basic block coverage for the whole program which implies path coverage for each segment in this special case. This also causes the reduced set of 387 out of 1287 paths that had to be analyzed and measured. In contrast, using a path bound of 100, all 2139 statically feasible paths have to be analyzed individually.

A critical point that we observe is the too pessimistic estimate for a path bound of 1. Our major concern is now to find better segmentation parameters and to improve the overall performance such that useful results can be derived over night.

Potential for Optimization

So far, measurement and test case generation are processed sequentially although they can be pipelined. Table 1 shows that measurements are too time consuming. This is due to a bottleneck in our prototypical measurements device and will be improved in the future.

All measurements are performed end-to-end such that a program execution causes the control flow to pass multiple segments sequentially. However, we do not test whether there is already a measurement in the repository for the segment path that we want to cover. We expect a drastic performance enhancement for this optimization technique.

Another option which is not accounted for so far is to initially apply heuristic test case generation prior to model checking. This technique proved to boost performance considerably in [21].

7 Conclusion and Outlook

In this paper we presented a measurement-based timing analysis approach that incorporates an abstraction technique to express a real-time system's temporal behavior a varying levels of detail. This enables the analysis to be adaptable in terms of complexity, precision and safety.

We have introduced the *segment graph* as a novel, flexible control flow abstraction that can be used to exclude dynamically infeasible paths from further analysis. As a basic operation on a segment graph, we have presented the splitting of individual segments at a given intra edge. This operation forms the foundation of the maximum betweenness segmentation algorithm, which tries to heuristically find a good segmentation.

Although we have only considered CFG-like program representations in this paper, the concept of a segmentation graph is not restricted to this representation. Segmentation graphs can be constructed over other graph-based representations, like, e.g., kripke structures.

Lastly, we have presented preliminary results of experiments that were performed using a prototype implementation of our approach.

Immediate next steps are the further improvement of our prototype implementation which is still in an early stage of development, and performing further experiments. Our more ambitious plans include the develop-

ment of an adaptive analysis approach that employs an incremental refinement strategy to improve the analysis results.

Acknowledgments. We are grateful to Michael Tautschnig and Andreas Holzer for providing FSHELL and the C Modeler. We also thank them for discussions on the topic of this paper.

References

[1] C. Ballabriga and H. Cassé. Improving the WCET Computation Time by IPET Using Control Flow Graph Partitioning. In R. Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[2] K. Berkenkötter and R. Kirner. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, chapter Real-Time and Hybrid Systems Testing, pages 355–387. Springer, July 2005.

[3] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, page 279, Washington, DC, USA, 2002. IEEE Computer Society.

[4] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs . In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] M. de Michiel, A. Bonenfant, H. Cass, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Kaohsiung, Taiwan, 25/08/2008-27/08/2008*, pages 161–168, http://www.computer.org, aot 2008. IEEE Computer Society.

[7] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[8] R. Ernst and W. Ye. Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 598–604, Washington, DC, USA, 1997. IEEE Computer Society.

[9] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. White Paper, AbsInt Angewandte Informatik GmbH, 22nd May 2009.

[10] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 209–213, Princeton, NJ, USA, July 2008. Springer.

[11] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. Query-Driven Program Testing. In N. D. Jones and M. Müller-Olm, editors, *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 151–166, Savannah, GA, USA, January 2009. Springer.

[12] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.*, 29(6):171–185, 1994.

[13] R. Kirner. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems*, 2009, 2009. doi:10.1155/2009/127945.

[14] R. Kirner and P. Puschner. Obstacles in worst-cases execution time analysis. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 333–339, Orlando, Florida, May 2008.

[15] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.

[16] S. C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[17] P. P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times — A Graph-BasedApproach. *Real-Time Syst.*, 13(1):67–91, 1997.

[18] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, December 2001.

[19] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. The Boost Graph Library. `http://www.boost.org/doc/libs/1_39_0/libs/graph/`, Juli 2009.

[20] I. Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[21] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar. 2005.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches

Damien Hardy Isabelle Puaut
Université Européenne de Bretagne / IRISA, Rennes, France

Abstract

Multi-core architectures, which have multiple processors on a single chip, have been adopted by most chip manufacturers. In most such architectures, the different cores have private caches and also shared on-chip caches. For real-time systems to exploit multi-core architectures, it is required to obtain both tight and safe estimations of a number of metrics required to validate the system temporal behaviour in all situations, including the worst-case: tasks worst-case execution times (WCET), preemption delays and migration delays. Estimating such metrics is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches, memory bus, etc.

In this paper, we propose a new method to estimate worst-case cache reload cost due to a task migration between cores. Safe estimations of the so-called Cache-Related Migration Delay (CRMD) are obtained through static code analysis. Experimental results demonstrate the practicality of our approach by comparing predicted worst-case CRMDs with those obtained by a naive approach. To the best of our knowledge, our method is the first one to provide safe upper bounds of cache-related migration delays in multi-core architectures with shared instruction caches.

1 Introduction

Most chip manufacturers have adopted multi-core technologies to both continue performance improvements and control heat and thermal issues. In most multi-core architectures, the different cores have private caches and also shared on-chip caches.

For real-time systems to exploit multi-core architectures, it is required to obtain both tight and safe estimations of a number of metrics required to validate the system temporal behaviour in all situations, including the worst-case:

- tasks worst-case execution times (WCET), for each task considered in isolation,
- worst-case preemption delays, including the time required to refill the architecture caches after a preemption,
- worst-case migration delays, including the time to reload the missing information into the caches after a migration.

Estimating such metrics is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches, memory bus, etc.

In this paper, we propose a new method to estimate the worst-case cache reload cost due to the migration of a task between cores. Such a delay is called hereafter CRMD for *Cache Related Migration Delay*. CRMD is due to the cache refill activity occurring after a migration, and is illustrated below in Figure 1. Figure 1 depicts the impact of task migration on the contents of private and shared caches in a multi-core platform. The depicted platform is made of C cores, each having a private L1 instruction cache and a shared L2 instruction cache.

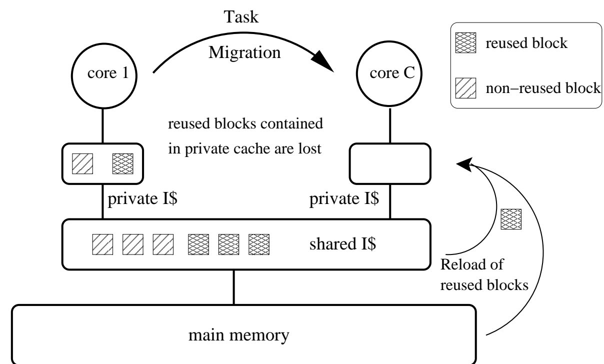


Figure 1. Impact of task migration on cache contents

Consider a task, initially running on *Core 1*, which migrates on *Core C*. At the migration point (see Fig. 1), both the private and the shared instruction caches contain some program blocks. Some program blocks, termed *reused*

blocks, will be used after the task migration, whereas some other blocks, termed *non-reused blocks*, will not be reused after the task migration. After the migration, all reused cache blocks will be reloaded in all levels of the cache hierarchy.

Task migration thus results in additional cache misses compared to a migration-free execution. Such cache misses occur in the private cache, to load reused blocks. They may also occur in the shared cache in case a reused block has been evicted after first loaded, which can occur when using non inclusive cache hierarchies.

Since exact migration points are not statically known, a task migration may result in additional cache accesses in the shared caches compared to a migration-free execution. We chose to account for these accesses when estimating the WCET of a task. Such a WCET, called *migration-aware WCET* assumes that the task may migrate and thus may cause additional accesses to the shared caches, but does not include the cache reload cost itself. Additional cache misses (in the private L1 cache and shared cache levels) are hereafter called *Cache Related Migration Delay (CRMD)*.

We propose in this paper methods to compute safe estimations of the *migration-aware WCET* and the *Cache Related Migration Delay (CRMD)*, using static analysis of the code of the task subject to migration. Experimental results demonstrate that estimated CRMDs are much lower than when using a naive approach assuming that all useful blocks must be reloaded in all cache levels after a migration.

Contributions. The paper contains two tightly-coupled contributions:

- The first contribution is the proposal of a *migration-aware cache analysis method*. The method estimates the worst-case number of cache hits/misses of an isolated task running on a multi-core platform and subject to migrations, regardless of the number of migrations it will suffer at run-time. The proposed migration-aware cache analysis method accounts for every possible migration point on the shared cache, but does not integrate the impact of the cache-related migration cost itself.

- The second contribution is a method to compute a *tight upper bound of the cache-related migration delay (CRMD)* an isolated task will suffer after each migration to reload the reused cache blocks. The provided CRMD is tight because the CRMD does not consider as misses the accesses that are already detected as misses by the migration-aware cache analysis method. This metric, together with the migration-aware WCET estimate, provides a safe bound of cache-related migration costs in a multi-core system. It can be used in any real-time multi-processor schedulability test for global and semi-partitioned scheduling [3, 2, 14] to the extent that the worst-case number migrations is known.

To the best of our knowledge, our method is the first one to provide safe upper bounds of cache refill costs in case of migrations for multi-core architectures with shared instruction caches. This approach focuses on the computation of the CRMD of a task in isolation and has to be used in combination with a cache-related preemption delays estimation method [20, 25].

Related work. Many static WCET estimation methods have been designed in the last two decades (see [28] for a survey). Static WCET estimation methods need a low-level analysis phase to determine the worst-case timing behavior of the micro-architectural components (pipelines and out-of-order execution, branch predictors, caches, etc.). Regarding cache memories on mono-core architectures, two main classes of approaches have been proposed: *static cache simulation* [18, 19], based on dataflow analysis, and the methods described in [9, 26, 10], based on abstract interpretation. Both classes of methods provide for every memory reference a classification of the outcome of the reference in the worst-case execution scenario (e.g. *always-hit*, *always-miss*, *first-miss*, etc.). These methods, originally designed for code only, and for direct-mapped or set-associative caches with a Least Recently Used (LRU) replacement policy, have been later extended to other replacement policies [13], data and unified caches [27], and caches hierarchies [12]. Cache-aware WCET estimation methods have recently been extended to multi-core platforms [29, 11]; the cited methods take into account the interferences caused by shared caches. The proposed method for evaluating migration-aware WCETs is based on [12], itself based on abstract interpretation for static cache analysis [9, 26, 10].

The presence of caches not only impacts the execution time of tasks considered in isolation but also results in an indirect cost required to refill the caches after a preemption. Static analysis techniques, close to those designed for cache-aware WCET estimation, aim at producing safe upper bound of CRPDs (cache-related preemption delays) [20, 25]. Such techniques statically analyze the code of the preempted and preempting tasks to determine which blocks from the preempted task may be reused after the preemption and will have to be reloaded. The method we propose to evaluate CRMDs uses similar analyses and data structures as the ones used to estimate CRPDs.

Extensive empirical evaluations of the impact of real-world overheads (including cache-related preemption and migration overheads) on multiprocessor scheduling algorithms have been presented in [5, 4]. In contrast to our work, these studies focus on giving average-case and worst-

measured overheads and do not aim at providing safe upper bounds of cache-related overheads.

Cache-aware multi-core scheduling have been presented in [6, 7] for soft real-time applications; the idea of this direction of work is to improve task scheduling in multi-core platforms based on the cache behaviour of real-time tasks. In this paper, we focus on the estimation of cache-related overheads, and consider their exploitation by multiprocessor scheduling algorithms as outside the scope of the paper.

Finally, [24] which is the work closest to ours, assumes a multi-core architecture with a private cache hierarchy. They introduce new hardware support to move the cache contents from one private cache to another to reduce the migration cost. Our approach do not require any hardware modification and the cache hierarchy can be shared between cores except the first cache level.

Paper outline. The rest of the paper is organized as follows. Section 2 presents the assumptions our analysis is based on, regarding the target architecture and task scheduling. Section 3 presents the migration-aware cache analysis method. Section 4 focuses on the estimation of cache-related migration delays. Experimental results are given and discussed in Section 5. Finally, Section 6 gives some conclusions and direction for future work.

2 Assumptions

A multi-core architecture is assumed. Each core has a private first-level (L1) instruction cache, followed by shared instruction cache levels. Each shared cache is shared between all the cores of the architecture. The caches are set-associative and each level of the cache hierarchy is non-inclusive:

- A piece of information is searched for in the cache of level ℓ if and only if a cache miss occurred when searching it in the cache of level $\ell-1$. Cache of level 1 is always accessed.
- Every time a cache miss occurs at cache level ℓ, the entire cache block containing the missing piece of information is always loaded into the cache of level ℓ.
- There are no actions on the cache contents (i.e. invalidations, lookups/modifications) other than the ones mentioned above.

Our study concentrates on instruction caches; it is assumed that the shared caches do not contain data. This study can be seen as a first step towards a general solution for shared caches. It can also push to the use of separate shared instruction and data caches instead of unified ones[1].

[1]Unified caches could be partitioned at boot time for instance in a A-way instruction cache and a B-way data cache.

Our method assumes a LRU (Least Recently Used) cache replacement policy. Furthermore, an architecture without timing anomalies as defined in [16] is assumed. The access time variability to main memory and shared caches, due to bus contention, is supposed to be bounded and known, by using for instance *Time Division Multiple Access (TDMA)* like in [23] or other predictable bus arbitration policies [21]. Figure 2 illustrates two different supported architectures.

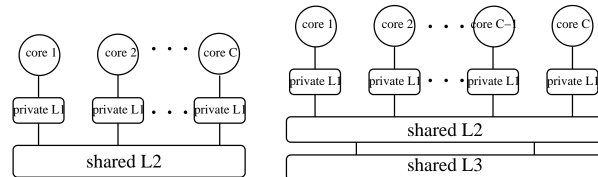


Figure 2. Two examples of supported architectures

The focus in this paper is to estimate the worst-case *cache related migration delay (CRMD)* suffered from a hard real-time task after a migration from one core to another in a multi-core platform. The migrated task is considered in isolation from the tasks running at the same time on the multi-core platform. The computation of interferences due to intra-core or inter-core of other tasks is considered out of the scope of this paper; for related studies tackling these issues, the reader is referred to [20, 25] regarding intra-core interferences or [29, 11] regarding inter-core interferences.

3 Migration-aware multi-level cache analysis

As a first step to present the *migration-aware cache analysis* method, paragraph 3.1 focuses on the analysis of the worst-case behaviour of the memory hierarchy when completely ignoring task migrations, what we call *migration-ignorant cache analysis*. The impact of task migration on shared caches is considered in paragraph 3.2.

3.1 Migration-ignorant cache analysis

The cache analysis, originally presented in [12] and briefly described is applied successively on each level of the cache hierarchy, from the first cache level to the main memory. The analysis is contextual in the sense that it is applied for every call context of functions (functions are virtually inlined). The references considered by the analysis of cache level ℓ depend on the outcome of the analysis of cache level $\ell-1$ to consider the filtering of memory accesses between cache levels, as depicted in Figure 3 and detailed below.

The outcome of the static cache analysis for every cache level ℓ is a *Cache Hit/Miss Classification (CHMC)* for each

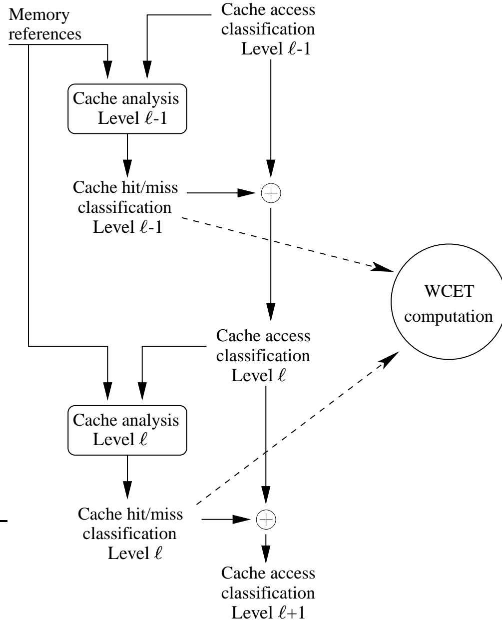


Figure 3. Multi-level cache analysis without task migration

reference, determining the worst-case behavior of the reference with respect to cache level ℓ:

- *always-miss* (AM): the reference will always result in a cache miss,
- *always-hit* (AH): the reference will always result in a cache hit,
- *first-miss* (FM): the reference could neither be classified as hit nor as miss the first time it occurs but will result in cache hit afterwards,
- *not-classified* (NC): in all other cases.

Moreover, at every level ℓ, a *Cache Access Classification (CAC)* specifies if an access may occur or not at level ℓ, and thus should be considered by the static cache analysis of that level. There is a CAC, noted $CAC_{r,\ell,c}$ for every reference r, cache level ℓ, and call context c[2]. The CAC defines three categories for each reference, cache level, and call context:

- A (Always): the access always occurs at cache level ℓ.
- N (Never): the access never occurs at cache level ℓ.
- U (Uncertain) when the access cannot be classified in the two above categories.

The cache analysis at every cache level is based on a state-of-the-art single-level cache analysis [26], based on abstract interpretation. The method is based on three separate fixpoint analyses applied on the program control flow graph, for every call context:

- a *Must* analysis determines if a memory block is always present in the cache at a given point: if so, the block is classified *always-hit (AH)*;
- a *Persistence* analysis determines if a memory block will not be evicted after it has been first loaded; the classification of such blocks is *first-miss (FM)*.
- a *May* analysis determines if a memory block may be in the cache at a given point: if not, the block is classified *always-miss (AM)*. Otherwise, if neither detected as always present by the *Must* analysis nor as persistent by the *Persistence* analysis, the block is classified *not classified (NC)*;

Abstract cache states (ACS) are computed for every basic block according to the semantics of the analysis (Must/May/Persistence) and the cache replacement policy by using functions ($Update$ and $Join$) in the abstract domain. $Update$ models the impact on the ACS of every reference inside a basic block; $Join$ merges two ACS at convergence points in the control flow graph (e.g. at the end of conditional constructs).

Figure 4 gives an example of an ACS of a 2-way set-associative cache with LRU replacement policy on a *Must* analysis (only one cache set is depicted). An *age* is associated to every cache block of a set. The smaller the block age the more recent the access to the block. For the *Must* analysis, each memory block is represented only once in the ACS, with its maximum age. It means that its actual age at run-time will always be lower than or equal to its age in the ACS.

At every cache level ℓ, the three analyses (*Must, May, Persistence*) consider all references r guaranteed to occur at level ℓ ($CAC_{r,\ell} = A$). References with $CAC_{r,\ell} = N$ are not analysed. Regarding uncertain references ($CAC_{r,\ell} = U$), for the sake of safety, the ACS is obtained by exploring the two possibilities ($CAC_{r,\ell} = A$ and $CAC_{r,\ell} = N$) and merging the results using the $Join$ function (see Figure 5). For all references r, $CAC_{r,1} = A$, meaning that the L1 cache is always accessed.

Since task migrations are not considered in this paragraph, the CAC of a reference r for a cache level ℓ only depends on CHMC of r at level $\ell - 1$ and the CAC of r at level $\ell - 1$ to model the filtering of accesses in the cache hierarchy (see Figure 3). Table 1 shows all the possible cases of computation of $CAC_{r,\ell}$ from $CHMC_{r,\ell-1}$ and $CAC_{r,\ell-1}$.

$CAC_{r,\ell-1}$ \ $CHMC_{r,\ell-1}$	AM	AH	FM	NC
A	A	N	U	U
N	N	N	N	N
U	U	N	U	U

Table 1. Cache access classification for level ℓ ($CAC_{r,\ell}$)

[2] The call context c will be omitted from the formulas when the concept of call context is not relevant.

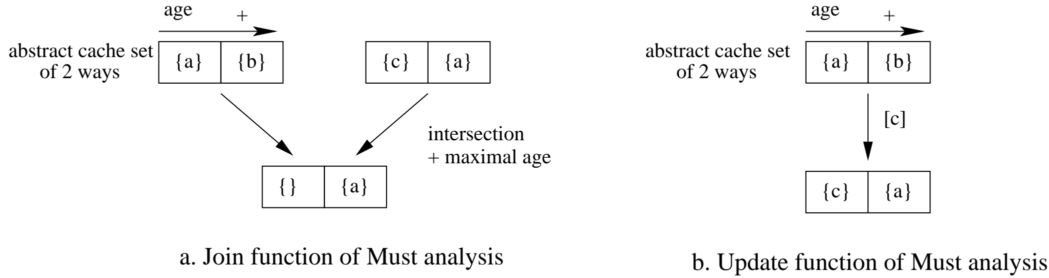


Figure 4. $Join$ **and** $Update$ **functions for the Must analysis with LRU replacement**

The CHMC of reference r is used to compute the cache contribution to the WCET of that reference (i.e. the sum of the cache level latencies where the access to r may occur plus the memory latency if the access may occur in the memory), which can be included in well-known WCET computation methods [17, 22].

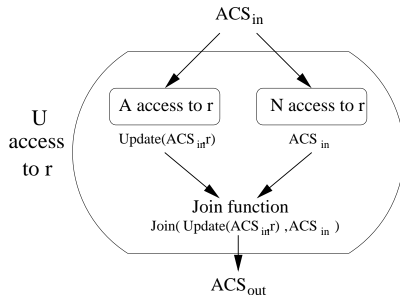


Figure 5. Function for U access

3.2 Migration-aware cache analysis

As previously depicted in Figure 1, migrating a task results in additional accesses to the shared caches after the migration. Since the exact migration points are not known off-line, some accesses to the shared cache levels that would not occur in a migration-free execution may occur after the migration. Thus, our migration-aware cache analysis account for every possible migration point without integrating the cache-related migration cost itself.

As it was previously demonstrated in [12], considering these additional accesses to the shared caches as always occurring might not be safe, because this can lead to an underestimation of the reuse distance of blocks in the shared caches. As a consequence, the migration-aware cache analysis considers all accesses to the first shared cache level (usually L2 cache) as *Uncertain* ($CAC_{r,\ell} = U$ with ℓ the first shared cache level). This ensures a safe cache analysis of the shared cache levels in the presence of unknown migration points. Apart from the introduction of U accesses in the first shared cache level, the cache analysis and computation of migration-aware WCETs, noted hereafter $WCET_{MA}$ are achieved as described in § 3.1.

Note that $WCET_{MA}$ is more pessimistic than its migration-ignorant counterpart. This is because $WCET_{MA}$ accounts for the impact of migrations on the shared cache(s), which are not accounted for when estimating the migration-ignorant WCET. The additional pessimism due to the consideration of possible task migrations is evaluated in Section 5.

4 Computing Cache-Related Migration Delay (CRMD)

This section focuses on the computation of the Cache-Related Migration Delay (CRMD) suffered by a task τ every time it migrates from one core to another. When τ migrates n times, its WCET is then:

$$WCET_{MA} + n * (CRMD + \delta)$$

with δ the migration cost excluding cache reloads. The maximum number of migrations suffered by a task at runtime depends on the scheduling policy[3].

Due to the use of the migration-aware cache analysis, the CRMD only depends on the additional accesses to the memory hierarchy after the migration. As explained before, and previously illustrated in Figure 1, extra accessed concern blocks reused after the migration of τ, and may introduce additional misses in the L1 cache as well as in the shared cache levels.

Useful cache blocks. To bound the number of reused blocks of the L1 cache at each program point, we use the notion of *useful cache blocks* previously defined in [15] for the computation of Cache-Related Preemption Delay (CRPD). A useful cache block is defined as follows: *a useful cache block at an execution point is defined as a memory block that may be re-referenced before being replaced*. In other

[3] This estimation is independent from any scheduling policies. It can be reduced by considering the n highest values of the CRMD instead of n times the maximal value with some extra restrictions on the migration points.

words, the set of useful cache blocks at a given program point p is a safe over-approximation of the set of reused blocks at program point p.

The technique used to determine the useful cache blocks is based on the traditional *reaching definitions* and *live variables* data flow analyses [1]:

- Similarly to the reaching definitions analysis, the *reaching memory blocks* (RMB) analysis determines all the memory blocks that may be in the cache at a program point p when p is reached via any incoming program path.
- As in the live variables analysis, the *live memory blocks* (LMB) analysis determines all the memory blocks that may be referenced before their eviction via any outgoing path from p.

The useful cache blocks at program point p (noted $useful(p)$) are the memory blocks that are present in the result of both the RMB analysis (noted $RMB(p)$) and the LMB analysis (noted $LMB(p)$).

$$useful(p) = RMB(p) \cap LMB(p)$$

Suppose that task τ migrates at program point p. Instead of considering a miss in *all* cache levels for each useful cache block at point p, our computation produces tighter results by integrating in the CRMD only misses which are not already integrated in the migration-aware WCET estimate.

Notations. Before detailing the computation of the CRMD, let us introduce some formulae obtained from the results of the migration-aware cache analysis. First, we introduce the notion of *always-persistent* block to determine if a cache block *cb* is ensured to hit after a migration in a given shared cache level ℓ (i.e. its cache hit/miss classification is *always-hit* or *first-miss* in all execution contexts):

$$always_persistent_\ell(cb) =$$

$$\begin{cases} true & \text{if } \forall_{ctx}, \forall_{instr} \in cb, \\ & \quad CHMC_{\ell,ctx}(instr) = AH \\ & \quad \vee CHMC_{\ell,ctx}(instr) = FM \\ false & \text{otherwise} \end{cases}$$

We also define the notion of *always-filtered* block by a previous shared cache level(s) of ℓ if the cache block *cb* is always-persistent in at least one previous shared cache level:

$$always_filtered_\ell(cb) =$$

$$\begin{cases} false & \text{if } \ell = 2 \\ \bigvee_{p\ell=2}^{\ell-1} always_persistent_{p\ell}(cb) & \text{otherwise} \end{cases}$$

Similarly, we introduce $at_least_once_persistent_\ell(cb)$ to detect the case where a cache block *cb* produces a hit in shared cache level ℓ in at least one execution context:

$$at_least_once_persistent_\ell(cb) =$$

$$\begin{cases} true & \text{if } \exists_{ctx}, \exists_{instr} \in cb, \\ & \quad CHMC_{\ell,ctx}(instr) = AH \\ & \quad \vee CHMC_{\ell,ctx}(instr) = FM \\ false & \text{otherwise} \end{cases}$$

and $at_least_once_filtered_\ell(cb)$ by a previous shared level(s) of ℓ if the cache block *cb* is at-least-once-persistent in at least one previous shared level:

$$at_least_once_filtered_\ell(cb) =$$

$$\begin{cases} false \text{ if } \ell = 2 \\ \bigvee_{p\ell=2}^{\ell-1} at_least_once_persistent_{p\ell}(cb) \\ \quad \text{otherwise} \end{cases}$$

Finally, we define *private-filtered* to determine if a cache block is completely filtered by the private L1 cache in at least one execution context during the computation of $WCET_{MA}$:

$$private_filtered(cb) = \\ \exists ctx, \forall instr \in cb, CHMC_{L1,ctx}(instr) = AH$$

Computing CRMD. A miss that could occur for the first reference in the case of a *first-miss* is already counted by the cache-aware migration analysis and there is no need to count it twice except in the case the access is private-filtered.

The L1 cache is always accessed, thus the latency of the L1 cache is already included in $WCET_{MA}$ and do not need to be counted in the CRMD. For a given shared cache level ℓ, an access to a useful cache block *ucb* after a migration has to be counted if the access is private-filtered because in this case, the access could be not have been counted during $WCET_{MA}$ computation. Moreover, if the access is not private-filtered but this access is not filtered by a previous shared cache level (i.e. $\neg always_filtered_\ell(ucb)$) and is at-least-once-persistent, the access has to be counted. Remark that if the access is ensured to never produce a hit (i.e. $\neg at_least_once_persistent_\ell(ucb)$), the latency of this access in shared cache level ℓ is already in $WCET_{MA}$. More formally, we define the cost added to the CRMD of a shared cache level ℓ at a given program point p as follows:

$$cost_share_level_\ell^p = | \{ucb \in useful(p), (\neg always_filtered_\ell(ucb) \wedge at_least_once_persistent_\ell(ucb)) \vee private_filtered(ucb)\} | * latency_\ell$$

The accesses to the main memory which have to be included in the CRMD are similar. If the access is private-filtered, this access could be not counted during $WCET_{MA}$ computation. Moreover, if the access is not private-filtered but this access is not filtered by any previous shared cache levels (i.e. $\neg always_filtered_{h\ell+1}(ucb)$ where $h\ell$ represents the level of the highest cache level and $h\ell+1$ represent the level of the main memory) and is at-least-once-filtered by a shared cache level, the main memory latency of the access have to be counted. More formally, we define the cost added to the CRMD of the main memory at a given program point p as follows:

$$cost_memory^p = | \{ucb \in useful(p), (\neg always_filtered_{h\ell+1}(ucb) \wedge at_least_once_filtered_{h\ell+1}(ucb)) \vee private_filtered(ucb)\} | * latency_{memory}$$

Thus the CRMD at program point p, noted $CRMD^p$ is the sum of the cost of each shared cache level plus the memory cost.

$$CRMD^p = cost_memory^p + \sum_{\ell=2}^{h\ell} cost_share_level_\ell^p$$

Finally, the CRMD of one single migration is equal to the biggest value of $CRMD^p$ computed for all the program points:

$$CRMD = max(CRMD^p, \forall p \in program)$$

5 Experimental results

5.1 Experimental setup

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization and with the default linker memory layout. The WCETs of tasks are computed by the Heptane timing analyzer [8], more precisely its Implicit Path Enumeration Technique (IPET). The analysis is context sensitive (functions are analysed in each different calling context). To separate the effect of the caches from those of the other parts of the processor micro-architecture, WCET estimation only takes into account the contribution of instruction caches to the WCET. The effects of other architectural features are not considered. In particular, timing anomalies caused by interactions between caches and pipelines, as defined in [16] are disregarded. The cache classification *not-classified* is thus assumed to have the same worst-case behavior as *always-miss* during the WCET computation in our experiments. For space consideration, WCET computation is not detailed here, interested readers are referred to [12].

The migration points considered in the experiments are the ends of basic blocks of the analyzed task.

Name	Description	Code size (bytes)
crc	Cyclic redundancy check computation	1432
fft	Fast Fourier Transform	3536
jfdctint	Integer implementation of the forward DCT (Discrete Cosine Transform)	3040
matmult	Multiplication of two 50x50 integer matrices	1200
minver	Inversion of floating point 3x3 matrix	4408
adpcm	Adaptive pulse code modulation algorithm	7740
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900

Table 2. Benchmark characteristics

Benchmarks. The experiments were conducted on seven benchmarks (see Table 2 for the applications characteristics). All benchmarks are maintained by Mälardalen WCET research group[4].

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a private 4-way L1 cache of 1KB with a cache block size of 32B and a shared 8-way L2 cache of 2KB (or 4KB for the two biggest benchmarks *adpcm* and *statemate*) configured with a cache block size of 32B or 64B. Cache sizes are small compared to usual cache sizes in multi-core architectures. However, there are no large-enough public real-time benchmarks available to experiment our proposal. As a consequence, we have selected quite small commonly used real-time benchmarks and adjusted cache sizes such that the benchmarks do not fit entirely in the caches. All caches are implementing a LRU replacement policy. Latencies of 1 cycle (respectively 10 and 100 cycles) are assumed for the L1 cache (respectively the L2 cache and the main memory).

[4] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

5.2 Results

First, the overestimation resulting from accounting for possible migration points when estimating the WCET of tasks is estimated. Then, the CRMD estimated using our method is compared to a baseline CRMD estimation method considering that all useful blocks are reloaded in all cache levels after a task migration. Finally, the execution time of CRMD estimation is evaluated.

Impact of migrations on task WCET for a non-migrating task. In this paragraph, we focus on the comparison of the estimated migration-ignorant WCET (noted $WCET_{MI}$) and the migration-aware WCET (noted $WCET_{MA}$) when the task does not migrate. The results are mainly given in Table 3, which shows the WCET overestimation in cycles resulting from considering every possible migration point. More details regarding the results of cache analysis are given in Table 4.

Benchmarks	WCET$_{MI}$ (cycles)	WCET$_{MA}$ (cycles)	delta (cycles)	ratio
crc (2KB-32B)	152753	152753	0	0%
crc (2KB-64B)	151953	152753	800	0.53%
fft (2KB-32B)	188655	188655	0	0%
fft (2KB-64B)	187555	188655	1100	0.59%
jfdctint (2KB-32B)	25389	25389	0	0%
jfdctint (2KB-64B)	20689	25389	4700	22.72%
matmult (2KB-32B)	16704	16704	0	0%
matmult (2KB-64B)	16504	16704	200	1.21%
minver (2KB-32B)	20646	20646	0	0%
minver (2KB-64B)	16446	20646	4200	25.54%
adpcm (4KB-32B)	310391	316391	6000	1.93%
adpcm (4KB-64B)	322125	383439	61314	19.03%
statemate (4KB-32B)	141303	142603	1300	0.92%
statemate (4KB-64B)	115903	152325	36422	31.42%

Table 3. Migration-ignorant WCET vs migration-aware WCET

We observe from Table 4 three different situations, which allows to explain the results given in Table 3.

- The first situation is when the migration-ignorant cache analysis does not detect any hit in the L2 cache, or detects very few hits in the L2 cache (in Table 4 number of L1 misses $\approx$ number of L2 misses). This situation occurs when the migration-ignorant cache analysis does not detect spatial and temporal locality in the L2 cache. In this situation, the migration-aware WCET is very close to the migration-ignorant WCET.

- The second situation occurs when the migration-ignorant cache analysis detects temporal locality but no spatial locality in the L2 cache (in Table 4 number of L1 misses $\gg$ number of L2 misses, with L2 cache lines of 32B). In this situation, the migration-aware

Benchmarks	Metrics	Migration-ignorant	Migration-aware
crc (2KB-32B)	nb of L1 accesses	141643	141643
	nb of L1 misses	101	101
	nb of L2 misses	101	101
crc (2KB-64B)	nb of L1 accesses	141643	141643
	nb of L1 misses	101	101
	nb of L2 misses	93	101
fft (2KB-32B)	nb of L1 accesses	80305	80305
	nb of L1 misses	7575	7575
	nb of L2 misses	326	326
fft (2KB-64B)	nb of L1 accesses	80305	80305
	nb of L1 misses	7575	7575
	nb of L2 misses	315	326
jfdctint (2KB-32B)	nb of L1 accesses	8039	8039
	nb of L1 misses	725	725
	nb of L2 misses	101	101
jfdctint (2KB-64B)	nb of L1 accesses	8039	8039
	nb of L1 misses	725	725
	nb of L2 misses	54	101
matmult (2KB-32B)	nb of L1 accesses	11204	11204
	nb of L1 misses	50	50
	nb of L2 misses	50	50
matmult (2KB-64B)	nb of L1 accesses	11204	11204
	nb of L1 misses	50	50
	nb of L2 misses	48	50
minver (2KB-32B)	nb of L1 accesses	4146	4146
	nb of L1 misses	150	150
	nb of L2 misses	150	150
minver (2KB-64B)	nb of L1 accesses	4146	4146
	nb of L1 misses	150	150
	nb of L2 misses	108	150
adpcm (4KB-32B)	nb of L1 accesses	186301	186301
	nb of L1 misses	3759	3759
	nb of L2 misses	865	925
adpcm (4KB-64B)	nb of L1 accesses	186435	186569
	nb of L1 misses	3779	3797
	nb of L2 misses	976	1589
statemate (4KB-32B)	nb of L1 accesses	10933	10933
	nb of L1 misses	1797	1797
	nb of L2 misses	1124	1137
statemate (4KB-64B)	nb of L1 accesses	10673	10945
	nb of L1 misses	1763	1798
	nb of L2 misses	876	1239

Table 4. Migration-ignorant vs migration-aware cache analysis (estimated number of accesses)

WCET is still close to the migration-ignorant WCET. The good result comes from the presence of the persistence analysis, which detects blocks as persistent even though accesses to the L2 cache are considered as *Uncertain* (U).

- Finally, the third and last situation occurs when the migration-ignorant cache analysis detects both temporal and spatial locality in the L2 cache (in Table 4 number of L1 misses $\gg$ number of L2 misses, with L2 cache lines of 64B). In this situation, the migration-aware WCET might be significantly larger than is migration-ignorant counterpart. This is because the introduction of U accesses in the migration-aware cache analysis prevents the cache analysis from detecting spatial locality in the L2 cache.

It can be remarked that there are for some benchmarks (*adpcm* and *statemate*) a variation of worst-case execution path between the migration-aware and migration-ignorant cases (different numbers of accesses along the worst-case execution path for the L1 cache).

Benchmarks	# useful cache block	CRMD baseline in cycles	CRMD in cycles
crc (2KB-32B)	31	3410	510
crc (2KB-64B)	31	3410	400
fft (2KB-32B)	32	3520	1050
fft (2KB-64B)	32	3520	610
jfdctint (2KB-32B)	20	2200	460
jfdctint (2KB-64B)	20	2200	360
matmult (2KB-32B)	17	1870	190
matmult (2KB-64B)	17	1870	140
minver (2KB-32B)	14	1540	280
minver (2KB-64B)	14	1540	240
adpcm (4KB-32B)	24	2640	970
adpcm (4KB-64B)	24	2640	690
statemate (4KB-32B)	5	550	20
statemate (4KB-64B)	5	550	110

Table 5. Estimated Cache-Related Migration Delay (CRMD)

Evaluation of CRMD. Table 5 compares for every benchmark and cache configuration the CRMD obtained by our proposed method (column 4) to a simple baseline, albeit safe method considering that all useful blocks have to be reloaded in all cache levels after a task migration (column 3). Column 2 gives the number of useful cache blocks per benchmark and cache configuration.

The numbers given in the table show that the estimated CRMD, obtained by the proposed approach, is much lower than when using the simple baseline approach. Comparing estimated CRMD with measured ones is left for future work.

5.2.1 Analysis time.

The longest measured time to estimate the migration-aware WCET plus to estimate the CRMD was 5 minutes for the biggest benchmarks. This shows empirically that the complexity of CRMD estimation is similar to the one of cache analyses used when estimating WCETs.

6 Conclusions and future work

We have proposed in this paper a new method, based on static analysis, to estimate the worst-case cache reload cost due to the migration of a task between cores (CRMD, for Cache Related Migration Delay). To the best of our knowledge, our method is the first one to provide *safe* upper bounds of cache-related migration delays in multi-core architectures with shared caches. Experimental results have shown that the estimated CRMDs are much less pessimistic than the simple baseline safe approach except when the cache block sizes in the different cache levels are not the same.

As future work, we plan to compare the estimated CRMDs with measured ones in order to evaluate the tightness of our approach. Other research directions will be to extend the approach to data or unified caches. Finally, selecting task scheduling based on CRMD information would be of interest.

Acknowledgments. The authors are grateful to Benjamin Lesage and to the anonymous reviewers for feedback on earlier versions of the paper.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] S. K. Baruah and T. P. Baker. Schedulability analysis of global EDF. *Real-Time Systems*, 38(3):223–235, 2008.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[4] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[5] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society.

[6] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 299–308, July 2008.

[7] J. Calandrino and J. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, July 2009.

[8] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[9] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[11] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, Washington D.C., USA, Dec. 2009. To appear.

[12] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, Dec. 2008.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, n7*, 2003.

[14] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS2009)*, San Francisco, CA, USA, April 2009.

[15] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computer*, 47(6):700–713, June 1998.

[16] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

[17] S. Malik and Y. T. S. Li. Performance analysis of embedded software using implicit path enumeration. *Design Automation Conference*, 0:456–461, 1995.

[18] F. Mueller. *Static cache simulation and its applications*. PhD thesis, Florida State University, 1994.

[19] F. Mueller. Timing analysis for instruction caches. *Real Time Systems*, 18(2-3):217–247, 2000.

[20] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206, 2003.

[21] M. Paolieri, E. Qui nones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2009. ACM.

[22] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real Time Systems*, 1(2):159–176, 1989.

[23] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[24] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. *SIGPLAN Not.*, 44(7):80–89, 2009.

[25] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 278–286, 2004.

[26] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real Time Systems*, 18(2-3):157–179, 2000.

[27] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real Time Systems*, 17(2-3):209–233, 1999.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[29] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

Impact of Code Compression on Estimated Worst-Case Execution Times

Haluk Ozaktas[b], Karine Heydemann[b], Christine Rochange[a], Hugues Cassé[a]

[a] IRIT – Université de Toulouse
118 route de Narbonne
31062 Toulouse cedex 9, France
{rochange, casse}@irit.fr

[b] LIP6 – Université de Paris VI
4, place Jussieu
75252 Paris cedex 5
{haluk.ozaktas, karine.heydemann}@lip6.fr

Abstract

Code compression techniques might be useful to meet code size constraints in embedded systems. In the average case, the impact of code compression on the performance is double-edged: on one side, the number of accesses to memory hierarchy is reduced because several instructions are coded in a single word, and this is likely to reduce the execution time; on the other side, the decompression penalty increases the processing time of compressed instructions. Nevertheless, experimental results show that the execution time might be lowered by code compression.

In this paper, our goal is to analyze the impact of code compression on the estimated Worst-Case Execution Time of critical tasks that must meet at the same time code size constraints and timing deadlines. Changes in the access patterns to the instruction cache are indeed likely to alter the accuracy of the cache analysis within the process of determining the WCET.

Experimental results show that, besides reducing the code size, our code compression scheme also improves the WCET estimates in most of the cases.

1. Introduction

Embedded systems are often constrained in terms of different criteria like code size, execution time or energy consumption. Various techniques have been proposed to improve one of these criteria: code compression schemes aim at reducing the code size, compiler optimizations help in improving the execution time and various approaches determine the best placement of instructions and data in the memory space to limit the energy requirements.

However, the impact of the techniques that improve one of the criteria onto the other ones is seldom analyzed. It is the goal of the French MORE[1] project to get insight into such board effects and to determine sets of code transformations that jointly improve several criteria.

In this paper, we focus on the impact of code compression techniques on the execution time and more particularly on the Worst-Case Execution Time (WCET) of time-critical software.

Code compression reduces the code size by compacting the original code into a non executable format. At runtime, a decompression step is needed to retrieve the initial code. Code compression has been and is still an active research area [5][24]. In this paper, we consider a compression scheme that combines two state-of-the-art approaches [7][21]. A dictionary-based compression algorithm is used to replace sequences of instructions by special instructions that trigger decompression at runtime. Decompression takes place in the processor pipeline, between the fetch and decode stages. This is compatible with wide-issue high performance superscalar architectures, contrary to post-cache decompression, while still leaving compressed code in the instruction cache. Thus, the number of cache misses is reduced which might improve both the execution time and the energy consumption.

To estimate Worst-Case Execution Times, we consider state-of-the-art techniques: the worst-case execution costs of basic blocks are computed using parameterized execution graphs [22], the behavior of the instruction cache is analyzed using abstract interpretation [1][3] and an upper bound of the whole program execution time is derived using the IPET method [19]. All these algorithms can be invoked within the OTAWA framework [6] and have been adapted for this study to take into account the effects of code compression: the execution costs of basic blocks include decompression penalties and the instruction cache analysis considers the instruction addresses in the compressed code.

The paper is organized as follows. Section 2 gives an overview of code compression techniques and details the algorithm considered in this study. Section 3 presents the strategy used to estimate WCETs and discusses the expected (theoretical) impact of code compression on the

[1] MORE stands for Multicriteria Optimization for Real-time Embedded systems. This project is supported by the ANR French National Agency for Research.

accuracy of the estimates. The methodology for experiments is detailed in Section 4 and experimental results are provided and analyzed in Section 5. Section 6 concludes the paper.

2. Code compression

2.1. State-of-the-art

Code compression has been and remains a hot topic [5][24]. The proposed approaches differ in the compression strategy (statistical as Huffman coding, dictionary-based or any combination of both) as well as in the implementation (by software or in hardware) and in the location of the decompression engine: between the cache and the memory for the *pre-cache* approaches, between the cache and the processor for *post-cache* schemes or inside the processor core.

A pre-cache decompression engine is only invoked on cache misses: decompression operations are then less frequent than for post-cache schemes but, since the cache contains original (uncompressed) code, the decompression time penalty cannot be balanced by a reduced number of cache misses. This makes it necessary to trade-off between the code size improvement and the execution time degradation [18]. IBM's Code Pack is an example of pre-cache dictionary-based compression scheme used in some processors of the PowerPC family [16]. Every half-word of a cache line is encoded using a variable-size encoding word. On an instruction cache miss, two compressed cache lines are decompressed and fetched into the cache. For some programs, this might act as a prefetch and balance the decompression timing overhead [18].

Besides reducing the size of the code, compression can also improve the performance and reduce the energy requirements if the compressed code is stored in the instruction cache [21]. However, since post-cache decompression is done on the critical path and is potentially needed on every access to the cache, it must be fast to avoid increasing the processor cycle time or the cache access time. This approach requires coping fast with two addressing spaces, one related to the compressed code and the other one seen by the processor for which the code compression is completely transparent [14][21]. Moreover, post-cache decompression is very hard to implement for superscalar processors and might impair the efficiency of a branch predictor.

Decompression can also be done within the pipeline: it is then very close to the translation engine for micro-coded instructions [7]. This is the solution that we have considered in this paper since it suits any superscalar architecture and avoids handling two address spaces.

Another approach to reduce the size of binary codes consists in using shorter instructions. Some processors support dual-width instruction sets: 16-bit instructions can be used to limit the code size while 32-bit instructions might be preferred to fit performance requirements. The ARM Thumb is the best known example of dual-width instruction sets [11]. The translation of 16-bit instructions into 32-bit codes is immediate in the decode stage. A binary code that uses 16-bit Thumb instructions is typically smaller by 30% than regular code and suffers longer execution times due to the limited expressiveness of 16-bit instructions. To limit the performance degradation, the most frequently executed code regions are usually compiled with 32-bit instructions while less frequently executed regions are compiled with 16-bit instructions.

Code compression techniques are orthogonal to the use of reduced instruction sets since, besides shortening the instruction codes, they exploit their redundancy.

Earlier works report a mean reduction of the code size by 20% with dictionary-based approaches and performance and energy gains that vary according to the applications and to the cache sizes.

As far as we know, the only paper on reducing the code size for real-time applications focuses on the use of a 16-bit instruction set [17]. It shows that it is necessary to trade-off between the reduction of the code size and an increase of the Worst-Case Execution Time. The proposed strategy then consists in limiting the use of 16-bit instructions to code regions that have a little impact on the overall WCET, so that it is not too much degraded.

In this paper, we show that the code compression technique that we considered can improve both the code size and the WCET of hard critical software.

2.2. Compression scheme

In the MORE project, we decided to use a post-cache code compression technique that is likely to optimize at the same time the code size and the energy consumption. Since our intention is to consider high-performance processors, we have opted for in-pipeline decompression that, in addition, avoids the complexity of handling different address spaces. Since the decompression overhead was critical, we designed dictionary-based compression scheme that might be less efficient (in terms of compression rate) than statistical algorithms but that allows faster decompression.

In our solution, the dictionary contains full instructions. In order to limit the cost of the dictionary and to keep its access time short, it is desirable to restrict its size. Keeping the dictionary small is also necessary to limit the width of the dictionary index (*log(n)* bits are required for an *n*-entry dictionary), which is important to insure the efficiency of the code compression scheme: the smaller the index width, the better the compression rate. Moreover, a dictionary does not need to hold all

the instructions that appear in the code: when an instruction in the dictionary appears only once in the code, the code size is not improved and even degraded [4] (since the instruction is stored twice: once in the code, in a compressed form, and once in the dictionary).

As far as the dictionary does not hold all the instructions, the compressed code contains both compressed and uncompressed instructions. For our compression scheme design, we have fixed the dictionary size to 256 entries, which is a standard size for hardware implementation and one-cycle decompression [12][21]. Besides, this size allows covering a significant part of the static code and reaching good compression rate even with large applications (the most redundant instructions are generally not numerous).

Our compression scheme replaces two or three successive instructions present in the dictionary by one 32-bit encoding instruction (ISA-width encoding avoid alignment issues). This encoding instruction is composed of an invalid code operation of the target ISA, two information bits and three 8-bit slots that contain the index of the dictionary entries that store the corresponding instructions. This is illustrated in Figure 1. Absolute branch instructions can be included in the dictionary by patching them afterwards. Relative jumps can also be included if the jump displacement is nullified and the patched relative value is encoded into the encoding instruction.

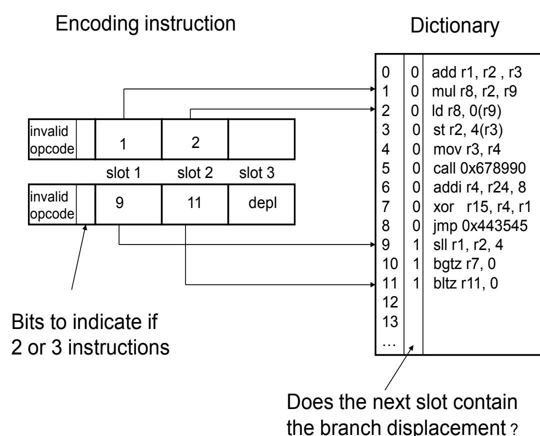


Figure 1. Encoding instructions

The main issue of a dictionary-based compression scheme is how the dictionary is built. To maximize code size reduction, it is preferable to include the most statically repeated instructions whereas selecting the most executed instructions favours the reduction of the number of instruction cache misses. To benefit from both code size and cache miss rate improvement, our compression scheme builds *P*% of the dictionary with the most executed instructions and fills the remaining entries with the most statically repeated instructions. Once the dictionary is built, sequences of instructions that are in the dictionary are encoded. To avoid impairing branch prediction, only instructions that belong to the same basic block can be encoded together.

2.3. Decompression

Decompression is done in the processor pipeline. A decompression stage must be added except if the processor already has a stage for translation of micro-coded instructions into instructions as in the Intel IA-32 architecture. The decompression stage is placed between the fetch and the decode stages. Non-compressed instructions are simply forwarded to the decode stage. In case of a compressed instruction, extra cycles are needed to access the dictionary. As the dictionary is much smaller and less complex than a cache, a one-cycle access is feasible. The dictionary access fills the pipeline with two or three new instructions depending on the number of instructions encoded into a single one.

3. WCET analysis

3.1. General overview

The estimation of Worst-Case Execution Times (WCETs) usually includes three steps: the *flow analysis* determines flow facts like loop bounds and infeasible paths [2][9][10][13][15]; the *low-level* analysis computes the worst-case execution costs of basic blocks taking into account the specifications of the target hardware [20][22][23]; and finally the WCET computation combines the flow facts and the execution costs to find out the longest path and its execution time [19].

The low-level analysis step is in turn split into two sub-steps: the first one examines the behavior of history-based components (mainly the instruction and data caches) and the second-one computes the execution cost of each basic block when executed in the pipeline.

Since code compression has no impact on flow facts, we focus, in this paper, on the low-level analysis.

3.2. Instruction cache analysis and computation of execution costs

Instruction cache analysis. The most popular technique to analyze the behavior of the instruction cache is based on the determination of Abstract Cache States (ACS): an ACS is the set of concrete cache states that are possible at a given point in the Control Flow Graph (CFG) during the execution of the program [1]. It associates a set *s* of possible *l*-blocks[2] to each cache line.

[2] An *l*-block results from the projection of the CFG on the cache line map: a cache line that contains instructions belonging to *n* different basic blocks is considered as *n* *l*-blocks.

Abstract interpretation techniques [8] are used to compute abstract cache states in input and output of each basic block. The *Update* function computes the output ACS of a basic block from its input ACS, and the *Join* function merges the output ACS of all the predecessors of a basic block to produce its input ACS. The *Update* and *Join* functions are applied repeatedly until the algorithm reaches a fixed point.

This process is applied to *May* and *Must* analyses that determine the set *s* of *l*-blocks that *may* (resp. *must*) be in the cache at each program point. A third analysis, called *Persistence* analysis, is used to detect *l*-blocks that belong to a loop body and remain in the cache between successive iterations (but might miss at the first iteration).

Finally, the results of the *May*, *Must* and *Persistence* analysis are used to assign a category to each *l*-block among: *Always Hit* (each fetch is guaranteed to hit in the cache), *Always Miss* (each fetch is guaranteed to miss), *Not Classified* (the analysis is not able to predict a fixed issue for this fetch) and *Persistent* (the fetch misses each time the heading loop is entered and hits while the loop iterates).

Execution cost computation. The execution cost of a basic block also depends on the history. The possible states of the pipeline when the block starts executing can be determined using abstract interpretation techniques, as in [23]. However, to keep the analysis cost (as well in terms of memory space as in terms of computation time), we have developed another technique that considers any possible pipeline state without enumerating them one by one [22]. It is based on execution graphs that express the data, control and structural dependencies between instructions and computes the possible instruction schedules as a function of the state of the pipeline when the block starts executing. From these possible schedules, an upper bound of the execution cost is derived. This technique is much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy.

Integration of cache miss penalties in execution costs. The instruction cache and the pipeline are often analyzed in a totally decoupled manner: the block execution costs are estimated considering cache hits and a penalty is added for each possible miss detected by the instruction cache analysis. While very convenient, this approach is not safe when the processor has not been proved “timing-anomaly-free”. The term of “timing anomaly” refers to situations where, by example, an increase of the latency of an instruction by *i* cycles leads to an increase of the block execution time by more than *i* cycles. As far as the instruction cache is concerned, this means that the block execution cost with a cache miss might be shorter than with a cache hit.

It is generally hard to prove that a processor is not prone to timing anomalies. In this case, a safe approach is to compute the possible costs of each basic block considering all the possible cache behaviours (for all the instructions of the block). As said before, *l*-blocks that have been classified as *Always Hit* or *Always Miss* have a fixed latency, while those labelled as *Not Classified* can experience either a hit or a miss latency. Thus, for the latter, both latencies must be considered when computing the block cost which means that, if *n* *l*-blocks in the basic block are *Not Classified*, as many as 2^n costs must be evaluated (and the maximum value is kept). Fortunately, cache analysis is usually accurate enough to limit the number of *Not Classified* *l*-blocks. *Persistent* *l*-blocks might undergo a miss latency when the heading loop is entered and always hit when the loop iterates. Again, both cases must be considered and two block costs must be computed: one for each entrance into the loop, and one for the other iterations. If *n* *l*-blocks in the block are *Persistent*, they generally have the same heading loop and then exhibit the same behaviour (they all hit or all miss). Then only two costs have to be computed for all these instructions.

This can be illustrated considering the example given in Figure 2. In this example, basic block b_j contains six *l*-blocks that belong to different categories. Three *l*-blocks are *Persistent* with two different headers. For this basic block, eight cost values must be computed. They are listed in Table 1 (‘H’ stands for *hit* and ‘M’ for *miss*). For *Persistent* and *Not Classified* *l*-blocks, both cases (hit and miss) must be considered. When two *l*-blocks are *Persistent* with the same header (lb_3 and lb_4), they must have the same behaviour.

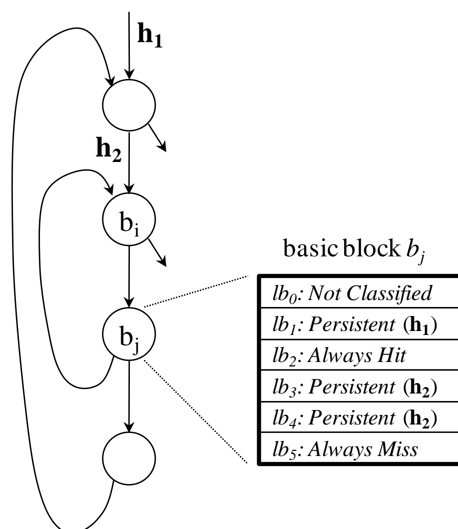


Figure 2. Example.

cost value	considered behaviours					
	lb₀	*lb₁*	*lb₂*	*lb₃*	*lb₄*	*lb₅*
$C^{[0]}$	H	H	H	H	H	M
$C^{[1]}$	H	H	H	M	M	M
$C^{[2]}$	H	M	H	H	H	M
$C^{[3]}$	H	M	H	M	M	M
$C^{[4]}$	M	H	H	H	H	M
$C^{[5]}$	M	H	H	M	M	M
$C^{[6]}$	M	M	H	M	M	M
$C^{[7]}$	M	M	H	M	M	M

Table 1. Possible cache behaviors for basic block b_j of Figure 2.

3.3. Expected impact of code compression on estimated WCETs

The decompression penalty of compressed instructions must be taken into account when estimating block costs. It is expected to have an impact equivalent to the one it has on the observed execution time.

In addition, code compression is likely to have an impact on the results of the instruction cache analysis. The reason for this is that it is expected to alter the number of *l*-blocks in the program as well as their size.

Figure 3 is a reminder of how *l*-blocks are built: in this example, basic block *b* has three *l*-blocks, one that we describe as *full* since it corresponds to a complete cache line and two that we describe as *partial* since they share their cache lines with other *l*-blocks that belong to basic blocks *b*-1 *and* b+1. Each basic block contains *f* full *l*-blocks, where *f* can take any value, including zero, and *p* partial *l*-blocks with *p* in {0, 1, 2}. The number of full *l*-blocks depends on the basic block length and the number of partial *l*-blocks depends on its alignment with respect to cache line boundaries.

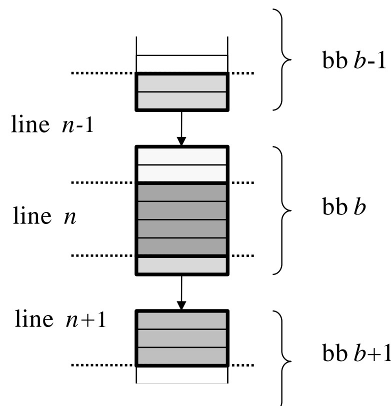


Figure 3. Construction of *l*-blocks.

Let us now discuss what can change when the code is compressed. Figure 4 shows the possible impact on the example code of Figure 3. Here, we assume that several instructions are compressed. As a result, the length of basic block *b* is decreased from 7 to 3 instructions and it has now a single (partial) *l*-block. More generally, code compression shortens the basic blocks and is then likely to reduce their number of full *l*-blocks. The impact on the number of partial *l*-block is less predictable since it depends on the alignment to cache line boundaries.

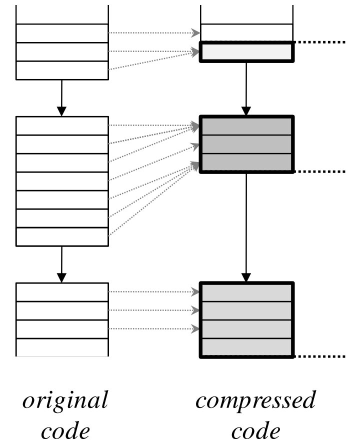


Figure 4. Construction of *l*-blocks in the compressed code.

Now, how these changes on the number and size of the *l*-blocks might impact the cache-related contribution to the WCET?

First, a smaller number of *l*-blocks means a smaller number of accesses to the instruction cache, and this is prone to reduce the Worst-Case Execution Time (as well as the average-case execution time).

Second, an increase of the proportion of partial *l*-blocks might change the distribution into categories. On one hand, partial *l*-blocks are more likely to *Always Hit* than full *l*-blocks since a cache line that contains the beginning of a basic block might have been fetched on the execution of the previous basic block that shares the cache line. In other words, partial *l*-blocks benefit from spatial locality. On the other hand, partial *l*-blocks are prone to generate inaccuracy in the cache analysis: it often cannot be determined whether an *l*-block will hit or miss in the cache when the basic block it belongs to has several possible predecessors. As a result, partial *l*-blocks are prone to be annotated as *Not Classified*.

To conclude, it is difficult to predict whether code compression will improve or degrade the WCET. It might improve it because it reduces the number of accesses to the instruction cache and because the proportion of remaining *l*-blocks classified as *Always Hit* is likely to increase. On the contrary, it might degrade the WCET because the proportion of *Not Classified* *l*-blocks should increase. The goal of this study is to decide between these two possibilities through an experimental approach.

4. Methodology

4.1. Implementation of code compression and WCET analysis

All the techniques involved in this study have been implemented within the OTAWA framework [6]. OTAWA comes as a library that provides a series of classes and tools used for WCET analysis.

Our code compression scheme has been implemented within OTAWA. Two new *Code Analysis* passes have been developed: the first one scans the binary code to compute the frequency of static instructions and the second one simulate the program execution to determine the dynamic frequency of the instructions. These passes have been complemented with a *Dictionary Builder* that is parameterized by the proportion P of the dictionary that must be filled with the instructions that exhibit the highest dynamic frequency. Finally, we implemented the *Code Compressor* that tries to build as many full (i.e. including three original instructions) encoding instructions as possible, while respecting basic block boundaries. The computation of the addresses in the compressed code, including the branch targets, is done after the encoding.

In order to avoid the cost of developing a compressed code generator and a compressed code loader for WCET analysis, our code compression algorithm annotates the Control Flow Graph of the program under analysis to indicate which instructions are compressed and what their addresses in the compressed code are. Then the same CFG can be used to analyze both the original and the compressed codes.

To handle compressed code, both the execution cost computation part (building of execution graphs) and the *l*-block builder have been modified to consider the addresses of compressed instructions.

OTAWA also includes a cycle-level simulator built on the SystemC library. This simulator has been modified to include the decompression engine.

4.2. Experimental procedure

So far, OTAWA is not able to consider several cost values for each basic block, related to the different possible cache behaviors found by the preliminary cache analysis. It considers instead a single cost value which is the maximum of all the computed values for the basic block.

In order to obtain results that correctly reflect the accuracy of the cache analysis, we have decided to compute estimated WCETs from flow information determined by profiling. The program under analysis is simulated and the execution count $x_{i,j}$ of each two-block sequence b_i-b_j is observed. Then what we refer to as the WCET in this paper is estimated as:

$$WCET = \sum_{s_{i,j} \in S} \left(x_{i,j} - \max_{h \in H_{i,j}}(x_h) \right) . c_{i,j}^{>1} + \max_{h \in H_{i,j}}(x_h) . c_{i,j}^{=1}$$

where S is the set of possible two-block sequences, $c_{i,j}^{=1}$ is the maximum cost of block b_j in sequence b_i-b_j, computed considering that the *l*-blocks of b_j that have been classified as *Persistent* miss (first loop iterations) and $c_{i,j}^{>1}$ the maximum cost computed when they hit (other iterations). These maximum cost values are estimated considering both possibilities for all the *Not Classified* *l*-blocks. The set of loop header edges related to the *Persistent* *l*-blocks is denoted as $H_{i,j}$ and x_h is the execution count observed for an header edge. Whenever the block contains several *Persistent* *l*-blocks with different headers, a maximum value is computed considering all the possible cost values, which is likely to generate overestimation. Fortunately, this case is rather infrequent. When the block does not contain any *Persistent* *l*-block, max(x_h) is null.

To illustrate this formula, let us consider the example given in Figure 2 and Table 1. In this example, the contribution of basic block b_j to the estimated WCET would be computed as:

$$C_{i,j}^{=1} = \max(C_{i,j}^{[1]}, C_{i,j}^{[2]}, C_{i,j}^{[3]}, C_{i,j}^{[5]}, C_{i,j}^{[6]}, C_{i,j}^{[7]})$$

$$C_{i,j}^{>1} = \max(C_{i,j}^{[0]}, C_{i,j}^{[4]})$$

This way, we estimate the Worst-Case Execution Time *related to the flow facts obtained by profiling*. For some of the benchmarks, the input data really drive the execution on to the longest path. For other ones, we were not able to determine the worst-case input data and thus the profiled execution path might not be the worst-case path. Nevertheless, we estimated the WCET for this path which makes sense since our goal is to analyze the accuracy of the cache and pipeline analysis, not that of the flow analysis.

4.3. Benchmarks

For the experiments, we used the benchmarks listed in Table 2. Most of them come from the collection hosted on the Mälardaalen University website [26], which is often used for WCET analysis experiments. The `seg` code, that we have developed, implements well-known algorithms, includes three functions that are considered as three benchmarks (but reside in the same executable file): `seg1` corresponds to the function that finds regions of adjacent similar pixels in the image, `seg2` refers to the function that fuses adjacent regions and `seg3` relates to the function that fuses pixels that belong to fused regions. We also have developed the `airbag` benchmark that implements the algorithms described in [25].

4.4. Processor architecture and cache configuration

Since we were mainly interested in the effects of code compression on the analysis of the cache instruction, we have considered a simple pipeline configuration: two-way superscalar, with in-order execution, no branch prediction and a perfect data cache (i.e. all the accesses to data hit in the cache).

We have considered several instruction cache configurations with a cache line size of 16 or 32 bytes and a cache size ranging from 128 to 2048 bytes (to get realistic results for small benchmarks). In all cases, the instruction cache has been considered as 4-way set associative.

adpcm	Adaptative Differential Pulse Code Modulation
crc	Cyclic Redundancy Check
compress	Data compression
matmul	Matrix multiplication
nsischneu	Simulation of a Petri net
seg1, seg2, seg3	Image segmentation (3 steps)
airbag	Airbag control software

Table 2. Set of benchmarks.

4.5. Code compression

Code compression is parameterized by the proportion of the dictionary that is built from dynamic instruction profiles instead of static code information. In a preliminary study, we have found that, for most of the benchmarks, a value of P=75% limits the degradation of the reduction in code size (compared to $P=0$) while increasing the quantity of compressed code fetched into the cache at runtime (which is maximum when $P=100\%$). This way, code compression is effective while also improving the execution time and the energy consumption. This is why we considered $P=75\%$ in our experiments.

5. Experimental results

5.1. Impact of code compression on the code size and on the observed execution time

Let us first examine how the code compression scheme is efficient in reducing the code size. Table 3 gives the compression rate for all the benchmarks considered in this paper: the first column indicates the raw compression rate of the text section while the second column accounts for the dictionary data into the compressed code size (these data might be included in the executable file and loaded into the dictionary before starting the execution). Since we do not analyze the execution time of the whole codes, but only that of the main function, we report in Table 4 the code size reduction of this function (this ignores the prologue and epilogue as well as unreached library functions). Sizes are given in bytes.

Benchmark	Compression rate	Compression rate including dictionary cost
adpcm	19.2%	9.5%
crc	24.5%	10.4%
compress	22.1%	10.1%
nsischneu	29.1%	18.4%
seg(1,2,3)	18.5%	11.3%
airbag	31.8%	25.1%

Table 3. Code size reduction of the whole benchmarks

Benchmark	Size of original code	Size of compressed code	Compression rate
adpcm	4 040	3 164	21.7%
crc	656	308	53.0%
compress	1 820	1 212	33.4%
nsischneu	3 092	1 744	43.6%
seg1	1 052	1 020	3.0%
seg2	1 600	1 564	2.3%
seg3	972	932	4.1%
airbag	9 076	5 196	42.8%

Table 4. Code size reduction of the analyzed functions

Now, as said before, the impact of code compression on the execution time is hard to predict because the penalty due to the decompression scheme might be balanced by the gain due to a lower number of accesses to the instruction cache. Figure 5 shows the variation in the observed execution time when the code is compressed. The different sets of bars relate to different cache configurations. For most of the benchmarks, the execution time is sometimes noticeably decreased, in particular for small caches and small cache lines.

These results can be explained considering the impact of code compression on the number of instruction cache misses per instructions. For some of the benchmarks (adpcm, seg1, seg2 and seg3) and cache configurations (larger than 512 bytes for crc, compress and nsischneu), the number of misses per executed instruction in the original code is very low, as shown in Table 5. This means that cache misses do not contribute much to the execution time. As a consequence, the reduction in cache misses due to code compression improves only slightly the execution time. Note that the execution time is even increased for seg3 with cache

configuration 32-128: this is due to the fact that the number of accesses to the cache is unexpectedly increased by code compression, which might be due to changes into the alignment of the code with respect to cache line boundaries.

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
adpcm	0.7%	0.7%	0.3%	0.4%	0.4%	0.2%
crc	7.9%	0.3%	0.2%	6.3%	0.2%	0.1%
compress	27.4%	1.9%	1.9%	18.6%	1.0%	1.0%
nsischneu	17.4%	4.5%	4.5%	16.9%	2.3%	2.3%
seg1	2.0%	1.4%	0.2%	1.2%	0.8%	0.1%
seg2	0.5%	0.2%	0.1%	1.0%	0.1%	0.0%
seg3	1.6%	0.5%	0.5%	6.1%	0.3%	0.3%
airbag	25.8%	7.9%	4.8%	5.8%	4.3%	2.5%

Table 5. Mean number of instruction cache misses per executed instruction in the original code

On the contrary, when the original code exhibits a significant number of cache misses per instruction (which is the case with 128-byte cache configurations for `airbag`, `crc`, `compress` and `nsischneu`), the decrease of the number brought by code compression is larger.

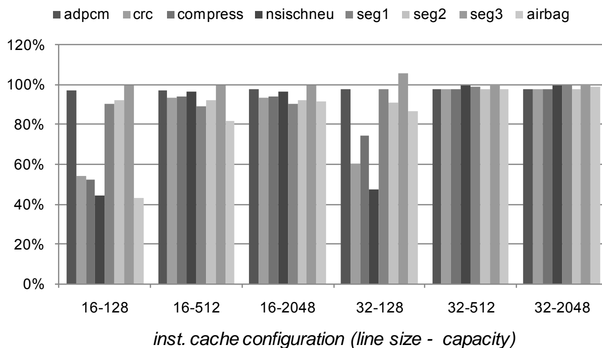


Figure 5. Impact of code compression on the observed execution time

The results above show that code compression, while mainly intended to reduce the code size, also improves the execution time in the average case. In the following section, we will check whether this is still true in the worst-case.

5.2. Impact of code compression on the Worst-Case Execution Time

The impact of code compression on the estimated Worst-Case Execution Time is shown in Figure 7 and Table 6 compares the average variation of the observed execution time (over all the benchmarks) to that of the WCET. On a mean, code compression improves the WCET less than the observed execution time for small caches and more than the observed execution time for larger caches. This respectively corresponds to a decrease or an increase of the WCET estimation accuracy. However, the impact in the WCET is noticeably different from one benchmark to the other one.

To validate the hypothesis that the lower improvement on the WCET than on the observed execution time is due to a loss of accuracy in the cache analysis, we have carried out some experiments considering perfect (always-hit) instructions caches. They showed that the WCET of the compressed code is almost the same as that of the original code, for every benchmark and cache configuration. This confirms that code compression sometimes impairs the instruction cache analysis.

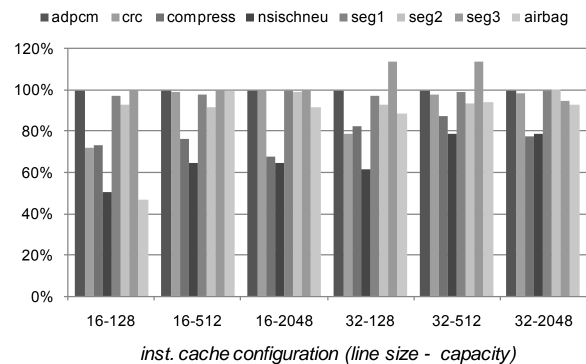


Figure 6. Impact of code compression on the WCET

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	-26.0%	-7.0%	-5.6%	-16.0%	-1.5%	-1.2%
WCET	-19.2%	-8.6%	-8.7%	-10.0%	-4.4%	-6.4%

Table 6. Impact of code compression on the mean observed and worst-case execution times

As mentioned in Section 3.3, code compression has an impact on the profiles of *l*-blocks. The curves in Figure 7 show that the rate of *partial l*-blocks is significantly increased for most of the benchmarks. The increase is greater with 16-byte cache lines because the proportion of partial *l*-blocks is already high in the original code with 32-byte lines (many basic blocks are shorter than 8 instructions).

Benchmarks that have few instruction cache misses per instruction do not see their estimated WCET much improved by code compression (in the same way as their observed execution time is not impacted). This is the case of `adpcm`, `seg1`, `seg2` and `seg3`.

Other benchmarks, like `compress` and `nsischneu` have their estimated WCET improved by code compression while their observed execution was not impacted (larger cache configurations). A look at the *l*-block categories for `nsischneu` reveals that the number of *Always Miss l*-blocks is cut by about 45% in the compressed code compared to the original code, which reflects that the spatial locality is improved. At the same time, the number of *Not Classified l*-blocks is cut by 50% to 70% (depending on the cache configuration) and this significantly helps the accuracy of WCET estimation. This explanation also holds for `compress`.

Finally, two benchmarks, `crc` and `airbag`, exhibit a reduction of their WCET mainly for small cache configurations.

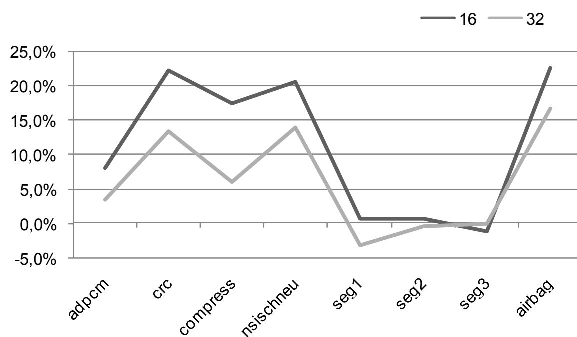


Figure 7. Impact of code compression on the number of *partial l*-blocks

To sum up, our code compression scheme tends to improve the accuracy of WCET estimates through an increased precision of the cache analysis. It can then beneficially be implemented in systems subjected both to time and memory size constraints.

6. Conclusion

Embedded systems often have to meet constraints of different nature: time deadlines for real-time applications, limitation of code size related to low memory capacity and restrictions on energy consumption imposed by requirements on autonomy and low power dissipation.

The goal of the MORE project is to develop a framework to optimize embedded software with respect to two or three of these criteria (WCET, code size and energy consumption) at the same time.

In this paper, we focus on the impact of code compression, techniques used to reduce the code size on the Worst-Case Execution Time.

The impact of code compression on the average-case execution time could be two-edged: on one side, the number of accesses to the instruction cache and the number of cache misses are likely to be reduced; on the other side, the overhead of decompression might increase the execution time. Thanks to the use of an in-pipeline decompression engine, the decompression time penalty is hidden by pipelined execution. This is why experiments show an improvement of the observed execution time besides to the reduction of the code size.

The impact on the Worst-Case Execution Time is more difficult to predict: it is expected that the decompression overhead would not have more impact on the WCET than on the observed execution time. But the changes in the placement of code in memory engendered by code compression are likely to impact the results of the cache analysis. This is confirmed by our experiments that show that the profile of *l*-blocks is modified (the proportion of partial *l*-blocks, shared by several basic blocks, is greater in the compressed code) and that the distribution of *l*-block categories is changed. The result is, in most of the cases, an improvement of the WCET that is more significant than that of the average-case execution time. In other words, the impact of the code compression on the statistics of the *l*-blocks translates into a more accurate cache analysis and then more accurate WCET estimates.

These results show that code compression can be used in real-time critical systems without negative impact on Worst-Case Execution Times.

As future work, we plan to study how WCET-related information could be used in addition to static and dynamic information within the compression process to increase the accuracy of the cache analysis. This might help to improve further the WCET estimates.

Acknlowledgements

The authors would like to thank the ANR French Research Agency for financial support of the MORE project.

References

[1] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, “Cache behavior prediction by abstract interpretation”, *Static Analysis Symposium (SAS)*, 1996.

[2] P. Altenbernd, “On the false path problem in hard real-time programs”, *8th Euromicro Workshop on Real-Time Systems,* 1996

[3] C. Ballabriga, H. Cassé, “Improving the First-Miss Computation in Set-Associative Instruction Caches”, *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[4] L. Benini, F. Menichelli, and M. Olivieri. “A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems”. *IEEE Transaction on Computers, 54(4)*, 2004.

[5] A. Beszedes, R. Ferenc, T. Gyimothy, A. Dolen and K. Karsisto, “Survey of code-size reduction methods”, *ACM Computing Survey, 35(3)*, 2003.

[6] H. Cassé, P. Sainrat, “OTAWA, a framework for experimenting WCET computations”, *3rd European Congress on Embedded Real-Time Software*, 2006.

[7] M. L. Corliss, E. C. Lewis, A. Roth, “The implementation and evaluation of dynamic code decompression using DISE”, *ACM Trans. Embedded Comput. Syst. 4(1)*, 2005.

[8] P. Cousot, R. Cousot, “Static determination of dynamic properties of programs”, *2nd International Symposium on Programming*, 1976.

[9] M. De Michiel, A. Bonenfant, H. Cassé, P. Sainrat, "Static loop bound analysis of C programs based on flow analysis and abstract interpretation”, *IEEE Int’l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[10] C. Ferdinand, F. Martin, R. Wilhelm, “Applying Compiler Techniques to Cache Behavior Prediction”, *ACM SIGPLAN Workshop on Languages, Compilers and Tool Support for Real-Time Systems,* 1997

[11] L. Goudge, S. Segars, “Thumb: Reducing the cost of 32-bit RISC performance in portable and consumer application”, *Proceedings of COMPCON6*, 1996.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench: a free, commercially representative embedded benchmark suite”, IEEE Workshop on Workload Characterisation, 2001.

[13] D. Hardy, I. Puaut, “WCET analysis of multi-level non-inclusive set-associative instruction caches”, *29th IEEE Real-Time Systems Symposium*, 2008

[14] J. Henkel, H. Lekatsas, V. Jakkula,. “Design of an one-cycle decompression hardware for performance increase in embedded systems”, *ACM Design Automation Conference (DAC),* 2002.

[15] N. Holsti, “Analysing Switch-Case Tables by Partial Evaluation”, *7th Workshop on WCET Analysis*, 2007.

[16] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, D. J. Auerbach, “A decompression core for PowerPC”, *IBM J. Res. Dev., 42(6)*, November 1998

[17] S. Lee, J. Lee, C.Y. Park, S. L. Min, “A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor”, *Workshop on Software and Compilers for Embedded Systems*, *LNCS 3199*, 2004.

[18] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.

[19] 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.

[20] X. Li, A. Roychoudhury, T. Mitra, “Modeling out-of-order processors for WCET analysis”, Real-Time Systems, 34(3), 2006

[21] E. W. Netto, R. Azevedo, P. Centoducatte, G. Araujo, “Multi-profile based code compression”, *ACM Design Automation Conference (DAC)*, 2004.

[22] C. Rochange, P. Sainrat, “A Context-Parameterized Model for Static Analysis of Execution Times”, *Trans. on High-Performance Embedded Architectures and Compilers*, 2(3), Springer, 2007.

[23] S. Thesing, *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, PhD thesis, Universität des Saarlandes, 2004.

[24] M. Thuresson, M. Själander, P. Stenström, “A Flexible Code Compression Scheme Using Partitioned Look-Up Table”, *HiPEAC Conference*, 2009.

[25] K. Watanabe, Y. Umezawa, “Optimal Triggering Of An Airbag”, *Intelligent Vehicles ‘93 Symposium*, 1993.

[26] WCET project / Benchmarks. Mälardaalen University. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Network

QoS-aware Routing for Real-Time and Multimedia Applications in Mobile Ad Hoc Networks

David Espes, Zoubir Mammeri
IRIT – Paul Sabatier University
Toulouse, France
espes@irit.fr, mammeri@irit.fr

Abstract

With the increasing development of real-time and multimedia applications, there is a need to provide bandwidth and delay guarantees. Most of QoS ad hoc network routing protocols select path guaranteeing delay and/or bandwidth. However, they don't consider throughput optimization, which results in a low number of admitted real-time and multimedia flows. In this paper, we propose a cross-layer TDMA-based routing protocol to meet delay and bandwidth requirements while optimizing network throughput. Since in TDMA-based ad hoc networks, slot reservation impacts two-hops neighbors, our routing protocol selects paths with the lowest number of neighbors. To show the effectiveness of our protocol, we present simulations using NS-2.

1. Introduction

With the continuously growing wireless technologies, mobile Ad hoc networks (MANETs) have emerged as a popular area of research. Recent growing interest in using MANETs to support real-time and multimedia applications led to the need to consider QoS support. One of the key issues to provide QoS guarantees in MANETs is routing.

Most routing protocols for MANETs, such as AODV [1], OLSR [2], DSR [3], are designed without explicitly considering QoS of the routes (also called paths) they select. Hop number is the most common criterion adopted by such routing protocols. It is becoming increasingly clear that such routing protocols are inadequate for real-time and multimedia applications, such as installation/environment monitoring and video conferencing, which often require QoS guarantees. QoS routing must find a path -from source to destination- which meets QoS requirements. In conventional wired networks, QoS support is easier to provide than in wireless networks. Moreover, the unpredictable and potentially rapid changes in routes and bandwidth availability are some significant challenges which need to be addressed before QoS techniques can be deployed in MANETs.

In spite of these difficulties, some QoS routing protocols in MANETs have been proposed, such as QoS-AODV [4], ODQOS [5], ADQR [6], QuART [7], MSMR [8], QoS-ASR [9], TDR [10], TBP [11], QRMP [12], QuaSAR [13], AQOR [14] and LAOR [15]. These protocols provide reactive routing, where control (i.e. routing) packets are only transmitted when important events occur such as route creation or route breakage. Almost all these protocols use slot reservation techniques during the creation route phase. None optimize the network bandwidth. They consider bandwidth constraints (eg. ADQR and ODQOS), delay constraints (eg. MSMR and LAOR) or both (eg. QoS-AODV), but don't meet these constraints while optimizing the network throughput.

We propose a reactive routing protocol, which provides bandwidth and delay constraints. The basic idea of our protocol to optimize network throughput is to minimize the number of neighbors associated with paths. Selecting paths with a low number of flows on neighboring nodes results in fewer collisions thus in more available slots to be used by nodes to establish real-time connections.

The rest of the paper is organized as follows. Section 2 is an overview of related work. Section 3 analyzes how time slots allocated to a flow may impact network throughput. Section 4 presents our routing protocol. Section 5 presents simulation results. Finally, we conclude the paper in section 6.

2. Related work

Providing QoS guarantees in MANETs is a challenge. Indeed node movement (i.e. network topology changes), low bandwidth, interferences and

collisions, make it very difficult to meet QoS constraints imposed by real-time and multimedia applications.

For collision avoidance, QoS routing protocols may use MAC protocols with no contention such as TDMA or CDMA-over-TDMA. In TDMA-based MANETs [16], nodes use their reserved slots to transmit data without collisions. Using contention-free MAC protocols, QoS routing protocols may easily provide some QoS guarantees in terms of bandwidth, delay, and jitter.

Other routing protocols may provide QoS guarantees even over contention MAC protocols. However, they only provide soft QoS guarantees. Consequently, observed QoS metrics (eg. delay or bandwidth) may exceed those bounds required by real-time and multimedia applications.

The following is a brief introduction to the most known and innovative routing protocols which provide bandwidth and/or delay guarantees. QoS-AODV [4], QoS-ASR [9], TDR [10], and AQOR [14] are reactive routing protocols, which provide bandwidth and delay guarantees

QoS-AODV forwards route search request only if the path meets bandwidth constraint and has a delay lower than the one of already received requests (if any). This protocol setups slot allocation when the source receives the route acknowledgement.

ODQOS [5] is a TDMA-based reactive routing protocol. It selects path to the destination with the minimum delay (or hops if the delay is the same for all paths). During the route search phase, all nodes, which receive a route request, reserve appropriate free slots. During the route acknowledgement phase, nodes that aren't on the selected path release reserved slots.

ADQR [6] is a multiple disjoint path reactive routing protocol. During route search phase, when a node receives a request, it forwards it only if the route is disjoint with previously received requests and the bandwidth requirements are met. Periodically, nodes transmit Hello packets. Neighbors determine signal strength and stability of the sender node. Source node selects the path with the highest stability. Resource reservation is done once the source node has selected the path.

QuART [7] is a reactive routing protocol, which selects routes with available bandwidth higher than required bandwidth. To correctly estimate the available bandwidth, route selection takes into account the potential interferences. Periodically, nodes send packets with their available bandwidth. When nodes receive these packets, they determine, according to the signal strength, if the sender is in the interference area or in the transmission area.

TBP [11] is another reactive routing protocol. It uses tickets to find route with QoS. Two types of tickets are used: yellow and green. A yellow ticket indicates a preference for paths with shorter delay. A green ticket indicates preference for lower-cost paths. Three levels of path redundancy are provided in TBP.

To determine eligible path, QoS-ASR protocol [9] uses a weight function taking into account seven metrics. During route search phase, nodes broadcast route request only if the sub-path meets the delay and bandwidth requirements and the path weight is less than a threshold.

With TDR protocol [10], each node sends periodically packets with its location and its mobility information. This protocol provides two methods to reroute packets when a breakage in the route is imminent. Nodes detect imminent breakage situations according to the signal strength of periodic packets.

AQOR [14] is IEEE 802.11 MAC based. Periodically, each node transmits Hello packets to inform its neighborhood about its available bandwidth. When a node receives a route request packet, it forwards the packet if it has sufficient available bandwidth.

None of the previous protocols optimize the network throughput. That is why, we propose a routing protocol to reduce time slot wasting due to the selection of paths including many neighbors.

3. Slot allocation impacts

In order to allocate the medium without collisions in the TDMA environment, the medium access time is divided into superframes. Each superframe is divided into control and data time slots. Each node is assigned a control time slot it uses to transmit its control information. The rest of the superframe is used for data transfer. Nodes must compete to reserve time slots.

A time slot s is considered free and may be allocated to send data from a node x to a node y if the following conditions hold [17]:

1) Slot s is not scheduled for receiving or transmitting in both nodes.

2) Slot s is not scheduled for receiving in any node z which is a 1-hop neighbor of node x.

3) Slot s is not scheduled for sending in any node z which is a 1-hop neighbor of node y.

When time slots are allocated on a link (x, y), 1-hop neighbours cannot use them, otherwise interferences may occur. Allocated time slots on a link impact nodes

of this link but also their neighbours. The higher the neighbour number is, the more important the impact of slot allocation is. “Time slot allocation impact” means how allocation of some time slots to support a flow f may prevent nodes to send or receive data packets other than flow f packets. Decreasing the number of free slots results in a decrease of either the bandwidth assigned to nodes or the number of admitted flows.

Slot allocation impacts two subsets of nodes: nodes forwarding the data packet of the new flow (i.e. nodes forming the new path) and their neighbour nodes. When slots are allocated on a link $<x, y>$:

- the previous hop of x doesn’t receive data in these slots and the next hop of y can’t send data in these slots to avoid interferences,

- all nodes in the neighbourhood of the sender can’t receive data and all nodes in the receiver neighbourhood can’t send data.

Consequently, it is of paramount importance not only to reduce the number of hops in a path but to select nodes such that the number of impacted neighbours is as low as possible.

The number of allocated slots takes into account the number of hops. In a path, an intermediary node receives data and relays them to next hop. So, it needs to reserve slots for reception and other slots for transmission. Thus, given a flow that requires k slots, each intermediary should be allocated $2k$ slots. Source (respectively destination) node should reserve k slots for transmission (respectively for reception).

The amount of time slots allocated for flows is given by theorem 1.

Theorem 1*: given a flow with k-slot requirements forwarded via a path* $P=<v_1,\ldots,v_N>$, *the amount of time slots allocated to such a flow is* $SA(P) = 2k (N-1)$.

When a node reserves bandwidth, the higher the number of neighbors is, the lower the network throughput is. Consequently, QoS-aware routing protocols should select paths with the lowest impact on the network, thus enabling the admission of more flows and/or flows with high bandwidth requirements.

The impact of slot allocation is given by theorem 2. A time slot at a node j is impacted by a node i (which relays flow f packets) if such a slot can’t be used to send or receive data to avoid interferences between nodes i and j. Let $SR(P)$ denote slots reserved for a flow f crossing path P by the number of slots impacted by the flow f. Theorem 2 gives the number of slots reserved by a flow f.

Theorem 2*: given a flow with k-slot requirements forwarded via a path* $P=<v_1,\ldots,v_n>$, *the flow impact on the neighborhood of path P, denoted SR(P), is :*

$$SR(P) \le k(N_1 - 1) + \sum_{i=2}^{n-1} 2k(N_i - 1) + k(N_n - 1)$$

where N_i *is the number of neighbors of node* v_i.

As shown by lemma 1, the impact of time slot allocation for a flow is derived from theorems 1 and 2. Lemma 1 provides a bound on the number of slots impacted by a flow f.

Lemma 1: *given a flow with k-slot requirements forwarded via a path* $P=<v_1,\ldots,v_n>$, *the flow impact on the overall network, denoted SI(P), is:*

$$SI(P) \le kN_1 + \sum_{i=2}^{n-1} 2kN_i + kN_n$$

4. Routing Protocol

4.1 Routing problem statement

Routing problem we are considering is denoted DBCONT (Delay and Bandwidth Constrained Optimal Network Throughput) routing.

Using Lemma 1, the optimal routing protocol which solves the DBCONT problem is defined as follows: *Given a source s and destination d, the optimal routing protocol is the protocol that returns a path* $P \in \pi(s,d)$ *such that P meets bandwidth and delay requirements and* $\forall P' \in \pi(s,d) \Rightarrow SI(P) \le SI(P')$. $\pi(s,d)$ *is the set of path between s and d.*

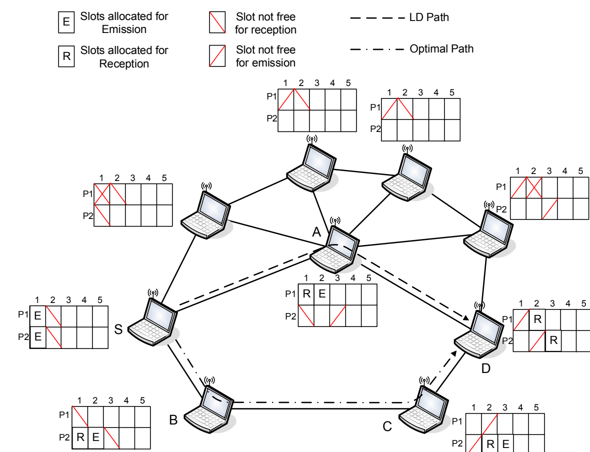


Figure 1. Impacts of paths selected by LD and optimal routing protocols

Figure 1 compares the Least Delay (LD) routing protocol to the optimal routing protocol. It considers a flow between nodes *S* and *D* that requires one time slot. The optimal routing protocol selects path P_1 whereas the LD routing protocol selects path P_2. Using path P_1, the number of impacted slots is 16. For path P_2, the number of impacted slots is 14. So the optimal routing protocol yields a lower impact on the neighbourhood compared to another routing protocol.

The effectiveness of a path *P* may be measured by means of impacted bandwidth, denoted *BI(P)*, which represents the bandwidth made unavailable because of slot allocation impact:

$$BI(P) = \frac{1}{T} SI(P) T_s C$$

where *T* is the TDMA superframe duration, T_s the slot duration and *C* the link capacity.

4.2. Overview of proposed routing protocol

Our protocol is an extension to the well-known AODV protocol. It relies on two procedures: route discovery and route maintenance. During the route discovery, it uses a weight function to determine the best path. It is loop-free.

Route discovery and maintenance procedures use three metrics for each path: end-to-end delay and bandwidth and the number of neighbors of all the nodes included in the path. These metrics are updated according to information captured at link layer (i.e. delay, bandwidth, and neighbors of each link forming the path).

Each node maintains two tables: a routing table and a reverse routing table. Routing table keeps information to reach the destination: source node, destination node, next hop, source sequence number, bandwidth, and delay requirements. Reverse routing table keeps information to forward the route confirmation from the destination to the source: source node, destination node, source sequence number, sub-path weight and previous node.

4.3 Weight function

To enable selection of the best path, intermediate nodes compute a cost function to decrease the impact of paths on the network. Path selection must meet the delay requirements and minimize the neighbor number. To minimize the latter, the path weight function penalizes paths with higher neighbor number and lower delay and privileges paths with higher delay and lower neighbor number.

The path with the lowest weight is selected by the destination. The weight function of path *P* is given by the formula (1):

$$w(P) = \begin{cases} \log\left(\dfrac{1}{1 - \dfrac{D(P)}{D_{e2e} + \varepsilon}}\right) \sum_{i=1}^{n-1} N_i & \text{if } D(P) < D_{e2e} + \varepsilon \wedge AS(i, i+1) \geq B_{e2e} \\ \infty & \text{else} \end{cases} \quad (1)$$

where D_{e2e} is the delay constraint, *D(P)* the path delay, *AS(i,i+1)* the available slots on link <*i*,*i*+1> which are the intersection between the slots available for transmission of *i* and the slots available for reception of *i*+1, B_{e2e} the bandwidth constraint and N_i *the number of node i neighbors.*

Notice that $w(P) \to \infty$ when $D(P) > D_{e2e} + \varepsilon$ and $w(P)=0$ when $D(P)=0$.

4.4 Route construction phase

This procedure is a modification to the one used in AODV. First, new fields are added in the route request packet (RREQ): bandwidth and delay requirements, sub-path neighbor number, sub-path delay, and time slot list. Moreover, according to node position along the path, three different algorithms may be executed as explained below.

1) Source node algorithm

The source node first checks its bandwidth availability. If there are sufficient free time slots at source node, the source sends a RREQ packet. If no response is received within a fixed time, the source node resends (a maximum number of RREQ retransmission is checked before retransmitting) another RREQ packet. Upon receiving a response packet (RREP), the path is setup. Then, the source node allocates time slots before starting data packet transmission.

2) Intermediate node algorithm

Upon receiving a RREQ packet, each intermediate node forwards such a request if it meets the QoS constraints (figure 2). Intermediate node checks if the route included in the request is better than previously received request for the same couple of source and destination nodes. The node updates the reverse path and inserts its transmission-free slots and its Id in the request if the path weight (given by formula 1) is better than the already known path weight and if it has sufficient free time slots to fulfill QoS constraints included in the received request. If both checks are

positive, the modified request is broadcast. Whenever an intermediary node receives a RREP packet, it allocates time slots according to the slot list included in RREP packet, and forwards it to the previous node on the reverse path.

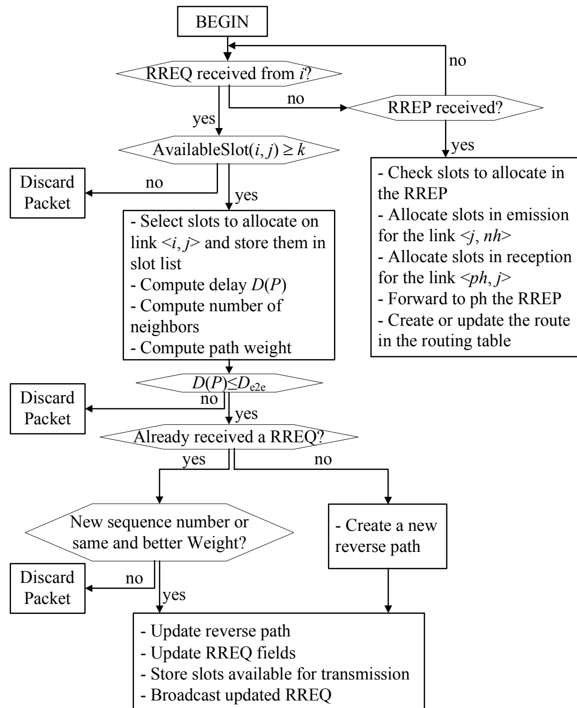


Figure 2. Intermediate node algorithm executed at node *j*

3) Destination node algorithm

Destination node algorithm is shown on figure 3. For each received RREQ packet, the total cost of the path is computed by the destination node. The latter maintains a timer for waiting RREQ packets. When the timer expires, the destination node selects the least-cost path. Then, it sends towards the source node a route reply packet (RREP) carrying the list of slots to reserve for the selected path.

4.5 Route maintenance

Node mobility may result in route broking, and consequently in degradation (loss) of QoS. Thus, route maintenance is of paramount importance for QoS routing in MANETs. We propose a simple route maintenance method. In case of node movement, broken route is detected by the upstream node (closer to source), e.g. assume the upstream node i sends a packet to node i+1. Node i will assume the route broken if it does not hear any transmission from node i+1 for a certain time. If the existing QoS route is broken, the upstream node on the route will send a RERR packet to the source. When an intermediary node receives the RERR packet it releases slots allocated for the broken flow. Downstream nodes release the slots when the connection timer expires (a timer is associated with each allocated slot and it is reset each time a packet is sent). When the source receives this packet it will start a new route discovery process.

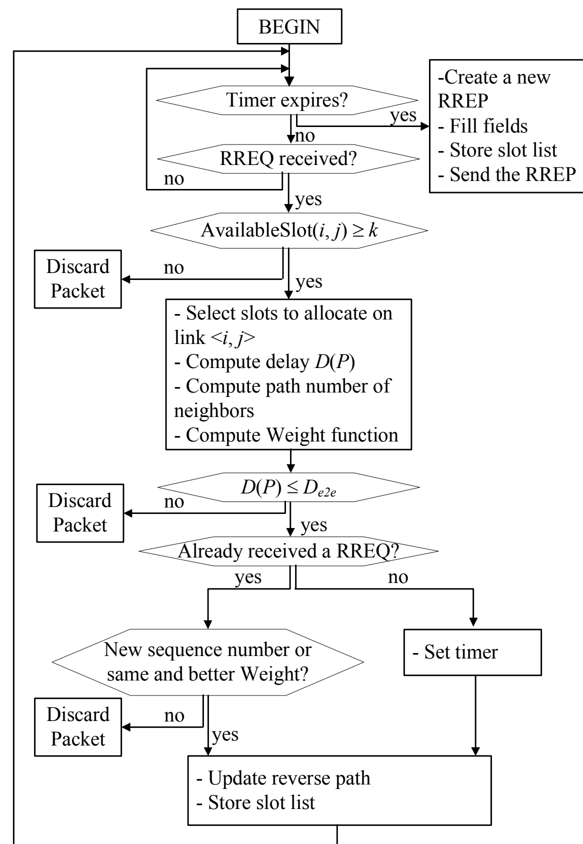


Figure 3. Destination node algorithm

5. Simulation

5.1 Simulation model

To assess the performance of our routing protocol, we conducted intensive simulation using network simulator NS-2. To analyze a realistic network model, we designed a program which randomly places M nodes on a 1000m*1000m plan.

The chosen node range is 150 meters. Link capacity is 11 Mb/s. The underlying MAC protocol is TDMA. There are 5 TDMA superframes. Each superframe is composed of 350 time slots. Each slot enables the transmission of a 500-byte packet. Since control slots are used either to send routing packets or TDMA control packets, the data slots is 350 - *M*.

Simulations use a communication model in which the half of nodes establish connections with the nodes of the other half. The traffic is CBR. The data packet length is 500 bytes. Each flow requires 20 kb/s. The simulation duration is 500 sec, and the flows start randomly in [0 .. 500 sec].

For each simulation run, we use 20 snapshots composed of different topologies with their traffic patterns. The reported results are the averages of 20 snapshot results.

We compare our algorithm with QoS-AODV and AODV protocols. QoS-AODV protocol returns the lowest delay path (LD path). Nodes forward RREQ packets only if the sub-path has a better delay than the previously stored sub-path associated with the same couple of source and destination nodes.

QoS-AODV and our protocol include slot reservation mechanism. For fair comparison between our protocol and AODV (which does not assume any reservation mechanism, as it is a best effort protocol), our simulation model is based on the following: once an AODV route is found, a procedure is undertaken to reserve slots along the route. If such a procedure succeeds, the flow is started. Otherwise, the route is rejected, a new attempt is made (no more than three reservation attempts are made).

5.2 Result discussion

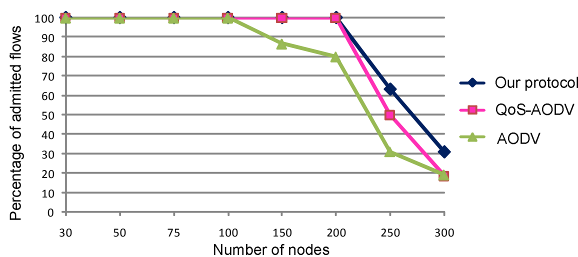


Figure 4. Percentage of admitted flows

When the number of nodes is higher than 100, AODV protocol results in more selected routes than the other protocols, because AODV does not check bandwidth availability along the selected routes. Once AODV has found a route, we use a procedure to reserve bandwidth. However, such a procedure may fail in reserving slots on the selected route when the traffic is high. Consequently the route selected by AODV is rejected. Above 200 nodes, QoS-AODV and our protocol may fail in finding routes. However, our protocol allocates up to 20% routes more than QoS-AODV at high load. Around 300 node density, QoS-AODV and AODV experience similar performance.

Our protocol weight function is efficient since it enables to select paths with a low number of neighbor nodes. QoS-AODV protocol doesn't optimize the network throughput. It only quickly returns a path which guarantees bandwidth and delay requirements.

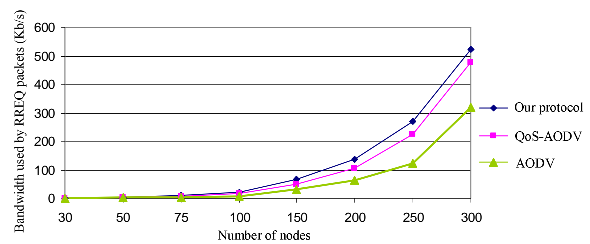


Figure 5. Routing packets sent to obtain paths

Figure 5 shows the overhead (in terms of routing packets) to obtain routes. The number of RREQ packets increases with the number of nodes of the scenario. Recall that the number of flows is the half of node number. After three failures in finding a route, the source stops sending RREQ packets. Route discovery failures increase the overhead of routing protocols because several attempts are needed to detect that no path meets QoS constraints.

More RREQ packets are sent by our protocol because its weight function takes into account not only the delay but also the number of neighbors.

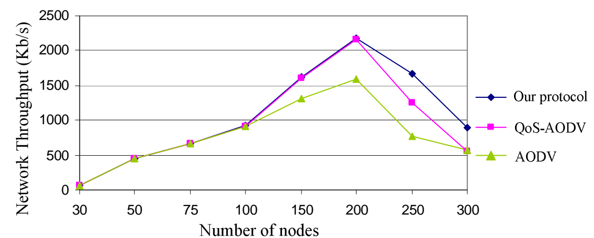


Figure 6. Network throughput

Figure 6 shows the network throughput, which is the bandwidth used by packets correctly sent.

When the number of nodes is less than 150, all flows can reserve slots. Consequently, the network throughput is the same for all the considered protocols.

When the number of nodes is greater than 100 some nodes have no available slots to establish new connections. AODV protocol returns paths but which do not meet QoS requirements because AODV does not check resource availability. In this case, AODV throughput is lower than the one of the other protocols.

Above 200 nodes, the flow number increases and thus the number of data slots decreases. For example, at 250 nodes, only 100 slots are allocated to data packets while there are 125 flows. All flows cannot meet their bandwidth requirements. In such a case, the network throughput decreases because a few flows are admitted in the network. When the network load is high, our protocol is more efficient since the bandwidth is less impacted compared to QoS-AODV. Our protocol enables more admitted flows than QoS-AODV.

6. Conclusions

In this paper, we present the importance of QoS routing in Ad hoc mobile networks, the challenges we tackle, and the approach we take. We discuss our extension to AODV protocol to provide QoS support. We propose a QoS routing protocol to be used in TDMA-based MANETs. Our protocol selects paths with a low impact on the network. Decreasing the impact (i.e. the amount of bandwidth consumed by admitted flows) of flows results in more accepted admitted flows and/or more bandwidth used by established flows.

To show the effectiveness of our protocol, we compare it to the well-known QoS-AODV and AODV protocols. From a performance point of view, our protocol has less impact on the network than the other protocols.

When the network load increases, our protocol provides a higher network throughput than other protocols. In such a case, more flows are admitted.

The improvement of network throughput comes with a cost. Our protocol has a higher overhead than QoS-AODV.

Finally, it should be noticed that our protocol is more scalable than QoS-AODV and AODV. It is particularly efficient in dense environments where MANET may be deployed.

7. References

[1] C. Perkins, E.M. Royer, S.R. Das, "Ad Hoc On-Demand Distance Vector routing", RFC 3561, July 2003

[2] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol", RFC 3626, October 2003.

[3] D. Johnson, Y. Hu, D. Maltz, The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4, RFC 4728, February 2007.

[4] C. Perkins, E. Belding-Royer, "Quality of Service for Ad-hoc On-demand Vector Routing", IETF Draft, October 2003.

[5] Y.-K. Ho , R.-S. Liu, "A Novel Routing Protocol for Supporting QoS for Ad Hoc Mobile Wireless Networks", Wireless Personal Communications Journal, v.22 n.3, p.359-385, 2002.

[6] Y. Hwang, P. Varshney, "An adaptive QoS routing protocol with dispersity for ad-hoc networks". 36th Annual Hawaii International Conference on System Sciences, January 2003, pp.302-311.

[7] T.S. Su, C.H. Lin, W. Hsieh, "A Novel QoS-Aware Routing for Ad Hoc Networks," 2006 Conference on Wireless Networks ICWN'06, June 2006, Las Vegas, USA.

[8] Y. S. Chen et al. "An on-Demand, Link-State, Multi-Path QoS Routing in a Wireless Mobile Ad-Hoc Network", Computer communication, 27(1):27-40, Jan, 2004.

[9] Labiod H., Quidelleur, "QoS-ASR: An Adaptive Source Routing Protocol with QoS Support in Multihop Mobile Wireless Networks". IEEE VTC'02, pp. 1978- 1982. 2002.

[10] S. De et al., "Trigger-Based Bistributed QoS Routing in Mobile Ad Hoc Networks," ACM Mobile Computing and Communications Review, 6(3):22--35, July 2002

[11] S. Chen, K. Nahrstedt, "Distributed Quality-of-Service Routing in Ad Hoc Networks", IEEE Journal on Selected Areas in Communications, pp. 1488-1504, August 1999.

[12] J. Wang, Y. Tang, S. Deng, J. Chen, "QoS Routing with Mobility Prediction in MANET", IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2001, PACRIM, pp. 357-360, August 2001.

[13] K.H. Vik, S. Medidi, "Quality of Service aware Source Initiated Ad-Hoc Routing", 1st IEEE International Conference on Sensor and Ad hoc Communications and Networks, Santa Clara, 2004

[14] Q. Xue,A. Ganz, "Ad hoc QoS on-Demand Routing (AQOR) in Mobile Ad Hoc Networks," Journal of Parallel and Distributed Computing, vol. 41, pp. 120-124, June 2003.

[15] J.H. Song et al, "Load-Aware On-Demand Routing (LAOR) Protocol for Mobile Ad Hoc Networks", IEEE VTC, April 2003, pp. 1753-1757.

[16] I. Jawhar, J. Wu, "QoS Support in TDMA-Based Mobile Ad Hoc Networks", J. Computer Science Technology, 20(6):797-810, 2005.

[17] W.H. Liao, Y.C. Tseng, K.P. Shih, "A TDMA-Based Bandwidth Reservation Protocol for QoS Routing in a Wireless Mobile Ad Hoc Network", IEEE International Conference on Communications, 2002.

Improvement of Schedulability Analysis with a Priority Share Policy in On-Chip Networks

Zheng Shi and Alan Burns
Real-Time Systems Research Group, Department of Computer Science
University of York, UK, YO10 5DD
{zheng, burns}@cs.york.ac.uk

Abstract

Priority-based wormhole switching with a priority share policy has been proposed as a possible solution for real-time on-chip communication. However, the blocking introduced by priority share complicates the analysis process. In this paper, we propose a new "per-priority" basis analysis scheme which computes the total time window at each priority level instead of each traffic-flow. By checking the release instance of each flow at the corresponding priority window, we can determine schedulability efficiently.

1 Introduction

On-chip networks (NoCs) [8, 3], have emerged as a new design paradigm to overcome the limitation of current bus-based communication infrastructure [9], and are increasingly importance in today's System-on-Chip (SoC) designs. The typical architecture of on-chip networks consists of multiple *intellectual property (IP)* modules connected through an interconnection network. This architecture offers a general and fixed communication platform which can be reused for a large number of SoC designs.

Multiple IP-cores based design using NoC allows multiple applications to run at the same time. These applications execute data processing and exchange information through the underlying communication infrastructure. Some applications have very stringent communication service requirements, the correctness relies on not only the communication result but also the completion time bound. A data packet received by a destination too late could be useless. These critical communications are called *real-time* communications. For a packet transmitted over the network, the communication duration is denoted by the *packet network latency.* The maximum acceptable duration is defined to be the *deadline* of the packet. A *traffic-flow* is a packet stream which traverses the same route from the source to the destination and requires the same grade of service along the path. For *hard real-time* traffic-flows, it is necessary that all the packets generated by the traffic-flow must be delivered before their deadlines even under worst case scenarios.

The on-chip network is a significant solution for complex communication of SoCs and outperforms the traditional busses or a point-to-point approach in many ways [8]. But it also introduces unpredictable network delay since the expensive hardware resources (e.g. link bandwidth and buffer space) are usually shared by a number of applications. When more than one packet tries to access the shared resource at the same time, contention occurs. The contention problem, which leads to packet delays and even missing deadlines, has become the major influence factor of network predictability. So how to solve contention problem is a key issue in implementing guaranteed performance service in NoC design.

Contention avoidance and contention acceptance are two basic approaches to address the contention problem. First approach considers that contention is avoidable by trying to pre-arrange and allocate resources before the start of the communication, so that two packets never access the same resource at the same time. Time division multiplexing (TDM) and circuit switching are two common contention avoidance schemes. In Ætheral [10] and Nostrum [15], the whole link transmission capacity is partitioned into fixed time-slots, each of which represents a unit of time when a single application can occupy this physical link exclusively for data transmission. But this scheme requires a global notion of time in the network. Besides that, the latency is coupled to bandwidth, preventing low latency from being provided to low rate requirement without over-allocating. A circuit-switching technique is used in [20, 19]. A dedicated connection is constructed between source and destination nodes by reserving a sequence of wiring resources. The major problem of this scheme is that the resources that have been reserved for a flow can not be used by any other flow which results in the under-utilized links. The contention avoidance policy requires the network resource to be configured before the communication which lacks flexibility and

wastes the links transmission capacity.

A contention acceptance policy normally utilizes an arbiter at running time. QNoC [6] divides network services into four levels and utilizes a priority arbiter to implement the differentiated services. But this scheme only offers the coarse-granularity service and does not seem to be suitable for hard real time application. Kavaldjiev *et al* [12] presented a simple round-robin arbiter to cater for the real time services. In Mango NoC project [5], a new arbiter is designed which combine round-robin and priority to bound latency and bandwidth. But both of them suffer the same problem as TDM that the latency is coupled with bandwidth. As a new solution, a priority based wormhole switching technique [17] is introduced to provide communication service guarantees. The hard timing bound is delivered by this approach with the support of a priority based router infrastructure which allocates each traffic-flow with a distinct priority and virtual channel independently. This scheme successfully overcomes the problem of latency coupled with bandwidth and thus supports a wide range of traffic types. Latency analysis and validation have been discussed in a number of papers [2, 11, 13, 14, 17]. But the drawback of the priority-based wormhole switching approach is precisely that the distinct priority per traffic-flow implementation policy results in higher area and energy overhead and hence limits its employment and development in on-chip networks. To solve this problem, Shi and Burns [18] proposed a priority share policy to reduce the resource overhead while still achieving hard real-time communication guarantees. The priority share policy permits multiple traffic-flows to contend for a single virtual channel and share the same priority level. In that paper [18], the authors also presented a composite model analysis scheme. But this approach requires that all the traffic-flows must meet the constrain that network latency is no more than period. This is a strong restriction. The more complex the system, with long communication delays over several hops, the greater the global delays will become.

In this paper, we propose a new analysis approach which can efficiently handle wormhole switching with a priority share policy. The new analysis is based on "per-priority basis", that is, it computes the total time window at each priority level instead of each traffic-flow. By checking the packet release instance of each traffic-flow at the corresponding priority window, we can verify the timing semantics of real time traffic-flow with a simple yet efficient mechanism. The deadline no more than period constraint is successfully removed by this approach with a low computational complexity. In addition, we also find that the previous result proposed in [18] is just a special case covered by this new analysis.

The rest of this paper is organized as follows: Section 2 introduces wormhole switching networks with a priority share policy. Section 3 describes the real-time communication model and notations used in this paper. A novel schedulability analysis technique and related example are presented in sections 4 and 5. Finally, section 6 concludes the paper.

2 Wormhole switching with priority share

2.1 Wormhole switching structure

Wormhole switching [16] is an increasingly common interconnect scheme for NoC as it minimizes communication latencies, requires small buffer and is simple to implement. Each packet in a wormhole network is divided into a number of fixed size flits [16]. The header flit takes the routing information and governs the route. As the header advances along the specified path, the remaining flits follow in a pipeline way. If the header flit encounters a link already in use, it is blocked until the link becomes available. In this situation, all the flits of the packet will remain in the routers along the path and only a small flits buffer is required in each router.

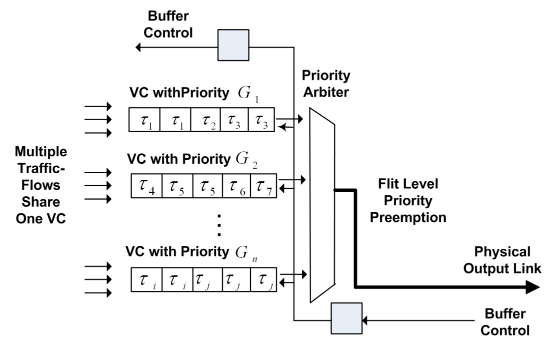


Figure 1. Output Arbitration with Priority Share

In order to ensure hard real-time service guarantees with limited resources, a priority share based flit-level arbitration structure is introduced [18], Figure 1 shows such a structure. There are a number of prioritized virtual channels [7] available at each router output port. The virtual channels (VCs) area resource allocation technique which provides multiple independent buffers for each physical link. Each of these buffers is considered as a virtual channel and can hold one or more flits of a packet. The credit-based flow control provides each virtual channel of each router with some credit, which is equal to the buffer size of that virtual channel of the subsequent router. The credit is decremented upon transmitting a flit and incremented upon receiving a buffer-free notification from the next router. A priority based arbitrator controls the access to the shared physical link for all the virtual channels. Since VCs are not mutually dependent on each other, the transmitting packet can bypass a blocked one through the different VCs. This strategy efficiently utilizes the network resource (link bandwidth) and improves the per-

formance with a very small buffer overhead [4].

Differing from previous works [11, 13, 17], the distinctive characteristic of the priority share scheme is that multiple traffic-flows per virtual channel are supported. These traffic-flows sharing the same virtual channel will be mapped to the same priority. Each packet generated by the traffic-flow inherits this priority. A packet with priority G_i can only request the virtual channels associated with priority G_i. At any time, a flit of a given packet will be sent out through its respective output port if it has the highest priority and it has credit(s). In addition, a higher priority packet can also preempt a lower priority packet during its transmission. As a hybrid solution, best-effort traffic-flows also can be multiplexed on the same links with lowest priority (any real time flow has higher priority than best-effort flows). In the case where no real time flow is available, best-effort flows make use of spare bandwidth.

2.2 The problem of blocking

By sharing priority, the hardware resource overhead can be reduced dramatically compared with the traditional distinct priority per traffic-flow scheme [11, 13, 17]. But on the other side, it may lead to significantly blocking and unpredictable network latency. Consider the fact that traffic-flows within the same virtual channel are served in first-in-first-out (FIFO) order because the priority preemption is only available between the different virtual channels. When a packet has to wait for the transmission of another packet (this packet can be released from the same flow or other flow) in the same buffer due to priority share, blocking occurs. Therefore, a packet can be blocked by every packet with the same priority which arrives just before it. Once a packet is blocked by another packet with the same priority which holds a virtual channel for a prolonged duration, it can block other packets, which can in turn block other packets, and so on.

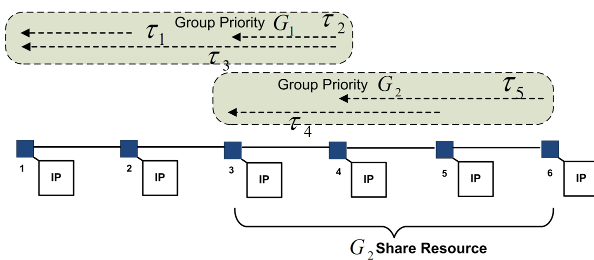


Figure 2. A Case of Traffic-flows with Priority Share

As a simple example to motivate the blocking problem, consider Figure 2, which illustrates a number of traffic-flows loaded on a NoC platform. Flows τ_1, τ_2 and τ_3 share the same priority G_1, τ_4 and τ_5 share the same priority G_2 and G_1 is higher a priority than G_2. We assume that a packet from τ_5 is released, because of priority share, τ_5 can be blocked by τ_4 if it arrives just after τ_4. During τ_4's transmission, it can be preempted by the packet releases from higher priority flows τ_2 and τ_3. Note that when τ_2 or τ_3 is active, τ_4's packet service will be suspended but will still occupy link resources. In this situation, only after τ_4's completion, can the packet from τ_5 resume its transmission service. So the interference suffered by τ_4 actually extends the possible completion time of τ_5. Besides that, flow τ_1 in this case also can introduce some interference (see [18]) which delays the network latency of τ_5 further. Eventhough flows τ_1, τ_2 and τ_3 never share any physical link with τ_5, they still can play a major role in determining τ_5's transmission latency. This phenomenon only exists when the network supports priority share. The latency analysis in this situation is very hard due to the complicated blocking inter-relationship between the flows.

To simplify the blocking problem, the analysis in [18] imposes a deadline no more than period restriction so that one traffic-flow can not be blocked by another flow with the same priority more than once; this is termed **single blocking**. With this property, all the flows sharing the same priority are transformed into a single scheduling unit and the maximum network latency is addressed by this new model. However, without this enforced constraint, we find two additional blocking phenomena may appear which also need to be considered.

Multiple blocking : A set of traffic-flows sharing the same priority; one could block another more than once. Figure 3(A) shows such a situation. The solid up arrow indicates the packet release instance. The packet's latency is depicted as horizontal arrow line. τ_a shares the same priority as τ_i, if the transmission latency of τ_i is bigger than the period of τ_a, τ_i could be blocked by τ_a more than once.

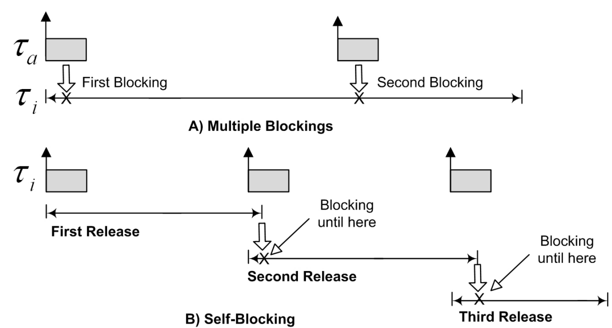


Figure 3. The Blocking Problem

Self-blocking : In a situation while the end flits from a previous packet are being delivered, the start flits of the next packet from the same flow are already introduced. Therefore, the possible blocking delay suffered by the new arrival packet

comes from not only the other flows with the same priority but also the flow itself, we refer to this as *self-blocking*. Figure 3(B) shows such a situation. In this example, the second packet released from τ_i is blocked by the first one until its completion. Similarly, the third packet is also blocked by the second one, etc.

3 System model and definition

3.1 Traffic-flow model

A wormhole switching real-time network Γ comprises n real-time traffic-flows $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$. Each traffic-flow τ_i has a set of properties and timing requirements which are characterized by six-tuple attributes $\tau_i = (G_i, C_i, T_i, D_i, J_i^R, J_i^I)$. We assume that all the traffic-flows which require timely delivery are periodic or sporadic. The lower bound interval on the time between releases of successive packets is called the period T_i for the traffic-flow τ_i. The maximum basic network latency C_i is the maximum duration of transmission latency when no traffic-flow contention exists [17]. Each real-time traffic-flow has relative deadline D_i which is the upper bound restriction of network latency. There is no restriction on the relationship between deadline D_i and period T_i. Any flow's deadline can be less than, equal to or greater than its period. J_i^R is the release jitter [1] and denotes the maximum deviation of successive packet releases from its period. If a packet from τ_i is generated at time a, then it will be released for transmission by time $a + J_i^R$ and have an absolute deadline of $a + D_i$. J_i^I is interference jitter [17] which denotes the maximum deviation of successive packets start transmission time. Besides these, each traffic-flow has a priority G_i. The value 1 denotes the highest priority and larger integers denote lower priorities. We assume the traffic-flow is prioritized by any possible priority assignment policy, e.g. a greedy priority allocation algorithm has been proposed in [18]. All the traffic-flows competing for the same virtual channel will be allocated to the same priority. Therefore, each priority level G_i can be regarded as a flow set denoted by $S(i)$. The priority should be assigned off-line and remain constant at run-time. We also define a functions $G(\tau_i)$ to obtain the corresponding priority for a traffic-flow τ_i. It follows that $\tau_i \in S(G(\tau_i))$.

3.2 Inter-relationships between traffic-flows

To capture the relations between traffic-flows and the physical links of the network, we formalize the mesh network topology defined as a directed graph $\mathbb{G} : V \times E$. V is a set, whose elements are called nodes, each node v_i denotes one router in the mesh network. E is a set of ordered pairs of vertices, called edges. An edge $e_{x,y} = \{v_x \rightarrow v_y\}$ is considered to be a real physical link from router v_x to router v_y; v_x is called the source and v_y is called the destination. We define a mapping space from the traffic-flow set to the physical links $\Gamma \rightarrow E$. Given a set of n traffic-flows Γ, we can map them to the target network. The routing $\Re_i$ of each traffic-flows τ_i is denoted by the ordered pairs of edges, $\Re_i = \{e_{1,2}, e_{2,3}, \ldots, e_{n-1,n}\}$. If a traffic-flow τ_i shares at least one link with τ_j, the intersection set between them is $\Re_i \cap \Re_j$. If $\Re_i \cap \Re_j = \emptyset$, τ_i and τ_j are disjoint.

Based on whether they share the same physical links, we introduce two different relationships between the traffic-flows: *direct competing relationship*, and *indirect competing relationship*. The direct competing relationship means a traffic-flow has at least one physical link in common with the observed traffic-flow. For any two traffic-flows τ_i and τ_j, if direct competing relationship exists between them, then $\Re_i \cap \Re_j \neq \emptyset$. In the indirect competing relationship, on the contrary, the two traffic-flows do not share any physical link but there is (are) intervening traffic-flow(s) between the given two traffic-flows. For example, if there are three flows τ_i, τ_j, τ_k meeting the following situation, $\Re_k \cap \Re_i = \emptyset$, $\Re_k \cap \Re_j \neq \emptyset$ and $\Re_i \cap \Re_j \neq \emptyset$; τ_j is the intervening flow in this case, then there is an indirect competing relationship between τ_i and τ_k. Notice that indirect competing has transitivity. If more than one intermediate flow exists, the indirect competing relationship still holds. Following the above case, if there is a new flow τ_a which has an indirect competing relationship with τ_j (τ_k is intermediate flow between τ_a and τ_j), and τ_a does not share any physical link with τ_i, then there is an indirect competing relationship between τ_a and τ_i (both τ_k and τ_j are intermediate flows). Figure 4 shows the situation.

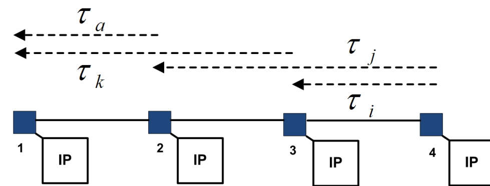


Figure 4. Transitivity in indirect competing relationship

Assuming that the priorities have been assigned to each traffic-flow, if a packet is released, the possible delays it suffered before completion consist of all the interferences from higher priority traffic-flows and the blocking from the traffic-flows with the same priority. Based on different priority levels and the competing relationships, we categorize the delays into four different types:

- **Direct interference from traffic-flow with higher priority**
  When two traffic-flows τ_i and τ_j have a direct compet-

ing relationship and meet the condition $G(\tau_j) > G(\tau_i)$, τ_j will force a direct interference with the observed traffic-flow τ_i. For flow τ_i, we define a direct interference set S_i^D which includes all the traffic-flows meeting the above conditions, $S_i^D = \{\tau_j | \Re_j \cap \Re_i \neq \emptyset$ *and* $G(\tau_j) > G(\tau_i)$ *for all* $\tau_j \in \Gamma\}$.

- **Indirect interference from traffic-flow with higher priority**
  When two traffic-flows τ_i and τ_k have an indirect competing relationship and meet the condition $G(\tau_k) \geq G(\tau_j) > G(\tau_i)$, τ_j is the intervening flow, τ_i may suffer an indirect interference from τ_k even when they do not share any physical link, see [17] for detailed description. For flow τ_i, we define an indirect interference set S_i^I which includes all the traffic-flows meeting the above conditions, $S_i^I = \{\tau_k | \tau_k$ has indirect competing relationship with τ_i *and* $G(\tau_k) \geq G(\tau_j) > G(\tau_i)$ *for all* $\tau_k \in \Gamma$, where τ_j is any intermediate flow$\}$.

- **Direct blocking from traffic-flow with same priority**
  When two traffic-flows τ_i and τ_j have the direct competing relationship and meet the condition $G(\tau_j) = G(\tau_i)$, if the packet from τ_j is release just before τ_i, τ_j will force a blocking with τ_i. For flow τ_i, we define a direct blocking set S_i^{SD} which includes all the traffic-flows meeting the above conditions, $S_i^{SD} = \{\tau_j | \Re_j \cap \Re_i \neq \emptyset$ *and* $G(\tau_j) = G(\tau_i)$ *for all* $\tau_j \in \Gamma\}$.

- **Indirect blocking from traffic-flow with same priority**
  When two traffic-flows τ_i and τ_k have an indirect competing relationship and meet the conditions $G(\tau_k) = G(\tau_j) = G(\tau_i)$, τ_j is the intervening flow, τ_i may suffer an indirect blocking from τ_k even they do not share any physical link. An indirect blocking example has been shown in Figure 2 where τ_1 blocks τ_3 and further blocks τ_2. For flow τ_i, we define an indirect blocking set S_i^{SI} which includes all the traffic-flows meeting the above conditions, $S_i^{SI} = \{\tau_k | \tau_k$ has indirect competing relationship with τ_i *and* $G(\tau_k) = G(\tau_j) = G(\tau_i)$ *for all* $\tau_k \in \Gamma$, where τ_j is any intermediate flow$\}$.

Note that the affect of the direct and indirect interferences has been presented in [17]. The priority share policy introduces the new direct and indirect blockings. Especially without constraint of D $\leq$ T, there are three different blocking relationships which severely complicate the analysis progress. So in this paper, we propose a new scheme which changes the analysis view from per flow to per priority. The detailed issues will be discussed in the next section.

Return to the example in Figure 2. Five traffic-flows τ_1, τ_2, τ_3, τ_4 and τ_5 are mapped into two sets, the set with priority G_1 includes τ_1, τ_2 and τ_3, $S(1) = \{\tau_1, \tau_2, \tau_3\}$, the set with priority G_2 includes τ_4 and τ_5, $S(2) = \{\tau_4, \tau_5\}$; $G_1 > G_2$ in this case. Traffic-flows τ_1, τ_2 and τ_3 have no shared links with any higher priority flow so no direct or direct interference. Due to sharing the same priority, the direct and indirect blocking set for τ_1, τ_2 and τ_3 are $S_1^{SD} = \{\tau_3\}$, $S_1^{SI} = \{\tau_2\}$, $S_2^{SD} = \{\tau_3\}$, $S_2^{SI} = \{\tau_1\}$, $S_3^{SD} = \{\tau_1, \tau_2\}$ and $S_3^{SI} = \emptyset$. Flow τ_4 directly competes with higher priority flows τ_2 and τ_3 and indirect suffers interference from τ_1, $S_4^D = \{\tau_2, \tau_3\}$, $S_4^I = \{\tau_1\}$. Flow τ_5 does not have any higher priority flow, so $S_5^D =$, $S_5^I = \emptyset$. Besides that, τ_4 and τ_5 share the same priority level, thus $S_4^{SD} = \{\tau_5\}$, $S_4^{SI} = \emptyset$, $S_5^{SD} = \{\tau_4\}$ and $S_5^{SI} = \emptyset$.

4 Network latency upper bound analysis

4.1 Priority window model

The correctness of the design and development of practical real-time applications in priority-based wormhole switching relies on efficient schedulability analysis. The schedulability test in this paper is based on the computation of the *worst case network latency* for each traffic-flow. If the worst case network latency, R, of a flow is no more than its deadline, $R \leq D$, then the traffic-flow is schedulable. If all the traffic-flows loaded on a network are schedulable, then the traffic-flow set is called schedulable.

The term *priority level-G_i shared resource* is introduced which denotes all the link resources required by the flows with the same priority G_i. This shared resource is modeled as a single competing unit. A *priority window* $W(i)$ is used to define a contiguous time interval during which this priority level-G_i shared resource keeps the network busy and serves all the traffic-flows of priority higher than or equal to the priority G_i. The priority window will continue until the time when the shared resource becomes idle, ready for the next transmission and yet there is no service requirement from priority level G_i or higher waiting to be transmitted. As shown in Figure 2, for the priority level G_2 shared resource, the links between router 3 and 6, the corresponding priority window is the contiguous time duration where the shared resource keeps serving all the queueing packets with priority G_2 or higher (G_1 in this case). Figure 5 illustrates a priority level-G_2 window. The bold circle denotes the time the packet is received completely at the destination node. In this example, the total window $W(2)$ at priority level-G_2 shared resource is the time span from the first release of τ_2 to the completion time of the second instance of τ_5.

For a set of traffic-flow $S(i)$ with the same system priority level G_i, next, we show how to compute the corresponding priority window $W(i)$.

Lemma 1. *The priority level-G_i window $W(i)$ upper bound can be calculated by the following relation:*

$$W(i) = E(i) + I(i) \tag{1}$$

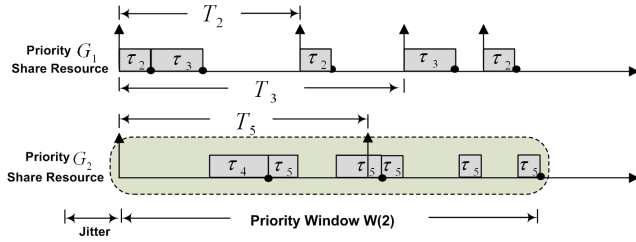


Figure 5. Priority Level-G_2 Window

where $E(i)$ denotes the summation of service requirements generated by all the traffic-flows with the priority G_i and $I(i)$ accounts all the interferences from the higher priority traffic-flows which contend the level-G_i share link resource during this window.

Proof. According to the definition of priority window, all the arrival packets of priority G_i or higher before the end of the priority window must be transmitted during the window. Besides that, any packet with priority lower than G_i is unable to delay current window. Therefore, the width of the priority window is equal to the time interval taken to serve the transmission requirements, $E(i)$, made by all the traffic-flows with the priority G_i and all the interferences, $I(i)$, from the higher priority traffic-flows which contend the level-G_i share link resource during this window.

□

The value $E(i)$ and $I(i)$ determine the priority window for the level-G_i shared resource. So if we can find an upper bound of $E(i)$ and $I(i)$, the maximum priority window $W(i)$ is then trivially computed. Note that, when we explain how to calculate the priority window for G_i priority level, we assume that analysis for all the higher priority $G_1, G_2, \ldots, G_{i-1}$ has been completed.

Theorem 1. *The maximum priority window $W(i)$ for priority level G_i share resource can be found by:*

$$W(i) = \sum_{\forall \tau_n \in S(i)} \lceil \frac{W(i) + J_n^R}{T_n} \rceil C_n + \sum_{\forall \tau_j \in hp(i)} \lceil \frac{W(i) + J_j^R + R_j - C_j}{T_j} \rceil C_j \quad (2)$$

where $hp(i)$ is the higher priority set, all the flows in S_i^D, $\tau_i \in S(i)$, are the members of the set $hp(i)$,

$$hp(i) = \bigcup_{\forall \tau_i \in S(i)} S_i^D \quad (3)$$

where $\bigcup$ is the union operation of the flow sets.

Proof. Supposing that we can find an upper bound of the total priority window $W(i)$, the maximum number of instances of a flow τ_n with priority G_i that can delay this window is computed as follows:

$$\lceil \frac{W(i) + J_n^R}{T_n} \rceil \quad (4)$$

assuming the worst case release scenario of τ_n: the first packet release starts J_n^R later than the first arrival, and the subsequent releases are maximum packet size with the maximum release rate of $1/T_n$. Consequently, the service requirement summation from all the flows in $S(i)$ is thus:

$$E(i) = \sum_{\forall \tau_n \in S(i)} \lceil \frac{W(i) + J_n^R}{T_n} \rceil C_n \quad (5)$$

On the other hand, the interferences produced by all the higher priority flows which compete the level-G_i shared resource also delay the corresponding priority window. The maximum interference analysis has been discussed in [18]. During any time interval, an upper bound of interference produced by any higher priority traffic-flow τ_j when interference jitter exists is:

$$\lceil \frac{W(i) + J_j^R + R_j - C_j}{T_j} \rceil C_j \quad (6)$$

where $R_j - C_j$ is the maximal possible jitter upper bound and R_j is the worst case latency of τ_j. Theorems 2 and 3 below will show that the network latency of τ_j can be found by calculating the corresponding priority window. The interference jitter phenomenon only happens when indirect higher priority flow exists. The paper [18] has discussed two possible conditions, $S_j^D \cap S_i^I \neq \emptyset$ or $S_j^{SD} \cap S_i^I \neq \emptyset$, where $\tau_i \in S(i)$ and $\tau_j \in S_i^D$, either of which can result in the interference jitter.

Any packet release from the higher priority flows which compete the priority level-G_i shared resource will finally extend the corresponding priority window $W(i)$. For convenience, we define a higher priority set $hp(i)$, all the flows in S_i^D, $\tau_i \in S(i)$ are inserted into set $hp(i)$. The maximum interferences produced by these higher priority flows can be computed as follows:

$$I(i) = \sum_{\tau_j \in hp(i)} \lceil \frac{W(i) + J_j^R + R_j - C_j}{T_j} \rceil C_j \quad (7)$$

Combining Eq.(1), Eq.(5) and Eq.(7) , an upper bound of priority level-G_i window in case of interference jitter and

release jitter is given by:

$$W(i) = \sum_{\forall \tau_n \in S(i)} \lceil \frac{W(i) + J_n^R}{T_n} \rceil C_n + \sum_{\forall \tau_j \in hp(i)} \lceil \frac{W(i) + J_j^R + R_j - C_j}{T_j} \rceil C_j \quad (8)$$

□

The result of $W(i)$ can be solved using the iterative technique [1]. The iteration starts with $W(i)^0 = \sum_{\forall \tau_n \in S(i)} C_n$ and terminates when $W(i)^n$ no longer increases, it has converged. By this iterative technique, the maximum priority window can be calculated ($W(i) = W(i)^{n+1} = W(i)^n$).

4.2 Maximum network latency

Based on the maximum priority window of G_i, the next step is how to find the maximum network latency for each flow in $S(i)$. For any observed flow τ_i, $\tau_i \in S(i)$, the maximum delay occurs when it is released with all the higher priority flows simultaneously and all the other flows sharing the same priority as τ_i start their services just before τ_i, this will produce the maximum service requirements on the share resource. So the earliest starting time of τ_i is the same as the priority level G_i window beginning. To calculate the worst case network latency, we need to find the latest completion time. Due to the multiple blocking and self-blocking problems, if more than one packet instance is released from the same flow during a priority level G_i window, then it is necessary to check these instances in order to find the overall worst case network latency of traffic-flow.

Motivated by the observation of the relation between priority window and period, we check the priority window at three different situations: $W(i) \leq min(T_n - J_n^R)$, $min(T_n - J_n^R) < W(i) \leq T_i - J_i^R$ and $W(i) > T_i - J_i^R$, (τ_n is any flow in $S(i)$) as showed in Figure 6.

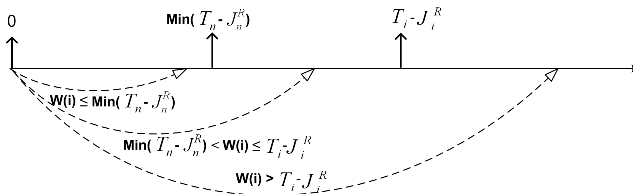


Figure 6. Three possible relations between priority window and period

Theorem 2. *The maximum network latency R_i for τ_i is given by:*

$$R_i = W(i) + J_i^R \quad (9)$$

when $W(i) \leq T_i - J_i^R$

Proof. The interval [0, $T_i - J_i^R$] is divided into two subintervals [0, $min(T_n - J_n^R)$] and ($min(T_n - J_n^R)$, $T_i - J_i^R$].

If condition $W(i) \leq min(T_n - J_n^R)$ for $\forall \tau_n \in S(i)$ is true, then the priority window ends at or before any repeated release from flow with priority G_i. This means that no traffic-flow can be blocked by any other flow sharing the same priority more than once. The multiple blocking and self blocking discussed in section 2.2 do not occur. In the worst case, τ_i will be the last flow getting transmission opportunity in this priority window. So the ending of the priority window is the completion time of τ_i's packet instance. The maximum network latency R_i for τ_i is hence given by:

$$R_i = W(i) + J_i^R \quad (10)$$

We also find that for $\forall \tau_n \in S(i)$, the relation $\lceil \frac{W(i)+J_n^R}{T_n} \rceil = 1$ is always true. Eq.(5) in this situation is simplified as $\sum_{\forall \tau_n \in S(i)} C_n$ and the priority window analysis scheme is degraded to the composite analysis presented in [18]. Actually, the composite model analysis is only a specific case in the priority window analysis when $W(i) \leq min(T_n - J_n^R)$.

If $min(T_n - J_n^R) < W(i) \leq T_i - J_i^R$ is true, this implies that only the first packet instance of τ_i is served during this window and no self-blocking occurs. But multiple packet instances from any other flow in $S(i)$ may fall into the current window because of $\exists \tau_n \in S(i)$, $\lceil \frac{W(i)+J_n^R}{T_n} \rceil > 1$. These multiple blockings will delay the completion time of the current packet, but the worst case latency still can be found by checking the priority window. In this case, only one packet instance is released by τ_i, hence the existing relation showed by Eq.(10) is still valid and hence provides the maximum network latency.

From the above discussion, the maximum network latency is calculated by Eq.(10) when $W(i) \leq T_i - J_i^R$.

□

If $W(i) > T_i - J_i^R$, then more than one packet instance of τ_i is generated during a priority level-G_i window. Some successive generated packets from τ_i might be blocked by previous ones. In this situation, the delay from self-blocking also needs to be taken into account.

Theorem 3. *The maximum network latency R_i for τ_i is given by:*

$$R_i = \max_{q=1,\ldots,\lceil \frac{W(i)+J_i^R}{T_i} \rceil} (w_q(i) - (q-1)T_i + J_i^R) \quad (11)$$

where q is the index of packet instance, and $w_i(q)$ is given

by:

$$w_q(i) = qC_i + \sum_{\forall \tau_n \in S(i), \tau_n \neq \tau_i} \lceil \frac{w_q(i) + J_n^R}{T_n} \rceil C_n + \sum_{\forall \tau_j \in hp(i)} \lceil \frac{w_q(i) + R_j - C_j + J_j^R}{T_j} \rceil C_j \quad (12)$$

when $W(i) > T_i - J_i^R$.

Proof. The number of packets that could be released from τ_i before the end of the priority window is given by:

$$\lceil \frac{W(i) + J_i^R}{T_i} \rceil \quad (13)$$

To determine the worst case network latency, we must check all the packet instances during the priority window. The maximum of these values gives the worst case network latency.

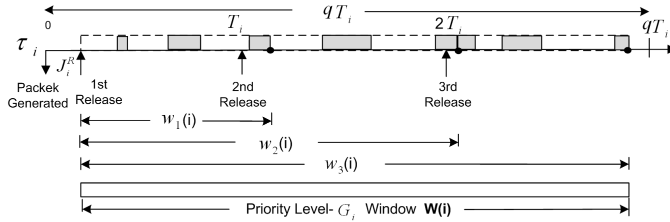


Figure 7. Priority Level-G_i window with Self-blocking

Figure 7 shows self-blocking during a priority level-G_i window. We use the index variable q to denote a packet instance of τ_i. The first packet in the window corresponds to $q = 1$ and the final one is $q = \lceil \frac{W(i)+J_i^R}{T_i} \rceil$. Therefore, the time from the first release of τ_i until achieving the qth transmission is given as a collection of service requirements from all the flows which compete the priority-G_i shared resource. We assume a new time phase $w_q(i)$ which denotes the time interval from the beginning of the priority window until the completion of the qth packet transmission. The time phase for the 1st, the 2nd and the 3rd packet in the window are shown in Figure 7 as $w_1(i)$, $w_2(i)$ and $w_3(i)$ respectively. The time phase $w_q(i)$ is given by:

$$w_q(i) = qC_i + \sum_{\forall \tau_n \in S(i), \tau_n \neq \tau_i} \lceil \frac{w_q(i) + J_n^R}{T_n} \rceil C_n + \sum_{\forall \tau_j \in hp(i)} \lceil \frac{w_q(i) + R_j - C_j + J_j^R}{T_j} \rceil C_j \quad (14)$$

The variable qC_i accounts for the transmission service time of the first q packet instances of τ_i during the priority window. The final part of the right hand side of this equation includes all the service requirements from priority level G_i or higher flows which fall in this time windows $w_q(i)$. The value of $w_q(i)$ can be found by the similar iteration policy while starting with $w_q(i)^0 = C_i + qT_i$ and ending when $w_q(i)^{n+1} = w_q(i)^n$. The generation time of the qth packet happens at instant $(q-1)T_i$ relative to the start of priority window so the network latency of the qth instance is given by:

$$R_i(q) = w_q(i) - (q-1)T_i + J_i^R \quad (15)$$

The maximum network latency can occur at any one of these packet releases during the priority window. We will consecutively analyze each release until τ_i stops blocking itself; which means the packet transmission service finishes within the same period as it is released, $W(i) \leq qT_i - J_i^R$. Thus maximum network latency is given by:

$$R_i = \max_{q=1,\dots,\lceil \frac{W(i)+J_i^R}{T_i} \rceil} (w_q(i) - (q-1)T_i + J_i^R) \quad (16)$$

□

Finally, flow τ_i is schedulable, if and only if $R_i \leq D_i$.

5 A case example

Revisiting the example given in Figure 2. The interrelations between these traffic-flows have been examined in section 3.2. The attributes of the traffic-flows are showed in Table 1. The time units are not necessary in this analysis as long as all the traffic-flows use the same base.

Real-Time Traffic-flow	C	G	T	D	J^R
τ_1	2	1	8	8	0
τ_2	2	1	11	11	0
τ_3	4	1	13	13	0
τ_4	3	2	8	12	0
τ_5	1	2	30	30	0

Table 1. Traffic-flows Description

Flows τ_1, τ_2 and τ_3 share the same priority G_1. Utilizing priority window analysis, first we calculate the $W(1)$ according to Eq.(2):

$W(1) = \lceil \frac{W(1)}{T_1} \rceil C_1 + \lceil \frac{W(1)}{T_2} \rceil C_2 + \lceil \frac{W(1)}{T_3} \rceil C_3$

Utilizing the iterative technique,

$W(1)^0 = 2 + 2 + 4 = 8$
$W(1)^1 = \lceil \frac{8}{8} \rceil 2 + \lceil \frac{8}{11} \rceil 2 + \lceil \frac{8}{13} \rceil 4 = 2 + 2 + 4 = 8$

The recurrence stops at $W(1)$ = 8 which is less than $min(T_n - J_n^R)$ for $\forall \tau_n \in S(1)$. So R_1=R_2=R_3 = $W(1)$ =8 less than D.

Flows τ_4 and τ_5 share the priority level G_2. The maximum window for G_2 not only considers the blocking but also the interference from higher priority flow. In this case, τ_2 and τ_3 contend for the priority level G_2 shared resources and hence contribute direct interference. Besides that, the activity of indirect higher priority flow τ_1 also can introduce some extra interference which is treated as interference jitter. So the window for G_2 is given by:

$$W(2) = \lceil \frac{W(2)}{T_2} \rceil C_2 + \lceil \frac{W(2)+R_3-C_3}{T_3} \rceil C_3 + \lceil \frac{W(2)}{T_4} \rceil C_4 + \lceil \frac{W(2)}{T_5} \rceil C_5$$

which iteratively results in $W(2) = 22$.

For τ_5, $W(2) < T_5$, so only one instance is released during this window. According to Theorem 2, $R_5 = W(2) = 22$.

For τ_4, since $\lceil \frac{W(2)}{T_4} \rceil = 3$, there are three packet instances released during the priority window. We need check all these instances to determine the worst case network latency. Utilizing Theorem 3, the window phases for the first packet, first two packets and the first three packets are $w_1(2)$, $w_2(2)$ and $w_3(2)$ respectively. Using Eq.(14) and Eq.(15), we get:

$w_1(2) = 10$ and $R_4(1) = 10 - 0 = 10$
$w_2(2) = 19$ and $R_4(2) = 19 - 8 = 11$
$w_3(2) = 22$ and $R_4(3) = 22 - 8 * 2 = 6$

The maximum network latency is thus $max(R_4(1), R_4(2), R_4(3)) = 11$. All the flows meet their deadlines and the set is schedulable.

6 Conclusion

The new on-chip communication architectures need to provide different levels of service for various components on the same network. Wormhole switching with fixed priority preemption has been proposed as a possible solution for real-time on-chip communication. Utilizing a priority share policy, the resource overhead can be reduced effectively. However, in order to simplify the analysis process, an existing technique imposes a deadline no more than period restriction which will bring inflexibility to network exploration and design. In this paper, we relax this constraint and present a novel analysis approach to cover all possible situations. Utilizing this new analysis scheme, we can flexibly evaluate at design time the schedulability of traffic-flow sets with different QoS requirements in a real-time communication platform.

References

[1] 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*, 8:284–292, 1993.

[2] S. Balakrishnan and F. Ozguner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(7):664–678, 1998.

[3] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[4] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computer Survey*, 38(1):1, 2006.

[5] T. Bjerregaard and J. Spars. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34–43, 2005.

[6] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(105-128), 2004.

[7] W. J. Dally. Virtual-channel flow control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, 1992.

[8] W. J. Dally. Route packets, not wires: On-chip interconnection networks. *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689, 2001.

[9] S. Furber and J. Bainbridge. Future trends in SoC interconnect. In *IEEE International Symposium on VLSI Design, Automation and Test*, pages 183–186, 2005.

[10] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[11] S. L. Hary and F. Ozguner. Feasibility test for real-time communication using wormhole routing. *IEE Proceedings - Computers and Digital Techniques*, 144(5):273–278, 1997.

[12] N. Kavaldjiev, Gerard J. M. Smith, P. G. Jansen, and P. T. Wolkotte. A virtual channel network-on-chip for GT and BE traffic. In *ISVLSI '06: Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, page 211, Washington, DC, USA, 2006. IEEE Computer Society.

[13] B. Kim, J. Kim, S. J. Hong, and S. Lee. A real-time communication method for wormhole switching networks. In *ICPP '98: Proceedings of the International Conference on Parallel Processing*, pages 527–534, 1998.

[14] Z. Lu, A. Jantsch, and I. Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 960–964, 2005.

[15] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Design Automation and Test Europe Conference (DATE)*, page 20890, February 2004.

[16] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[17] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceeding of the 2nd ACM/IEEE International Symposium on Networks-on-Chip(NoCS)*, pages 161–170, 2008.

[18] Z. Shi and A. Burns. Real-time communication analysis with a priority share policy in on-chip networks. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2009.

[19] D. Wiklund and D. Liu. Socbus: Switched network on chip for hard real time embedded systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 78.1, Washington, DC, USA, 2003. IEEE Computer Society.

[20] P. T. Wolkotte, G. J. M. Smit, G. K. Rauwerda, and L. T. Smit. An energy-efficient reconfigurable circuit-switched network-on-chip. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 155.1, Washington, DC, USA, 2005. IEEE Computer Society.

Node activity scheduling in wireless sensor networks*

Saoucene Mahfoudh and Pascale Minet
INRIA
Rocquencourt
78153 Le Chesnay cedex, France
saoucene.mahfoudh@inria.fr, pascale.minet@inria.fr

Abstract

Wireless sensor networks have resources of limited capacity (e.g. bandwidth, processing power, memory and energy). That is why these resources should be efficiently used. Node activity scheduling is a technique that allows nodes to alternate sleep and awake states. This technique spares energy insofar as the sleep state is the state using the smallest power. Moreover, by allowing several nodes to transmit simultaneously without interfering, spatial reuse of the bandwidth is obtained. Furthermore with a smart schedule, data gathering can be done in a single cycle. All these reasons render node activity scheduling very attractive in wireless sensor networks. We propose in this paper a three-hop coloring algorithm for data gathering applications. Simulation results allow us to evaluate the number of colors needed to color all network nodes and hence to determine the reduced size of the activity period in each polling cycle. The complexity of our algorithm is given in terms of number of messages sent per node. We can then determine the network configurations for which coloring brings interesting benefits, namely a more efficient use of the bandwidth and the node energy, as well as a shorter delay to collect data ensuring their time consistency.

1. Introduction

With the increasing number of applications in domains as various as environment protection (detection of forest fire or seismic event, wild life protection), civilian protection (building and bridge monitoring), emergency rescue, exploration mission in hostile environments and home monitoring, wireless ad hoc and sensor networks have a promising future. However, nodes in such networks can have a limited amount of energy. This energy can be difficult, expensive or even impossible to renew. The protocols operating in such networks should be energy efficient to maximize the network lifetime. Node activity scheduling is one class of energy efficient techniques [1]. Since the sleep state is the least power consuming state, compared with the receive, transmit and idle states, the idea of the node activity scheduling protocols is to schedule node state between sleeping and active to minimize energy consumption while ensuring network and application functionalities.

Node activity scheduling allows nodes not only to spare energy but also to use bandwidth more efficiently. Some solutions take advantage of spatial reuse to determine the time intervals dedicated to node activity. Indeed, during the same time interval two transmitters can transmit simultaneously and successfully if they do not interfere. Spatial reuse can be obtained by means of a coloring algorithm. The coloring algorithm must allow unicast as well as broadcast transmissions. For instance, nodes need to broadcast $Hello$ messages in their one-hop neighborhood to indicate that they are still alive and to declare on the one hand the nodes they hear only (i.e. unidirectional links), and on the other hand the nodes they hear and are heard from (i.e. bidirectional links). Moreover as the radio propagation is versatile, an immediate acknowledgement is often required in unicast transmissions. Thus the sender is ensured that the (one-hop) receiver has correctly received its message, otherwise it retransmits it.

In this paper, we focus on node activity scheduling obtained with a node coloring algorithm and study the benefits brought to a data gathering application. The collected data correspond to a physical phenomenon that evolves with time. It is then essential to minimize the delay needed to collect these data from sensors generating them. Furthermore, ensuring that all data are gathered in a single polling cycle guarantees their time consistency. To achieve these goals and then maximize the advantage brought by coloring, coloring must take into account the data gathering tree in color selection. Otherwise, it can take, in the worst case, a number of polling cycles equal to the number of hops to reach the data sink from the considered sensor. The data gathering tree can be computed by the Prim's algorithm where the sink broadcasts a message to all sensor nodes. This message contains a number representing the level of the sending node in the tree. Initially, this number is set to 0 by the sink and is incremented at each retransmission.

This paper deals with node activity scheduling in wireless sensor networks used by data gathering applications. It is

*This study has been partly funded by the ANR OCARI project.

organized as follows. In Section 2 we give a brief state of the art related to node activity scheduling distinguishing four classes of solutions. We then recall the complexity of node coloring and present different types of coloring algorithms. In section 3, we first justify the design choices of our algorithm. The main principles of the coloring algorithm are given as well as the messages exchanged and the information maintained by each node. The algorithm itself is described. The performances of this algorithm are evaluated in Section 4 by means of simulations with different network configurations. The number of colors allows us to determine when coloring provides real benefits. The average number of messages sent per node is an indication of the induced overhead. With this information, we can then evaluate the activity period and the data gathering delay. In Section 5, we show how the coloring algorithm can adapt to message losses, tree changes and late arrivals of nodes. Finally, we conclude in Section 6.

2. State of the art

We first present different techniques for scheduling node activity. We then move to graph coloring that can be used to schedule node activity.

2.1. Node activity scheduling

The best way to save energy nodes is to turn off the sensor radio when it does not receive or transmit data, so keeping sensor nodes in the sleep state. This must be accompanied by a node scheduling activity to prevent network partition and message loss when some nodes are in sleep state. We can distinguish four classes of node activity scheduling

- Computation of connected sets of active nodes. In [2], Simplot et al propose a solution building a connected dominating set (i.e. each node is either in this subset or is a neighbor of a node in this subset). Only the nodes of this set are active. All other nodes can change their state to sleep mode. It is a distributed and localized solution: only information about one hop and two hops neighbors is needed. Other solutions like [3] propose to extend network lifetime by dividing the network nodes in disjoint sets, such that each node set meets the network and application functions. These sets are activated successively, and at any time only the nodes of one set are active. All others nodes are in the sleep state. To improve network lifetime, the number of disjoint sets must be maximized which is NP-complete. The solution proposed is centralized. These authors have shown in [4] that network lifetime can be improved by allowing non-disjoint sets.
- CSMA/CA. In [5], S-MAC, an energy efficient MAC protocol for sensor networks is introduced. The main goal of S-MAC protocol is to reduce energy consumption by using a periodic sleep-wake up cycle, while supporting good scalability and collision avoidance. It consists of three major components: 1) periodic listen and sleep, 2) collision and overhearing avoidance, and 3) message passing. It is based on a CSMA/CA channel access and the RTS/CTS mechanism, whose overhead reduces the protocol efficiency. Many other variations of S-MAC are proposed like T-MAC [6] with an adaptive length of active state, D-MAC [7] to reduce the network latency, O-MAC [8] to improve the throughput...
- TDMA. In its basic version, TDMA provides one transmission slot per node in the network. It provides a guaranteed access per cycle for every node and avoids collisions. However, it does not adapt to traffic variations. Many improvements have been proposed in order to take advantage of spatial reuse in wireless networks. Among them, USAP (Unifying Slot Assignment Protocol) [9, 10] has drawn a lot of attention. USAP [9] is a distributed TDMA slot assignment protocol for mobile multihop packet radio networks. It allows any node N_i to select a slot for transmission that is unassigned in its neighborhood. A slot is unassigned from N_i point's of view if no one-hop neighbor of N_i transmits or receives during this slot.

  In [11], a deterministic solution based on slot assignment named TRAMA is proposed. It consists in three modules: 1) a neighborhood discovery protocol, 2) a schedule exchange protocol and 3) an adaptive election algorithm that selects the transmitter and receiver(s) for each time slot. Only nodes having data to send contend for a slot; notice however, that a node does not know which of its 1-hop and 2-hop neighbors have data to send. The node with the highest priority in its two-hop neighborhood wins the right to transmit in the slot considered. Each node declares in advance its next schedule containing the list of its slots and for each slot its receiver(s). The adaptivity of TRAMA to the traffic rate comes at a price: its complexity. To reduce the complexity of TRAMA, FLAMA [12] is introduced. However, this protocol is designed only for data gathering applications in sensor networks based on tree structure. The protocol is simplified both in terms of message exchange and processing complexity. The number of slots allocated by FLAMA to a node with a given traffic rate highly depends on node priority computation.
- Hybrid. Z-MAC [13] operates like CSMA under low contention and like TDMA under heavy contention, reducing collisions among two-hop neighbors by means of an initial slot assignment made by DRAND [14]. The goal of Z-MAC is to optimize the bandwidth efficiency of the MAC protocol, selecting CSMA/CA and TDMA when they exhibit the best performance. We can notice that Z-MAC does not allow an immediate acknowledgement of unicast messages. Indeed, this acknowledgement could cause a

conflict, as illustrated in Figure 1. Moreover, since a slot that is unused by its owner can be used by one of its neighbors, the nodes must stay awake in order to be able to receive this message if they are the destination. From the energy point of view, Z-MAC reduces the activity period in the polling cycle enforced by the application but does not allow nodes to sleep during the activity period, what does our coloring algorithm presented in Section 3, whose goal is to maximize network lifetime by scheduling node activity. DRAND [14] assigns slots to nodes in such a way that one-hop and two-hop neighbors have different slots. This randomized algorithm has the advantage of not depending on the number of network nodes but at the cost of an asymptotic convergence.

It appears that different techniques have been used to schedule node activity. The techniques based on CSMA variants behave well in case of light loads and easily adapt to topology changes, whereas techniques based on TDMA variants outperform them in case of high loads. In this study, we focus on wireless sensor nodes, where only a few nodes are mobile. Graph coloring has been introduced to increase the efficiency of TDMA with spatial reuse: two nodes with the same color transmit simultaneously without interfering [15, 23].

2.2. Graph coloring

One-hop graph coloring consists in coloring nodes with the minimum number of colors such that two adjacent nodes have not the same color. The problem of one-hop coloring has been shown NP-complete in [18] for the general case. The first one-hop graph coloring algorithms proposed were centralized. Among the greedy algorithms (i.e. no color backtracking), Dsatur, [16, 17], where the vertex with the highest number of already colored neighbor vertices is colored first, exhibits very good performances, even if it is not optimal. It is then followed by Largest First, where the node with the highest degree is colored first.

Distributed one-hop graph coloring algorithms also exist. Some are probabilistic such as [20, 19]. Other algorithms are deterministic such as [22] where Distributed Largest First (DLF) is proposed. Another approach consists in finding maximum independent sets and then coloring these sets independently, as in [24], because both problems are related [21].

The efficiency of a distributed coloring algorithm, [22, 21], can be evaluated by:

- the number of colors needed to color a graph G.
- the average number of messages sent per node. This shows the overhead induced by the coloring algorithm.
- its time complexity, expressed in the case of a distributed algorithm, by the maximum number of rounds needed to color each node. A round is such that every node can:
  - send a message to all its one-hop neighbors,
  - receive the messages sent by them,
  - perform some local computation based on the information contained in the received messages.

In wireless ad hoc and sensor networks, distributed coloring algorithms based on decision rounds require less messages than those based on the alternation of proposal/decision rounds, such as Distributed Largest First [22]. A comparative performance evaluation can be found in [15] for two-hop graph coloring. In fact, in a round, a node sends a message to its one-hop neighbors, this message contains its color and the colors of its one-hop neighbors. It receives the messages from its one-hop neighbors and takes a decision if it has the highest priority. This is the principle of our coloring algorithm presented in Section 3.
Two-hop graph coloring has also been investigated in the case of data gathering applications [23], where the goal is to reduce the duration needed to gather data from all sensors. Starting to color the highest level in the tree and ending by the sink as in [23] is easy, but the insertion of a new leaf can lead to recolor the tree.

3. Algorithm for three-hop coloring of a tree and properties

3.1. Design choices

From the state of the art, it turns out that:

- the coloring problem being NP-complete, we seek for an heuristic. The state of the art points out that the algorithms achieving the best performances are those taking into account the node degree (i.e. number of one-hop neighbors if one-hop coloring) and having identical rounds (no alternate of proposal/decision rounds).
- Furthermore, link coloring specifies the sender and the receiver whereas node coloring specifies only the sender. Hence, only node coloring enables broadcast transmissions.
- Node coloring must be such that two nodes having the same color can transmit simultaneously without interfering. Interferences being assumed to be limited to two-hop, two-hop coloring is needed. However, as Figure 1 shows, it is not sufficient if a node is allowed to transmit an immediate acknowledgement of the received data: nodes A and D, 3-hop away, use the same color and the $data$ message sent by A collides with the ack message sent by C to node D. Hence, three-hop coloring is needed to accommodate immediate acknowledgement of unicast

transmissions. In other words, with three-hop coloring, the same color can be reused four-hop away. For instance, nodes A and E can have the same color. In such a case, node A can transmit data to B while D is acknowledging the data received from E without interfering.

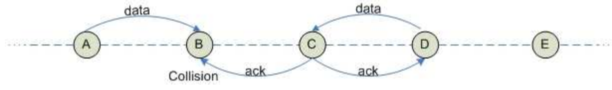


Figure 1. Collision with two-hop coloring and immediate acknowledgement.

- Without care, node activity scheduling increases the delay needed to collect information from a sensor. In the worst case, a number of cycles equal to the distance, in hop number, to the sink is needed to reach the sink. To avoid this phenomenon, a node must transmit its data before its parent in the data gathering tree. To achieve that, the color of a node must be higher than the color of its parent in the tree and slots are assigned in the decreasing order of colors.

3.2. Notations

We consider any data gathering tree $\mathcal{T}$ and any node N in this tree.
Let $\mathcal{N}^3(N)$ denote the neighborhood up to 3 hops from N. This set contains the 1-hop, 2-hop and 3-hop neighbors of N.
Let $parent(N)$ denote the parent of N in $\mathcal{T}$. By descendant, we mean the children, children of the children and so on...
Let $Desc(N)$ denote the set of descendants of N in $\mathcal{T}$.
Let $|Desc(N)|$ denote the cardinal of this set.

For any node N, we say that N has a higher priority than node $N' \in \mathcal{N}^3(N)$ if and only if:

- either N' meets $|Desc(N')| > |Desc(N)|$
- or $(|Desc(N')| = |Desc(N)|$ and $identifier(N') < identifier(N))$.

Colors are assumed to be represented by positive integers starting with 0, the color of the root of the data gathering tree.

3.3. Principles

To achieve the goals given in section 1, our coloring algorithm is based on the two following rules:

- **Rule R1:** any node N colors itself if and only if:
  - all nodes in $\mathcal{N}^3(N)$ having a strictly higher number of descendants are already colored,
  - and all nodes in $\mathcal{N}^3(N)$ having the same number of descendants and a smaller identifier are already colored.
- **Rule R2:** node N takes the smallest color available in $\mathcal{N}^3(N)$ strictly higher than the color used by its parent.

It follows that:

- the first node to color is the root of the tree.
- each node has a color strictly higher than the color of its parent and of all its ascendants.
- each node has a color higher than or equal to its level in the tree (i.e. distance to the root in the tree expressed in hop number).

Figure 2 depicts the colors assigned to 15 nodes called A,B...O building a tree rooted at node A. Eight colors are needed. Hence, eight slots are sufficient instead of 15 for classical TDMA. The same color can be used by several nodes (e.g; color 6 is used by nodes K, L, M and O).

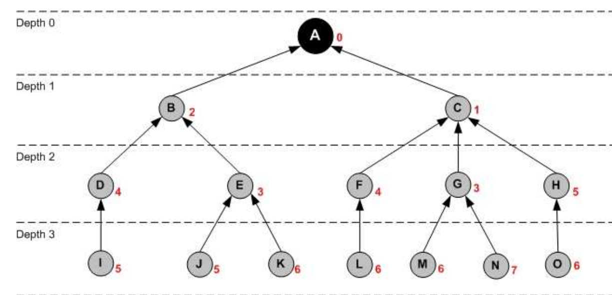


Figure 2. Colors assigned to a tree of 15 nodes.

Figure 3 depicts the slot assignment. For instance, nodes K , L, M and O that have the color 6 transmit in the same slot, numbered 6. Notice that each child transmits before its parent. Hence, in the cycle of eight slots illustrated by this figure, any sensor can send its value to the sink, assuming data aggregation.

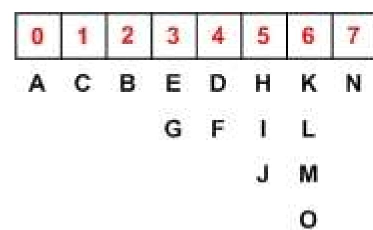


Figure 3. Slots assignment.

3.4. Messages exchanged

In this algorithm, two messages are exchanged:

- the $Color$ message: it is the only message needed to color all network nodes.
- the $MaxColor$ message: this message is needed at the end of the coloring algorithm to inform the root of the tree of the number of colors used in the tree.

The $Color$ message contains:

- the node identifier,

- its sequence number,
- its number of descendants,
- its color, if already assigned,
- for any one-hop neighbor node:
  - the node identifier,
  - its sequence number,
  - its number of descendants,
  - its color, if already assigned,
- for any two-hop neighbor node:
  - the node identifier,
  - its number of descendants,
  - its color, if already assigned,

The $Color$ message is broadcast one-hop. The sequence number of the sender is incremented at each change in the fields of the $Color$ message, except the sequence number fields. In order to tolerate message losses, a node N retransmits its $Color$ messages until all its one-hop neighbors have sent a $Color$ message containing a sequence number for N equal to the current one. Notice that in case of topology changes, a link can be broken causing an update in the set $\mathcal{N}^3(N)$.

The $Maxcolor$ message contains:

- the node identifier of the sender,
- its sequence number,
- the maximum color used by all the descendants of the sender node.

The $MaxColor$ message is sent from any node N to its parent until reaching the root. This unicast message is acknowledged. When it reaches the root, it contains the maximum number of colors used.

3.5. Information maintained by each node

Each node maintains the following information:

- its node identifier,
- its sequence number,
- its number of descendants,
- its color, if already assigned,
- its parent in $\mathcal{T}$,
- for each child in $\mathcal{T}$,
  - the node identifier,
  - the maximum color used by this node and its descendants,
  - a sequence number,
- for any one-hop neighbor node:
  - the node identifier,
  - its sequence number,
  - its number of descendants,
  - its color, if already assigned,
- for any two-hop neighbor node:
  - the node identifier,
  - its number of descendants,
  - its color, if already assigned,
- for any three-hop neighbor node:
  - the node identifier,
  - its number of descendants,
  - its color, if already assigned,

3.6. Algorithm

We now give the three-hop coloring algorithm for a data gathering application:

Algorithm 1 Three-hop coloring algorithm of a data gathering tree

1: **Procedure Process(**$Color$**message)**
2: **begin procedure**
3: **if** there is a change in the 1, 2 or 3-hop neighborhood or in the tree **then**
4: update $\mathcal{N}^3(N)$
5: update $\mathcal{T}$, $parent(N)$ and $\mathcal{D}esc(N)$
6: **end if**
7: **if** $|Desc(N)|$ has not yet been computed and the 1,2 and 3-hop neighborhoods as well as $|Desc(M)|$ for each child M are known **then**
8: compute $|Desc(N)|$ by adding the values of $1 + |Desc(M)|$ for each M, child of N.
9: **end if**
10: maintain the priority and color of any node in $\mathcal{N}^3(N)$
11: **if** N is the node with the highest priority among the uncolored nodes in $\mathcal{N}^3(N)$ **then**
12: N selects the smallest color unused in $\mathcal{N}^3(N)$ strictly higher than $color(parent(N))$
13: **end if**
14: **end procedure**
15: **Main**
16: **repeat**
17: **repeat**
18: N broadcasts (1-hop) its $Color$ message containing:
- a sequence number seq incremented at each change in the $Color$ message fields, except the sequence number fields
- its identifier
- its priority and color if already assigned
- its list of one-hop neighbors with their identifier, their sequence number, their priority and color if already assigned
- its list of two-hop neighbors with their identifier, their priority and color if already assigned.

19: N waits for the $Color$ message of its 1-hop neighbors
20: **upon receipt of a** $Color$ **message,**
21: N Process($Color$message)
22: **until** all one-hop neighbors have received the $Color$ message of N with number seq
23: **until** all nodes in $\mathcal{N}^3(N)$ are colored

3.7. Computation of the number of colors

We now evaluate the number of colors needed to color all network nodes. We first assume that the neighborhood up to 3-hop does not bring additional constraints to the one represented in the tree.

Property 1: If at each level p in the tree, each node has the same number of children $nbchild_p$ and there is no additional constraints to the one depicted in the tree, then the number of colors used is equal to:

$nbcolor = 1 + \sum_{p=0}^{depth-1} nbchild_p$.

If $\mathcal{N}^3(N)$ introduces additional constraints to those given in the tree, there exists a node N that has for 1-hop, 2-hop or 3-hop a node N' that is neither its parent, grandparent, brother, uncle, child, nephew, grandchild. in such a case, colors cannot be reused so easily, we then get:

Property 2: If at each level p in the tree, each node has the same number of children $nbchild_p$ and there are additional constraints to the one depicted in the tree, then the number of colors used is equal to:

$nbcolor = 1 + \sum_{p=0}^{depth-1} nbchild_p + \sum_N \delta_{N,N'}$, with $\delta_{N,N'} = 1$ if and only if N cannot take a color in $[color(parent(N)) + 1, cmax]$ with $cmax$ the highest color used by the nodes N' colored before N. Initially $cmax = color(parent(N)) + 1$ and $cmax = cmax + 1$ each time a new color is used to color N.

We can get the formula giving the color of a node, depending on the color of its parent and the colors already used in its neighborhood up to three hops.

Property 3: The color of any node N is given by:

$color(N) = 1 + color(parent(N)) + \sum_{N' \ already \ colored \ \in \mathcal{N}^3(N)} \delta_{N,N'}$

with $\delta_{N,N'} = 1$ if and only if N cannot take a color in $[color(parent(N)) + 1, cmax]$ with $cmax$ the highest color used by the nodes $N' \in \mathcal{N}^3(N)$ colored before N. Initially $cmax = color(parent(N)) + 1$ and $cmax = cmax + 1$ each time a color in $[color(parent(N))+1, cmax]$ cannot be used to color N.

3.8. Comparison with another tree coloring algorithm

In this section, we compare our coloring algorithm with another one proceeding level by level (from the lowest level to the highest one) and applying the rules R'1 and R2 instead of rules R1 and R2, where the rule R'1 is:

- **Rule R'1:** any node N colors itself if and only if:
  - any node N' in $\mathcal{N}^3(N)$ having a level strictly lower than the level of N is already colored,
  - and any node N' in $\mathcal{N}^3(N)$ of the same level as N but $|\mathcal{N}^3(N')| > |\mathcal{N}^3(N)|$ is already colored,
  - and any node N' in $\mathcal{N}^3(N)$ of the same level as N but $|\mathcal{N}^3(N')| = |\mathcal{N}^3(N)|$ and a smaller identifier than N is already colored.
- **Rule R2:** node N takes the smallest color available in $\mathcal{N}^3(N)$ strictly higher than the color used by its parent.

The simple example depicted on Figures 4 and 5 shows that the number of colors used by our algorithm (here 4 colors) is smaller than this used by this other algorithm (here 5 colors). The reason comes from the fact that our algorithm favors the nodes with a high number of descendants. As they are colored first, they can get smaller colors unlike in the other algorithm. The nodes with a small number of descendants are colored at the end, but they can reuse colors used by nodes that are not in their up to 3-hop neighborhood. We will see in subsection 4.4 that the number of colors used by our algorithm is considerably smaller than the number of colors used by the other algorithm using rules R'1 and R2.

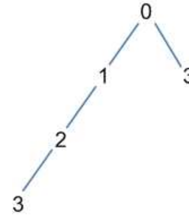


Figure 4. Tree colored with our algorithm.

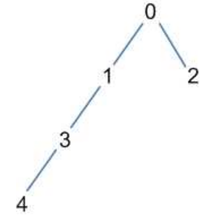


Figure 5. Tree colored with the other algorithm.

4. Performance evaluation

4.1. Simulation parameters

For performance evaluation purpose, we have considered different network configurations. Each configuration is characterized by a node number and a node density. The nodes are randomly deployed over the network area. The number of nodes ranges from 50 to 200, whereas the node density (i.e. the average number of one-hop neighbors per node) ranges from 10 to 20. For each configuration, we have run 10 simulations and the result depicted in the curves is the average of these 10 simulations. The data gathering tree is the tree of the shortest paths. This study can also be extended to the case where the data gathering tree is built taking into account residual energy of nodes and their capacity to forward messages.

For all the network configurations generated, we have computed different performance criteria such as the number of colors needed to color all network nodes, the average number of messages sent per node. We then show how this small number of colors benefits to the active period duration and the data gathering delay.

4.2. Number of colors

We now evaluate the number of colors needed by our algorithm to color all network nodes of the generated configurations. Simulation results are illustrated in Figures 6 and 7.

We first depict in Figure 6 the number of colors as a function of the number of nodes for different densities. We observe that this number strongly depends on node density and weakly on node number. As the color a node N cannot be reused in its neighborhood up to 3-hop, whose size is proportional to the density, this explains the strong dependency between the number of colors and the density. It is very interesting to notice that all curves depicting the number of colors are below the first bisectrix. This means that the number of colors is much smaller than the number of nodes, except for the network configuration of 50 nodes and a density of 20, where almost all nodes are 1, 2 or 3-hop neighbors. In other words, spatial reuse is always obtained for all the network configurations considered in the simulations. The higher the distance to the first bisectrix, the higher the spatial reuse. The average number of nodes using the same color is given by the number of nodes divided by the number of colors.

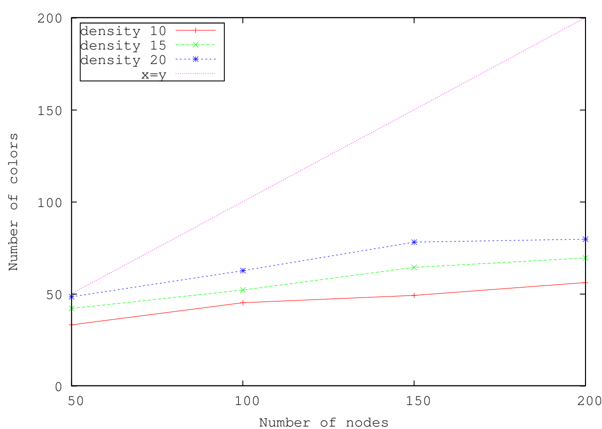


Figure 6. Number of colors as a function of the number of nodes and the density.

We now depict in Figure 7 the number of colors as a function of the tree depth for different densities. Notice that for a given density, the number of nodes increases with the tree depth. The number of colors increases with the tree depth, this is due to the fact that the color of a node is higher than the color of its parent. Moreover, the impact of tree depth seems to stabilize in the simulated configurations. The stabilization point is reached sooner for high densities.

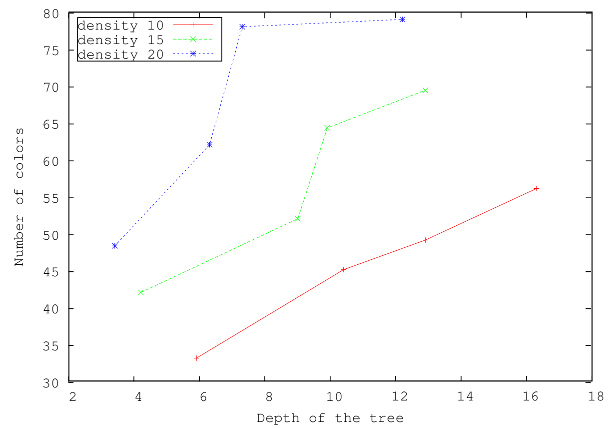


Figure 7. Number of colors as a function of the tree depth and the density.

4.3. Average number of messages sent by a node

In order to evaluate the overhead induced by the coloring algorithm, we compute the average number of messages sent per node in all the generated network configurations. Simulation results are illustrated in Figure 8. The number of messages exchanged strongly increases with density and moderately with the number of nodes. It remains reasonable in all configurations.

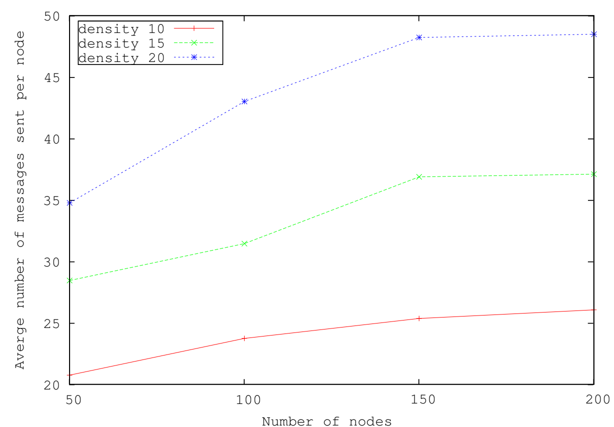


Figure 8. Average number of messages sent per node.

4.4. Comparative evaluation

We now compare the number of colors needed in our coloring algorithm with this needed by the other algorithm presented in subsection 3.8. We consider a network density of 10. Our algorithm exhibits very good performances as illustrated in Figure 9. This is due to the fact that it assigns a smaller color to nodes having a high number of descendants. Other nodes (with a small number of descendants) can reuse already used colors. Hence, the smallest number of colors. We can also observe that this trend increases with the number of nodes.

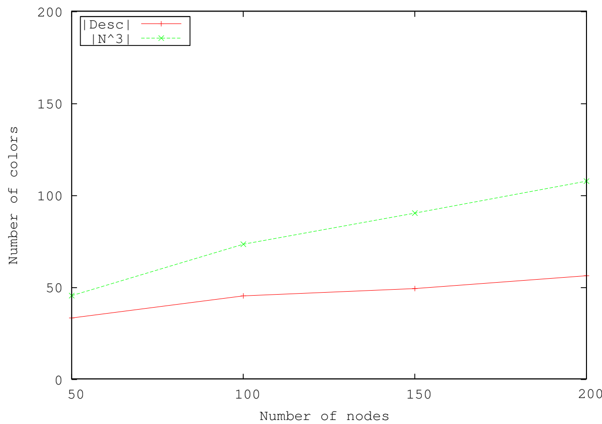


Figure 9. Comparative evaluation of the number of colors used by the $|Desc(N)|$ and $|\mathcal{N}^3(N)|$ algorithms.

4.5. Activity period duration

The physical phenomenon that is monitored by the wireless sensor network determines the polling cycle of the sensors. This polling cycle comprises two periods, as represented in Figure 10: an activity period during which sensors are active and an inactive period where all sensors are sleeping. Only the activity period is mandatory.

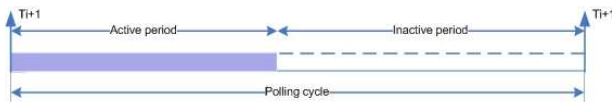


Figure 10. The active period in the polling cycle.

The coloring algorithm is used for slot assignment in the activity period. Slots are assigned to nodes per color instead per node like in classical TDMA: all nodes of the same color can transmit simultaneously without interfering. Hence, this space reuse leads to a considerable reduction of the activity period. The benefit is equal to $(numberofnodes - numberofcolors) \cdot slotsize$.

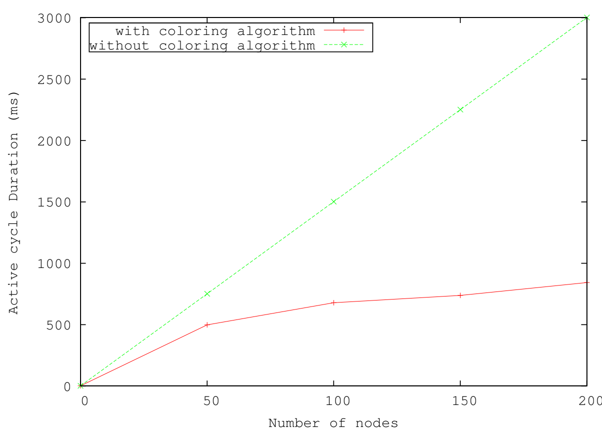


Figure 11. Duration of the active period as a function of the number of nodes.

Figure 11 shows the quantitative improvement brought by the coloring algorithm with regard to a classical TDMA. A slot size of 15 ms has been taken.

4.6. Data gathering delay

The data gathering delay is the time needed to collect at the sink all data transmitted by the sensors. If we assume that the slot has the capacity to contain the aggregated information transmitted by any child of the root (i.e.: the nodes having the highest quantity of information to transmit), then the time needed to collect the data from all sensors is equal to the activity period, whereas the latency is equal to the polling cycle duration.

Ensuring that each node has a color higher than its parent in the tree allows our colring algorithm to reduce the data gathering delay and to obtain time-consistent data. In fact, in only one cycle, all data collected can reach the sink by assigning slots to colors in the decreasing order: this allows a child to transmit before its parent that can then aggregate the data received from its children. The increasing order is chosen in reverse case of data dissemination. Moreover, collecting data in a single cycle guarantees time consistency of the data collected from the different sensors: for example temperature, humidity degree and pression are measured in the same cycle.

4.7. Benefit brought by coloring

A coloring algorithm brings several advantages:

- an efficient use of the bandwidth by enabling spatial reuse and avoiding medium access contention.
- an increased network lifetime by enabling nodes to sleep during the activity period and the increased inactivity period. During the activity period, a node sleeps in all the slots except:
  - the slot assigned to its color to transmit its messages,
  - the slots assigned to its one-hop neighbors to receive their messages in case it is the destination.

  Optimizations can be brought in order to allow a node to sleep the soonest possible. For example, if a node has not detected any signal in the slot of its one-hop neighbor during a predetermined duration, it can deduce that this neighbor has nothing to transmit and goes back to the sleep state. Similarly, a node can sort the messages it has to transmit: firts the broadcast transmissions and second the unicast transmissions ordered by the destination identifier. It follows that a one-hop neighbor can easily detect that there is no message for it and then sleeps again.
- a shorter delay to collect data from sensors. In a single cycle, all data can be collected, ensuring their time consistency.

- slots assigned according to the increasing order of colors reduces the time needed to disseminate data in the network: for example, information sent by the sink to all nodes.

5. Adaptivity of the coloring algorithm

5.1. Message loss

Our coloring algorithm tolerates message losses by means of the sequence number used to detect losses. A node N retransmits its $Color$ message with sequence number seq as long as it has not received a $Color$ message from all its one-hop neighbors containing the sequence number seq for N.

When a one-hop neighbor N' receives the $Color$ message of N, it updates the neighborhood information locally stored. Two cases are possible:

- There is a change concerning:
  - either the color or the number of descendants of the node itself, or of a one-hop or two-hop neighbor,
  - or the appearance or disappearance of a one-hop or two-hop neighbor.

  N' increments its own sequence number seq' and sends a $Color$ message to its one-hop neighbors.
- Otherwise N' sends a $Color$ message without incrementing its sequence number.

It follows that any node will know a change in its neighorhood up to 3-hop, whatever the change is: node, color or number of descendants.

5.2. Tree change

In this section, we consider the case where the link between a node N and its parent in the tree is broken. This node N will first try to attach itself to another node N' in the tree. We assume that this link is an existing link: the parent of N is chosen among its existing one-hop neighbors. Hence this parent is already taken into account in $\mathcal{N}^3(N)$. We will see in the next section what happens when this assumption is not true.

Two cases are possible:

- the tree change does not impact the colors already assigned: no node in $\mathcal{N}^3(N)$ has the same color as N and $color(N) > color(N')$.
- the tree change creates a violation of the rule 2: the color of N is not higher than the color of its new parent N'. In this case, node N selects the smallest color available in $\mathcal{N}^3(N)$ belonging to $[color(N')+1, min_{child}color(child) - 1]$. If there is no color available, N takes the smallest color of its children, and so on.

5.3. Topology changes

By topology change, we mean that a change in $\mathcal{N}^3(N)$ occurs: a new link is created or an existing link is broken. It can be caused by:

- node mobility: a node moves in the network area causing the breakage of its existing links and the creation of new ones.
- late node arrival: when a new node joins the already colored network, new links will appear.

In both cases, when a new link is created, it may have as consequence that two nodes that were not 1-hop, 2-hop or 3-hop neighbors become 1-hop, 2-hop or 3-hop neighbors. Hence, a color conflict between two nodes can occur. A color conflict is defined as follows:

A color conflict occurs between two nodes having the same color when these nodes prevent each other or some neighbor destination to receive correctly the intended message because of a collision.

Notice that in the absence of mobility and late node arrival, nodes that are 1, 2 or 3-hop neighbors never conflict by definition of the algorithm. Furthermore, this definition takes advantage of the capture effect and considers the only color conflicts where the intended destination is prevented to receive its message.

When a color conflict occurs between two nodes N and N', the node with the highest priority keeps its color whereas the node with the smallest priority takes another color according to rule R2.

6. Conclusion

In this paper, we proposed a three-hop coloring algorithm for wireless sensor networks supporting data gathering applications. We shown that this algorithm uses a small number of colors despite the rule that the color of a node is higher than the color of its parent in the data gathering tree. This algorithm has been designed for the ANR OCARI project [25] for a medium whose physical layer is the IEEE 802.15.4. The benefits brought by this coloring algorithm are quadruple:

- the bandwidth is used more efficiently, taking advantage of spatial reuse and avoiding medium access contention by means of colors;
- the nodes spare their residual energy in sleeping while they have no message to send or to receive;
- the data transmitted by the sensors can be collected in a single activity period, ensuring their time consistency.
- colors can also be used to disseminate information from the sink to all sensor nodes in a single cycle.

Our coloring algorithm contributes to maximize network lifetime by reducing the activity period as well as allowing any node to sleep in this period (in the slots that are not assigned to its color and the colors of its 1-hop neighbors).

References

[1] S. Mahfoudh, P. Minet, *Survey of energy efficient strategies in wireless ad hoc and sensor networks*, ICN 2008, IEEE International Conference on Networking, Cancun, Mexico, April 2008.

[2] J. Carle, D. Simplot-Ryl, *Energy-Efficient Area Monitoring for Sensor Networks*, Computer, vol. 37, no. 2, pp. 40-46, Febuary, 2004.

[3] M. Cardei, M. Thai, Y. Li, W. Wu, *Energy-efficient target coverage in wireless sensor networks*, IEEE INFOCOM'05, Miami, Florida, March 2005.

[4] M. Cardei, D. Du, *Improving wireless sensor network lifetime through power aware organization*, ACM Jal of Wireless Networks, May 2005.

[5] W. Ye, J. Heidmann, D. Estrin, *An Energy-Efficient MAC Protocol for Wireless Sensor Networks*, IEEE INFOCOM, New York, USA, June 2002.

[6] T. V. Dam, K. Langendoen, *An adaptive energy-efficient MAC protocol for wireless sensor networks*, ACM SenSys'03, November 2003.

[7] G. Lu, B. Krishnamachari, C. Raghavendra, *An Adaptive Energy-Efficient and Low-Latency MAC for Data Gathering in Wireless Sensor Networks*, Parallel and Distributed Processing Symposium, April 2004.

[8] G. Lu, B. Krishnamachari, C. Raghavendra *O-MAC: An Organized Energy-Aware MAC Protocol for Wireless Sensor Networks*, IEEE ICC, Glasgow, UK, June 2007.

[9] C.D. Young, *USAP: a unifying dynamic distributed multichannel TDMA slot assignment protocol* IEEE MILCOM 1996, Vol. 1, October 1996.

[10] C.D. Young, *USAP multiple access: dynamic resource allocationfor mobile multihop multichannel wireless networking* IEEE MILCOM 1999, Vol. 1, November 1999.

[11] V. Rajendran, K. Obraczka, J.J. Garcia-Luna-Aceves, *Energy-efficient, collision-free medium access control for wireless sensor networks*, Sensys'03, Los Angeles, California November 2003.

[12] V. Rajendran, J.J. Garcia-Luna-Aceves, K. Obraczka, *Energy-efficient, application-aware medium access for sensor networks*, IEEE MASS 2005, Washington, November 2005.

[13] I. Rhee, A. Warrier, M. Aia, J. Min, *Z-MAC: a hybrid MAC for wireless sensor networks*, SenSys'05, San Diego, California, November 2005.

[14] I. Rhee, A. Warrier, L. Xu, *Randomized dining philosophers to TDMA scheduling in wireless sensor networks*, Technical Report TR-2005-21, Dept of Computer Science, North Carolina State University, April 2005.

[15] P. Minet, S. Mahfoudh, *SERENA: SchEduling RoutEr Nodes Activity in wireless ad hoc and sensor networks*, IWCMC 2008, IEEE International Wireless Communications and Mobile Computing Conference, Crete Island, Greece, August 2008.

[16] D. Brelaz, *New methods to color the vertices of a graph*, Communications of the ACM, 22(4), 1979.

[17] J. Peemoller, *A correction to Brelaz's modification of Brown's coloring algorithm*, Communications of the ACM, 26(8), 1983.

[18] M. Garey, D. Johnson, *Computers and intractability: a guide to theory of NP-completeness*, W.H. Freeman, San Francisco, California, 1979.

[19] I. Finoccho, A. Panconesi, R. Silvestri, *Experimental analysis of simple distributed vertex coloring algorithms*, SODA 2002, San francisco, California, January 2002.

[20] C. Busch, M. Magdon-Ismail, F. Sivrikaya, B. Yener, *Contention-free MAC protocols for wireless sensor networks*, DISC 2004, Amsterdam, Netherlands, October 2004.

[21] F. Kuhn, R. Watttenhofer, *On the complexity of distributed graph coloring*, PODC 2006, Denver, Colorado, USA, July 2006.

[22] J. Hansen, M. Kubale, L. Kuszner, A. Nadolski, *Distributed largest-first algorithm for graph coloring*, EURO-PAR 2004, Pisa, Italy, August 2004.

[23] S. Gobriel, R. Cleric, D. Mosse, *Adaptations of TDMA scheduling for wireless sensor networks*, RTN 2008, International Workshop on Real-Time Networks, Prague, Czech Republic, July 2008.

[24] T. Herman, S. Tixeuil, *A distributed TDMA slot assignment algorithm for wireless sensor networks*, ALGOSENSORS 2004, Turku, Finland, July 2004.

[25] OCARI project, http://ocari.lri.fr/.

Resource Management

Spare Capacity Distribution Using Exact Response-Time Analysis

Attila Zabos, Robert I. Davis, Alan Burns
Department of Computer Science
University of York, UK

Michael González Harbour
Computers and Real-Time Group
Universidad de Cantabria, Spain

Abstract

Real-time systems designed for use in dynamic open environments allow applications to enter and leave the system during runtime. This leads to changing runtime scenarios where the load and the spare capacity of hardware resources is influenced by the resource demand of running applications. Flexible real-time application components, with variable temporal parameters, can adapt their timing behaviour to these changing runtime scenarios and improve both, their Quality of Service (QoS) and the utilisation of system resources. In these open systems application components are often designed independently of each other, introducing the need for system management of resources at runtime. This management requires a mechanism to distribute the available system resources among the running application components, in a way that maximises resource usage and increases QoS with respect to their importance and temporal limits. This paper introduces a runtime algorithm for the distribution of spare capacity in flexible real-time systems. The efficiency of the presented algorithm is evaluated by empirical tests and performance measurements on embedded hardware.

1. Introduction

In embedded real-time systems being developed today it is common to find requirements for flexibility and support for dynamic behaviour that are key driving factors in the design of their architectures and scheduling methods. Manufacturers of these embedded and resource constrained systems are faced with the difficulty of providing guarantees on the real-time behaviour of their applications while at the same time handling flexibility and dynamic changes to the applications being executed. Traditional real-time scheduling focuses on worst-case behaviour and using it for static configurations implies that a large amount of capacity, that is only used occasionally, is statically allocated in order to manage dynamic changes in application demand. This conflicts with the common requirement to be able to use the maximum possible amount of the available resources.

In this context of requirements for flexibility and support for dynamic behaviour it is possible to design applications that adapt themselves to the available resources, trading the quality of the response with the usage of resources. In a system developed with this adaptivity in mind it is possible to maximise resource usage while trying to provide the best possible QoS.

The complexity of modern embedded systems is also driving the need to independently develop applications or application components that may join and leave the system during runtime. The available resources should be dynamically adapted to these changing situations. In most platforms these dynamic changes may be frequent, but not as fast as the regular application periods. We may have new applications that stay in the system for seconds or minutes, while their own internal periods may be in the milliseconds range.

2. Motivation

Flexible applications come in many different forms. For instance, multimedia systems need to process different kinds of video and audio streams that have highly variable computation times but require real-time processing and rendering. It is common that classic industrial control applications, such as a robot control, get mixed together with multimedia activities when the process in which the robot is working requires image capture and analysis, or remote video monitoring. Mobile phones and other embedded devices are turning themselves into devices with multiple capabilities, including audio and video processing, GPS positioning, Internet navigation, and more, in addition to their original radio link capabilities. In these systems it is common that the user starts or downloads new applications that must run together with the previous ones in an environment with limited resources; and many of these applications have real-time constraints.

Not all the applications running in the system are capable of adapting or adapting equally to the available resources. Most applications will require a minimum amount of resources to produce results of the minimum acceptable quality. Then, some applications may be able to use additional resources, while others won't. Of those that may adapt to the available resources the level of adaptation may be different. For instance, a video player may upgrade itself to a higher frame rate if more resources are available, changing the rate in a continuous way, up to a maximum level after which there is no perceived increase in the quality of the output. These type of applications are referred to as *continuous*. A con-

trol algorithm on the other hand may have two versions: a fast one with a low quality output and one requiring more computation time and providing a high quality output. In this case the system should allocate resources to run either one version or the other. Applications with this type of behaviour are referred to as *discrete*. In general we find applications that can operate and generate valid results at different frequencies (i.e. having a variable period), and/or handle different processing time assignments (i.e. having a variable execution time).

The FRESCOR[1] EU project is developing a middleware layer, that is placed on top of a real-time operating system, with the capability of running multiple application components and scheduling the resources that they need to run. Each application component describes the resource requirements through one or more *contracts*. These contracts specify the minimum resource requirements and the way in which the component is able to use any additional available resources. The contract is negotiated with the system in order to verify that the minimum resources required can be granted to the application component. Once a contract has been accepted by the system a *virtual resource (VR)* (which has similar properties to servers [10]) is allocated to the application component. This VR represents a resource reservation, and manages the consumption of the resource by the application component to ensure that it can always get its allocated budget, and that it never exceeds it, so that other application components can also run in the system with full temporal protection. On the other hand, when application components complete processing and terminate, their VRs are destroyed. In this situation the utilisation of the processing hardware resource decreases, causing an increase of the spare capacity. As a consequence this spare capacity can be distributed to all of the currently active VRs by adapting their parameters.

Since the decision making process for the selection of VR parameters is an online task, there will be a trade-off between the selection of optimal VR parameters and the maximal affordable processing time for this task.

These mixed application and system requirements give us the motivation for an online anytime algorithm that is capable of distributing the spare capacity available in a given system resource. This resource is commonly a processor running tasks, but may also be a network transmitting message streams. The algorithm presented in this paper is designed for fixed priorities, although we believe that it could be easily extended to other scheduling policies. We call this algorithm SCD, for *spare capacity distribution*.

The SCD algorithm is based on the idea of maximising the utilisation of the processing hardware resource, whilst optimising the QoS of the applications as dictated by their importance and weight. The two attributes, importance and weight, specified in each contract allow the system integrator to control the spare capacity distribution among active applications in the system by defining their significance to the system and relative to each other. The SCD algorithm maximises the hardware resource utilisation by modifying the VRs' temporal attributes during runtime (i.e., budget, period, deadline and as a consequence their priorities as well), under the condition that their temporal requirements are not violated. The search for a feasible spare capacity distribution is an incremental process and it is based on bisection that provides the foundation for a simple and efficient implementation of the presented algorithm in a runtime framework. Furthermore, the algorithm uses response time analysis to test for schedulability, and since this method is known to be exact no schedulability losses are incurred by the test itself. The transition to a system state where the new temporal values and priority ordering are used, is performed at a feasible time instant (e.g., at an *idle instant* [21], or according to the idle instant optimised protocol [11]). The protocol for the change from one set of applications to another one, is currently a subject of extensive research. The purpose of this paper is to present and evaluate the SCD algorithm.

3. Related work

In real-time systems, servers are often used as a resource reservation mechanism to accomplish temporal partitioning among running application components. They provide a budgeting mechanism, which prevents malicious applications from affecting the operation of other applications in the system. Although server concepts [19, 20, 16] have been introduced to improve the response time of aperiodic tasks, their application has been adapted to provide temporal partitioning among tasks [1].

The composition of independently developed applications has been considered by Deng et al. [9]. They defined a scheme for a two-level hierarchically scheduled open system. Their work was based on dynamic scheduling. Kuo and Li [13] later adapted Deng's approach [9] to a fixed-priority operating system scheduler.

The FIRST[2] project addressed the need for a scheduling framework that could handle applications with varying processing demand [2]. The algorithm to adapt the server parameters used utilisation-based schedulability tests which are not exact, thus causing a lower resource usage in the system.

Utilised in this paper are the improvements to the exact schedulability test for fixed-priority tasks that were presented in [8] by Davis et al. and in [7] by Davis and Burns. In our case, the exact schedulability test is applied to VRs and determines whether the full amount of each VR's budget can be utilised before its deadline by the associated application tasks.

The server attributes, *importance* and *weight*, that are related to the spare capacity distribution, were intro-

[1] Framework for Real-time Embedded Systems based on COntRacts [10]

[2] Flexible Integrated Real-Time Systems Technology

duced in [2]. They were further adopted for VRs in [10]. These attributes allow the applications to influence the outcome of the spare capacity distribution.

Rosu et al. [18] described an adaptive resource allocation mechanism for distributed real-time systems. The expected application resource needs are specified by configurations. The choice of the appropriate configuration and the resulting resource allocations depends on environmental states, availability of resources in the system and the achievable system performance. The resource allocation is carried out as a response to events in the environment and changes in the processing demand of a complex distributed application.

Resource adaptive soft real-time systems were considered by Lin et al. [15]. Here, the Rate-Based Earliest Deadline (RBED) scheduler uses a heuristic algorithm to increase the overall benefit by adjusting the QoS level of the adaptive soft real-time tasks. Since the resource demand varies with the QoS levels, the processing of the adaptive tasks is adjusted so that they can be accommodated on the available resources.

An elastic tasking model has been defined by Buttazzo et al. [5, 6], where the task periods can be adjusted in order to adapt to different load conditions.

Rajkumar et al. [17] introduced a QoS based resource allocation model. Resource allocation to applications is made in terms of resource utilisation, but it is the application's responsibility to choose the appropriate budget and period. The algorithm that determines the resource allocation, requires QoS functions representing resource dependent changes of the application's contribution to the system value. However, the definition of such a function is not always straightforward.

Almeida et al. [3] presented an approach to adapt during runtime the temporal parameters of flexible periodic tasks. Each task's execution time and period is expressed as an *n*-tuple vector for *n* different QoS levels. From the set of all possible combinations of task parameters, a set of schedulable configurations is deduced by an offline method. This set is used by the online QoS manager to adapt the task parameters.

The SCD algorithm presented in this paper addresses the following issues of previous work: The resource allocation model in [18] is designed for high performance distributed systems, rather than for embedded real-time systems. Compared to the resource allocation model in [15], we focus on fixed-priority scheduled systems. The elastic task model [5, 6] addresses resource adaptation by changing task periods but not their allocated processing times. In contrast to the scheduling framework in [2], the SCD algorithm can be used as an anytime algorithm, and terminate after a maximum execution time providing non-optimal but schedulable VR parameters. Furthermore, loss of distributable spare capacity due to the schedulability test of VRs is prevented by using an exact test. The adaptation method presented in [3] conflicts with the notion of open systems due it assumption of static set of tasks that are known before runtime, and is therefore not well applicable to open systems.

4. System model

It is assumed that a set of flexible application components $\{A_1, A_2, \cdots, A_m\}$ are mapped to a set of virtual resources $\Gamma = \{S_1, S_2, \cdots, S_n\}$, with $m \leq n$.

The tasks of each flexible application component are mapped to and are executed by one or more VRs. In the former case the application's taskset is mapped to one VR, where in the latter case the taskset is divided into subsets and each subset is mapped to a VR. Unless a one-to-one mapping of task to VR is used, a hierarchical scheduler can be utilised to determine the order of execution of tasks on the same VR. The presence of one or more tasks on a single VR does not affect the spare capacity distribution, as it is done solely for VRs according to their requirements specified in their respective contracts.

The tasks of one application component are not mixed with tasks of other components in the same VR, in order to ensure the temporal protection among them.

Each *virtual resource* S_i is characterised by its processing *budget* C_i, replenishment *period* T_i, *deadline* D_i, *importance* I_i and *weight* W_i.

Each virtual resource S_i is either defined as a continuous or discrete VR [10] depending on the domain of its temporal parameters:

- For continuous VRs, the operational ranges of period and budget are defined by a lower and upper bound, $[T_i^{min}, T_i^{max}]$ and $[C_i^{min}, C_i^{max}]$, respectively. The actual value assigned to a VR's temporal attribute can take any value from within corresponding operational ranges. The budget and period of continuous VRs are independent of each other and therefore can be adjusted independently.
- For discrete VRs a finite set of (C_j, T_j, D_j) triples are defined. Only values from this set of (C_j, T_j, D_j) triples can be assigned to discrete VRs' temporal attributes. This definition implies that temporal attribute values are linked to each other.

The budget C_i denotes the maximal processing time that can be consumed by a VR before it is suspended. The budget is replenished to its full amount after every period T_i.

The deadline D_i of a VR specifies the relative time from the point when it has been released until it has to complete the supply of its budget to the associated application component. The deadline D_i of a virtual resource S_i can be defined to be:

1. equal to its period T_i for a continuous VR,
2. equal to D_j from the selected (C_j, T_j, D_j) triple for a discrete VR or,
3. a constant value for both types of VRs.

Additionally, it is assumed that the deadline of a VR is always less than or equal to its period, $D_i \leq T_i$.

The priority P_i of a VR S_i is assigned to it using a

fixed-priority assignment policy. The priority of VRs may change during the spare capacity distribution, but it remains fixed during the steady operational state of the system. If a VR's temporal attribute, which is used by the priority assignment policy, is modified during the execution of the SCD algorithm, then the priority assignment has to be updated. Nevertheless, the constraint $D_i \leq T_i$ is not violated by the modification of the period.

For a VR S_i two of its attributes, *importance level* I_i and *weight* W_i, are used to influence the outcome of the SCD algorithm. The precedence, in which the spare capacity is assigned to VRs is determined by the importance level I_i of the VRs. For the spare capacity distribution, VRs with the same importance level are logically combined into groups. A group G_l is the set of all VRs with the same importance level l (see Equation1).

$$G_l = \{S_i \in \Gamma | I_i = l\} \tag{1}$$

Each major iteration of the SCD algorithm is limited to a group G_l of VRs. Usually several major iterations of the SCD algorithm are required until a solution is found for the given set Γ of VRs.

The *weight* attribute W_i influences the fraction of the spare capacity that a VR will get. The fair share value [12] denoted as H_i is used as a factor to determine the fraction of additional capacity that a virtual resource S_i will get when a certain amount of spare capacity is distributed at the currently examined importance level I_C (see Equation 2). In this equation, I_j denotes S_j's importance level.

$$H_i = \frac{W_i}{\sum_{\forall j: I_j = I_C} W_j} \tag{2}$$

The usage of shared resources and the consequent blocking factors have no direct impact on the SCD algorithm itself. Therefore, there are neither assumptions nor restrictions on the usage of shared resources, since the corresponding blocking factors affect only the schedulability test.

The *utilisation* caused by a virtual resource S_i on a processor is the amount of assigned budget C_i per replenishment period T_i. The processor's capacity that is unused by VRs is denoted as *spare capacity*.

The processor's utilisation is calculated as the sum of all VRs' utilisation. The largest utilisation of the processor, determined by the SCD algorithm, at which the set of VRs in the system becomes unschedulable is referred to as the *breakdown utilisation*. The breakdown utilisation mainly depends on the VR types in the system. With only continuous VRs it is likely that close to 100% processor utilisation is achieved since the utilisation assigned to these VRs can be gradually increased. Whereas with discrete VRs the largest processor utilisation is likely to be significantly less than 100% due to the limited number of VR budget and period values.

5. Spare capacity distribution algorithm

5.1. SCD algorithm characteristics

The intention of the SCD algorithm is to allocate as much as possible of the processor's spare capacity U_s to VRs in the system. Of course, after the application of the SCD algorithm, the set of VRs with their new utilisation values has to be schedulable.

The utilisation probe U_p is the amount of spare capacity that can be distributed at once among the VRs at the processed importance level.

Since the presented algorithm is a search based approach, feasibility of various probe values need to be tested. An efficient approach to find a feasible probe value $U_p \in \{0, 0 + \delta U_p, \ldots, U_s - \delta U_p, U_s\}$ is provided by bisecting the interval $[0, U_s]$ and applying the binary search algorithm to it. The binary search algorithm is fast, has minimal memory requirements and the required processing resources are also low.

For a given set of VRs, the maximal distributable spare capacity can be found within $1 + \lceil log_2 N \rceil$ iterations, where N is the number of potential values for U_p. Given the granularity δU_p of the interval $[0, U_s]$, the number of potential values for U_p can be calculated as $N = \frac{U_s}{\delta U_p}$. For example, a typical value of 1% for δU_p and the possible maximum value for U_s of 100%, limits the number of potential values N for U_p to 100. By applying the binary search on the 100 possible values, a feasible U_p can be determined by checking the schedulability of $1 + log_2 100 = 8$ different spare capacity distribution scenarios.

A virtual resource S_i is defined to be available/active for spare capacity distribution if it is able to increase its utilisation due to the following conditions:

- During the last iteration of the spare capacity distribution, S_i did not render the system unschedulable,
- S_i has not reached its predefined maximal utilisation and can therefore utilise a higher spare capacity allocation.

In order to determine the optimality of the SCD algorithm the following assumption is made, driven by the FRESCOR project: The spare capacity supplied at higher importance levels is considered infinitely more valuable than at lower importance levels. This implies that different importance levels are incomparable. The SCD algorithm starts with the spare capacity distribution at the highest importance level. Spare capacity is supplied by decreasing the periods and increasing budget of VRs at this importance level. Only after all possible spare capacity has been allocated at the highest importance level, does the algorithm considers the next highest level and so on. Hence, under the assumption that importance levels are incomparable, the SCD algorithm provides the optimal spare capacity distribution.

The schedulability test used by the SCD algorithm is an exact test for fixed-priority scheduled uniprocessor real-time systems [8]. Before the SCD algorithm is ap-

plied to the VRs in the system, the schedulability using their minimal timing requirements is ensured. Changes to the set of the VRs in the system are only permitted if the changed set remains schedulable using the minimal temporal requirements of the VRs.

5.2. Description of the SCD algorithm

As input, the SCD algorithm requires a priority ordered list of VRs along with their temporal attributes. This *priority ordered list (*Π*)* contains VRs that want to benefit from the additional assignment of spare capacity. Π is also used as the output list.

The SCD algorithm starts with the VRs' temporal parameters set to their minimum timing requirements. The minimum timing requirement is either the minimum budget C_i^{min} and largest period T_i^{max} in the case of a continuous VR, or (C_j, T_j, D_j) triple with the smallest utilisation in the case of a discrete VR. A soon as the SCD algorithm finds an intermediate or final solution, the new temporal values of the VRs are stored in Π.

With the SCD algorithm, at each importance level the search for the largest feasible U_p is carried out in order to determine a feasible spare capacity distribution (see Algorithm 1).

InOut: Π: Priority ordered list of VRs
foreach *importance level l (in decreasing order)* **do**
  Determine H_i for all S_i^l that are active;
  while *not all possible* U_p *values checked* **do**
    Calculate ΔU_i (see Equation 3) and increase the current utilisation of S_i^l to U_i^{new} by ΔU_i;
    Determine S_i^l new parameters using Algorithm 2;
    Determine new priority ordering for the schedulability test;
    if *all VRs are schedulable* **then**
      Store new priority ordering and temporal values of all active S_i^l in Π;

Algorithm 1: Spare capacity distribution

Each major iteration of the SCD algorithm is limited to VRs at a particular importance level, starting at the highest level.

The steps of the SCD algorithm that are executed at every importance level are as follows:

1. Calculate the fair share values.
   The fair share value H_i of every active virtual resource S_i at the currently processed importance level is calculated using Equation 2. This value is used to determine the fraction of capacity that will be assigned to the active VRs.
2. Search for a feasible spare capacity distribution.
   The algorithm considers only VRs that are active (i.e. capable of accepting more than their current utilisation) at the currently examined importance level. Using binary search (represented by the *while*-loop condition in Algorithm 1), the distributable spare capacity U_p is narrowed down to the largest value at which the system is schedulable, but where an additional amount (δU_p) of spare capacity assignment (i.e. $U_p + \delta U_p$) would cause an infeasible schedule.
   For each utilisation probe U_p, the temporal parameters of active VRs at the currently examined importance level are recalculated using Algorithm 2, where U_i^{new} denotes the increased utilisation of S_i by ΔU_i.
   $$\Delta U_i = U_p \cdot H_i \quad (3)$$
3. Increase active VRs' utilisation.
   If a schedulable spare capacity distribution has been found, the new VR temporal parameters are stored in the output list Π regardless of whether these values are intermediate or final.

One possible approach to determine the budget and period of a continuous VR S_i (with an increased utilisation equal to U_i^{new}) is presented in Algorithm 2. If possible, the VR's smallest period T_i^{min} is used and its budget is calculated accordingly. If it is not feasible to use its smallest period, then the minimum budget C_i^{min} is fixed first and the period is calculated accordingly. Under both circumstances, the calculated values are restricted to the specified interval for the budget and the period, respectively.

if $(C_i^{min}/T_i^{min}) > U_i^{new}$ **then**
  $C_i = C_i^{min}$;
  $T_i = \min\big((C_i^{min}/U_i^{new}),\ T^{max}\big)$;
else
  $T_i = T_i^{min}$;
  $C_i = \min\big((T_i^{min} \cdot U_i^{new}),\ C^{max}\big)$;

Algorithm 2: Budget and period calculation

For a discrete VR, the utilisation values of its different discrete (C_j, T_j, D_j) triples are calculated. Then the (C_j, T_j, D_j) triple with the biggest utilisation value, which is less than or equal to U_i^{new}, is selected.

5.3. Priority assignment

In this paper, the deadline-monotonic priority assignment [14], as provided by the FRESCOR framework, was used for the empirical evaluation. The implementation of the SCD algorithm considers two different VR configurations for the deadline:

1. Virtual resource's deadline is equal to its period,
2. Virtual resource's deadline is constant.

In the case that a VR's deadline is configured to be equal to its period, whenever the period is changed during the spare capacity distribution its deadline is adjusted accordingly and priority reordering is carried out.

Furthermore, the schedulability test of the VRs is carried out in decreasing order of their priorities.

6. Empirical evaluation

6.1. Test data generation

To evaluate the performance of the SCD algorithm, VR sets were generated where the variation of particular isolated test-case parameters was examined. For each

of these VR sets 100000 different configurations were created. Each configuration consists of the predefined number of VRs (i.e. 5, 10, 15, ..., 50) for which randomly generated VR parameters (i.e. budget, period and deadline) were created.

The approach, presented by Bini and Buttazzo [4], to create tasks parameters for a given maximal utilisation (subsequently referred to as the *Initial Target Utilisation (ITU)*), was used in this work to generate the random VR parameters.

The ITU was chosen so that the performance of the spare capacity distribution could be observed in different scenarios where more or less spare capacity was available. The chosen ITUs were 30%, 50% and 80%. Before the VR sets were processed by the SCD algorithm, it was ensured that the generated sets were schedulable at the chosen ITU. Test-cases with initially unschedulable VR sets were not considered in the measurement and they were replaced with a new and schedulable VR set.

The period and budget of the VR is derived from its randomly generated utilisation value. First, the VR's period is chosen according to a uniform random distribution from a randomly selected period magnitude range (i.e. $[10^3, 10^4]$, $[10^4, 10^5]$, $[10^5, 10^6]$ or $[10^6, 10^7]$). Given the VR's utilisation and period, the calculation of its budget is straight forward.

To take advantage of the flexibility of VRs, a definition for the upper and the lower bound of their utilisation ranges is required. For each VR the initially generated random utilisation values are considered in the tests as their lower utilisation bounds. This lower utilisation bound is used to derive the VR's minimum budget and maximum period. The minimum budget is then multiplied and the maximum period is divided by a factor in order to determine the VR parameters (i.e. maximum budget and minimum period) for its upper utilisation bound. A factor of 2.0 for the 30% ITU, and a factor of 1.5 for 50% and 80% ITU was used.

For discrete VRs, additional to their lower and upper utilisation bounds, a random number of intermediate utilisation values were generated. The number of intermediate values was in the range of 1 to 3, where 5 is the maximum number of discrete temporal attributes defined by the FRESCOR framework.

6.2. Experiment evaluation

This section evaluates the data, which was collected from various measurements by varying different parameters as indicated in Section 6.1. The diagrams show the progress of the spare capacity distribution as a function of the number of ceiling operations needed by the schedulability analysis, a useful proxy for algorithm execution time [8]. This section uses the number of ceiling operations to give insight into the complexity of the algorithm. Section 7.2 extends this information by mapping it onto absolute time. In the following, the term *number of ceiling operations* will be referred to as *number of iterations*.

The presented experiments contain a mixed set of VRs (both continuous and discrete). The only exception is the experiment in which the impact of the different temporal attribute types (i.e. discrete or continuous) on the SCD algorithm is examined.

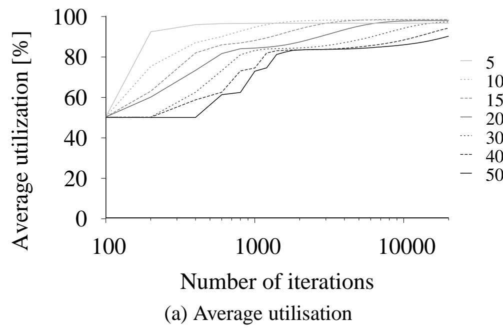
For the sake of clarity, the diagram types used to support the empirical evaluation are introduced in Figure 1. Of main interest are the following properties of the SCD algorithm: how fast can the spare capacity be distributed and the number of iterations required by the algorithm to terminate. The average increase of processor's utilisation as the SCD algorithm progresses, is depicted in Figure 1a. The percentage of test-cases terminated by a certain number of iterations is depicted in Figure 1b. This figure can also be interpreted as the probability that the SCD algorithm will terminate within a given number of iterations for a given number of VRs.

For all experiments the SCD algorithm was executed until it terminated itself. The processor's utilisation achieved in this way is the highest schedulable utilisation of the active VRs in the system.

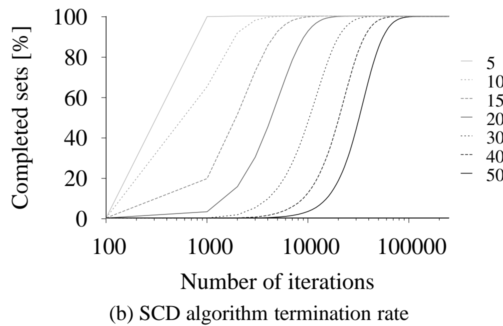
In Figure 1 the effect of the number of VRs on the SCD algorithm is examined. Figure 1a shows that the rate of the average utilisation increase from an ITU of 50% slows up when the number of VRs in the system is increased. The number of iterations required to terminate with a particular probability also increases with the number of VRs (see Figure 1b). The diagrams indicate that more iterations are required to find a feasible spare capacity distribution as the number of VRs in the system increases. This comes about due to the increased number of schedulability tests that are required for each VR utilisation level.

In dynamic real-time systems the processor's utilisation at which application components are submitted into the system cannot be predicted in advance. Therefore, the impact of the initial processor's utilisation on the progress of the SCD algorithm was also considered (see Figure 2). In Figure 2a it can be observed that the ITU has an effect on the processor's utilisation increase only at the beginning of the algorithm. In the long term (i.e. above 2000 iterations) the ITU becomes irrelevant and the utilisation increase is dominated by the number of VRs in the system. The probability for the termination of the SCD algorithm is influenced from the beginning by the number of VRs and the ITU has only a negligible effect (see Figure 2b). The effect of the ITU is small since the number of schedulability tests that are carried out during the runtime of the SCD algorithm stay nearly identical for various ITUs.

The VR's temporal attributes can be either of continuous or discrete type. Hence, there can be three different VR sets in the system. Only continuous, only discrete or a mixed set of VRs. Our experiments show that the temporal attribute type has only a minor effect on the increase of the processor's utilisation and the runtime of

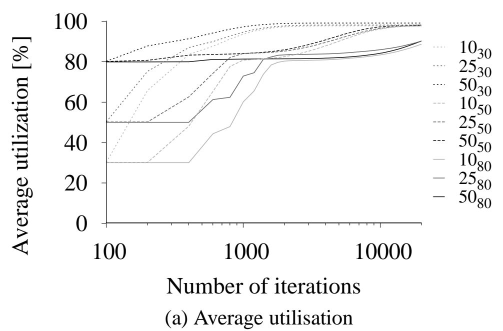


(a) Average utilisation

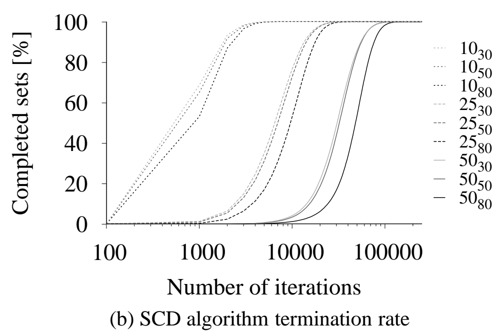


(b) SCD algorithm termination rate

Figure 1. Number of VRs: 5, 10, ..., 50



(a) Average utilisation

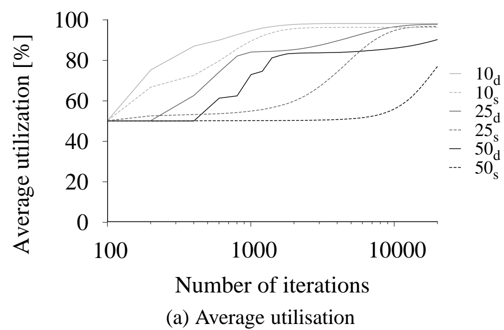


(b) SCD algorithm termination rate

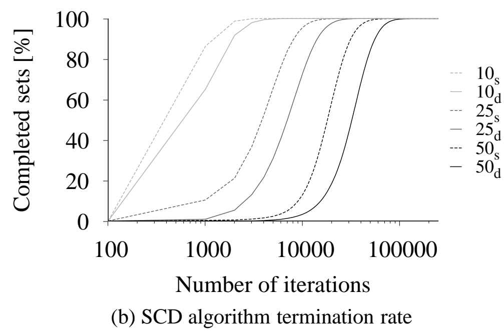
Figure 2. ITU: 30%, 50% and 80%

the SCD algorithm. For space reasons, these results are not included in this paper but are available in a technical report [22].

The VR assignment to importance levels has been analysed in two scenarios (see Figure 3). In one scenario, the VRs have been randomly distributed among the available importance levels. In the other scenario, they have been assigned to a single importance level. Figure 3a shows that the use of a single importance level had a negative effect on the processor's utilisation increases. These are much slower although in long term, the processor's utilisation values for both scenarios converge towards each other. On the other hand, the number of iterations required for the termination of the algorithm decreased if a single importance level was used (see Figure 3b). In this case, performing the schedulability test for the empty importance levels was avoided, which led to the reduction in iterations.



(a) Average utilisation



(b) SCD algorithm termination rate

Figure 3. Importance level allocation

The measured data, presented in this section, reveal that the performance of the SCD algorithm is mainly influenced by the number of VRs in the system.

7. Performance evaluation

This section provides information about the embedded hardware that was used for the performance measurements and the absolute values for the execution times obtained. Additionally, using a pragmatic approach, a linear upper bound equation is derived for the execution time of the SCD algorithm. This allows us to determine the required budget for the SCD algorithm, in order to provide a complete or partial solution for the spare capacity distribution of the system.

7.1. Test environment

The empirical evaluation, which was presented in Section 6, is extended by performance measurements on an embedded platform. The intention of this task is

to determine which parameters influence the SCD algorithm's execution time.

In order to exclude operating system and other undesirable overhead, an embedded system was configure in a way that only the spare capacity distribution was executed. This provided the facility to obtain absolute values for the execution time of the SCD algorithm.

The embedded platform consisted of an MPC555 microcontroller running at 40 MHz system clock.

To create the target executable file, a development environment comprising the GNU C compiler and RapiTime version 1.2, a tool for worst-case execution time analysis, was used. The executable files were optimised by the compiler using the option *-O2*. To avoid falsification of the measurements by intermediate performance monitoring of the SCD algorithm, only end-to-end execution times were recorded.

Due to the limited performance of the embedded system, the number of test cases used was reduced to 3000 randomly generated VR sets for each fixed number of VRs in the set (i.e. 5, 10, 15,..., 45, 50 VRs). The ITU (i.e. 30%, 50% or 80%) and the type of the VR sets (only continuous, only discrete or mixed type) were randomly created for each of the 3000 sets. Nevertheless, sufficient data was captured to analyse the behaviour of the SCD algorithm on real hardware.

7.2. Measurement

During the design phase of real-time systems, information is required about the expected complexity of the SCD algorithm. In the following, an upper bound equation will be defined that allows engineers to determine during the design phase the required amount of budget for the SCD algorithm.

As an initial step, the dependency of the SCD algorithm's execution time on the number of iterations and VRs in the system was examined. Figure 4 shows for different numbers of VRs the execution time plotted against the number of iterations and the corresponding regression lines. Since the scatter of the measured values along the x-axis is very narrow for the test-cases with 5, 10 and 15 VRs, their regression lines are omitted.

To determine the necessary values for the linear upper bound equation, the least-squares linear regression method was applied to the collected data. Table 1 summarises the parameters of the regression lines from Figure 4. The data in Table 1, as well as visual inspection of Figure 4 indicate that the slope of each regression line is very similar. This observation suggests a linear dependency of execution time on the number of iterations.

But a single linear equation is not sufficient to express the execution time of the SCD algorithm. Since the regression lines do not overlap but have an offset between them, the dependency of this offset on the number of VRs has been examined as well. In Figure 4 the intersection points of the regression lines with z-y-plane shows that the offset also increases linearly with the number of VRs in the system.

Table 1. Regression line parameters

VR#	Function	Slope (10^{-3})	Y-axis offset
50	$f_{50}(x)$	1.5425	99.287
45	$f_{45}(x)$	1.5520	87.238
40	$f_{40}(x)$	1.5579	75.519
35	$f_{35}(x)$	1.5722	64.308
30	$f_{30}(x)$	1.5862	53.022
25	$f_{25}(x)$	1.6263	42.277
20	$f_{20}(x)$	1.6386	32.486

The equation, which describes the y-axis intersection of the execution time regression lines against the number of VRs, was also determined via linear regression (see Equation 4).

$$y_0(v) = 2.381 \cdot v - 21.397 \quad (4)$$

The former analysis reveals that the execution time depends mainly on two parameters. Figure 4 shows the execution times plotted along the y-axis. The x-axis represents the number of iterations and the z-axis the number of VRs. The observable linear behaviour of the plotted data along the x-axis as well as along the z-axis suggests the definition of the execution time upper bound as a plane equation with the number of iterations and VRs as independent variables.

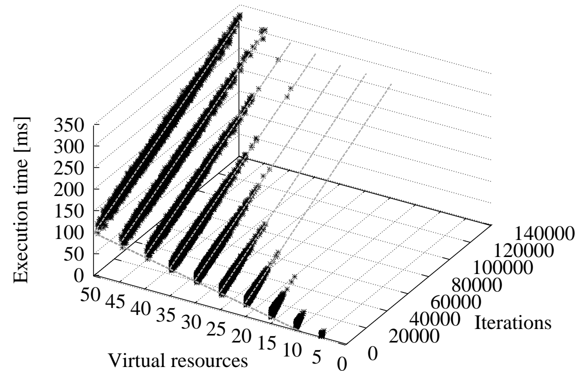


Figure 4. Execution time samples

The general form of the plane equation can be defined as $C(n, v) = a_1 \cdot n + a_2 \cdot v$, with $C(n, v)$ representing the execution time, n the number of iterations and v the number of VRs.

Next, the coefficients, a_1 and a_2, for the plane equation were specified. They were derived from the data that was obtained by measurements on the embedded platform.

The first coefficient, a_1, was determined by calculating the average slope of the execution time regression lines. The average value is approximately $1.58224 \cdot 10^{-3}$, but for ease of use, this value has been rounded up to $1.6 \cdot 10^{-3}$.

Finally, the value of the second coefficient, a_2, was defined. Based on Equation 4 and on the data of Figure 4, the value of 2.8 was determined for a_2. This value was obtained by the application of pragmatic steps in or-

der to simplify Equation 4. The aim was to preserve just a single factor that expresses the dependency between the number of VRs, and the y-axis intersection of the lines that represent the execution time upper bounds for each set of VRs. The value of coefficient a_2 was increased from the value starting at 2.381 (as specified by Equation 4) until the regression lines in Figure 4 became the upper bound on the measured execution times for the corresponding set of VRs (i.e. every execution time sample was below the upper bound).

Using this information, an execution time upper bound equation for the SCD algorithm was derived (see Equation 5). The equation represents a pragmatically derived execution time upper bound for the MPC555 microcontroller that was used to carry out the performance measurements.

$$C(n, v) = 0.0016 \cdot n + 2.8 \cdot v \quad (5)$$

To get an idea of how the execution time upper bound increases with system complexity, the upper bound was plotted against the number of VRs in the system and the number of iterations. The data that was generated by the application of Equation 5 spans a plane in three dimensional space. The plane in Figure 5 illustrates the execution time upper bound of the SCD algorithm. For the sake of clarity contour lines are rendered at 50 ms intervals.

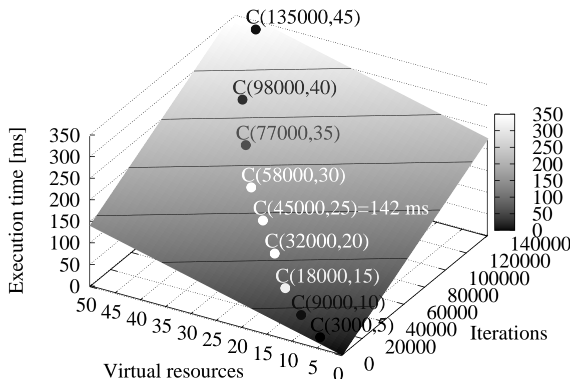


Figure 5. Execution time upper bound

In order to determine the required budget for the SCD algorithm, the expected maximal number of VRs in the system and the granted maximal number of iterations for the runtime of the algorithm has to be specified. The parameter, maximal number of iterations, also influences the probability of the SCD algorithm terminating within the determined budget. The probability of termination within a certain number of iterations has already been investigated during the empirical evaluation in Section 6.2. It can be summarised as follows.

Table 2 shows the maximal number of iterations that were required by the SCD algorithm to terminate for 99.99%, 99.90%, 99.00% and 90.00% of the test-cases. There is a slight difference in the number of required iterations for the algorithm among the test-cases that were performed with 30%, 50% and 80% ITU; but the greatest observed values were chosen.

Table 2. Number of iterations

VR#	99.99%	99.90%	99.00%	90.00%
5	3000	2000	1000	1000
10	9000	6000	4000	3000
15	18000	14000	10000	6000
20	32000	24000	18000	12000
25	45000	36000	27000	18000
30	58000	49000	38000	26000
35	77000	66000	51000	36000
40	98000	86000	68000	49000
45	135000	108000	85000	62000
50	157000	132000	107000	77000

Based on the data that was collected during the empirical evaluation and the performance measurements, the required budget for the SCD algorithm can be derived for a real-time system using an MPC555 microcontroller and a specified maximum number of VRs.

As an example, we now determine the budget for two different configurations. First, the budget for the SCD algorithm on a system with a maximum of 5 application components is calculated. The budget is chosen such that the algorithm can terminate in 99.99% of the cases. Applying the information from Table 2 in Equation 5, provides a budget estimate of $C(3000, 5) = 0.0016 \cdot 3000 + 2.8 \cdot 5 = 18.8ms$. For the second example, we assume a system with at most 25 components. Again, the budget should allow the SCD algorithm to terminate in 99.99% of the cases. Hence, the estimated budget is $C(45000, 25) = 0.0016 \cdot 45000 + 2.8 \cdot 25 = 142ms$. The calculated values for the examples are also illustrated in Figure 5.

The two coefficients, a_1 and a_2, of the linear equation depend on various hardware factors, like the availability and size of data caches, data bus bandwidth, external memory speed, etc. They also depend on the location (i.e. internal or external memory) of the relevant processing data. Therefore, the linear equation representing an upper bound for SCD algorithm's execution time, has to be individually derived for each hardware platform. This can however easily be performed using a suitable program for calibration, such as the one used to generate the results presented here.

8. Conclusion

In this paper we presented an easily and efficiently implementable algorithm for the distribution of a processing resource's spare capacity. The contribution can be summarised as:

- an anytime SCD algorithm that can generate useful results even if the algorithm's execution time is limited,
- runtime adaptation of continuous and discrete VR temporal parameters (i.e. budget and period),
- preventing schedulability test based inefficiency using an exact test.

The efficiency of the SCD algorithm was examined by empirical evaluation and performance measurements on an embedded platform.

The performance evaluation of the SCD algorithm shows that, for example in a system with up to 25 mixed VRs, the algorithm terminates in 99.99% of the cases within 45000 iterations, and within the same number of iterations the processor reaches an average utilisation of 98%. Based on the upper bound equation (Equation 5), the worst-case execution time for 45000 iterations and 25 VRs is equivalent to 142 ms on an MPC555 microcontroller with a system clock of 40 MHz. The MPC555 is not however a sufficiently powerful processor to support 25 application components.

On faster processors, the cost for one iteration of the SCD algorithm, as well as the overall execution time, decreases. Therefore the complexity of a system, measured in terms of the number of active VRs, for which the SCD algorithm can be considered applicable, increases. For example, a processor with approximately 10 times the performance of a 40MHz MPC555 might be used in Avionics or Telecommunications applications that need to support 10 to 25 application components. In this case, such a processor would require at most 15ms to execute the SCD algorithm.

The SCD algorithm shown in this paper has been integrated into the FRESCOR framework, giving it the capability to distribute spare capacity among different application components with the objective of fully using the available processor capacity and maximising the QoS. While the algorithm is based on the current implementation of FRESCOR on fixed priorities, we believe that it is easily extensible to other scheduling policies and tests. As future work we will investigate the mode change protocols used to make effective the calculation of a new resource allocation as a result of the SCD algorithm. Furthermore, the allocation of multiple resources (e.g. processor, network bandwidth, memory, etc.) to application components and their interdependency will be examined.

Acknowledgements

This work was funded in part by the EU FRESCOR project (contract number FP6/2005/IST/5-034026). We would also like to thank Rapita Systems Ltd.[3] for providing the necessary equipment and tools for the performance measurements. Furthermore, we would like to thank Daniel Sangorrín López for his valuable comments.

References

[1] L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec 1998.

[2] M. Aldea Rivas, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, T. L. J. Javier Gutiérrez Garcia, G. Lipari, J. L. M. P. José M. Martínez, J. C. Palencia Gutiérrez, and M. Trimarchi. Fsf: A real-time scheduling architecture framework. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–124, 2006.

[3] L. Almeida, S. Fischmeister, M. Anand, and I. Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *Proceedings of the 7th ACM & IEEE international conference on embedded software*, pages 67–74, 2007.

[4] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[5] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 286–295, Washington, DC, USA, 1998. IEEE Computer Society.

[6] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[7] R. I. Davis and A. Burns. Response time upper bounds for fixed priority real-time systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008.

[8] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.

[9] Z. Deng, J. W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Toledo, Spain, Jun 1997.

[10] M. González Harbour and M. T. de Esteban. Architecture and contract model for integrated resources. Technical Report D–AC.2v1, Universidad de Cantabria, 2007.

[11] M. González Harbour, D. S. López, and M. T. de Esteban. Mode change protocol for budget changes in contract-based scheduling. Technical report, Universidad de Cantabria, 2008.

[12] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[13] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 256–267, 1999.

[14] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, Dec 1982.

[15] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse soft real-time processing in an integrated system. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 369–378, 2006.

[16] J. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[17] R. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A resource allocation model for qos management. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[18] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time application. In *Proceedings of the 18th IEEE International Real-Time Systems Symposium*, pages 320–329, 1997.

[19] L. Sha, J. P. Lehoczky, and R. R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proceedings of the 7th IEEE Real-Time Sytems Symposium*, pages 181–191, 1986.

[20] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[21] K. W. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

[22] A. Zabos, R. I. Davis, and A. Burns. Utilization based spare capacity distribution. Technical Report YCS-2008-427, University of York, 2008.

[3] Rapita Systems Ltd. Available at http://www.rapitasystems.com (8 January 2009)

Software Transactional Memory: Worst Case Execution Time Analysis

Toufik Sarni and Audrey Queudet
LINA - University of Nantes
France
FirstName.LastName@univ-nantes.fr

Patrick Valduriez
LINA and INRIA
Nantes - France
Patrick.Valduriez@inria.fr

Abstract

While real-time applications are becoming more and more concurrent and complex, the drive toward multicore systems raises new challenges related to the parallelization of such performance-critical applications. Transactional memory is an attractive concept for expressing parallelism for programming multicore systems as it avoids the problems of lock-based methods and eases programming. However, it has not yet been exploited for real-time systems. In this paper, we propose the first real-time directed case study of software transactional memory. In particular, our goal is to identify the origin of the variation of the worst-case execution times (WCET) of transactions in memory. Based on a real implementation, we show through various experiments that for soft real-time, transactions rollback times are not the main cause of execution times variation. A good memory allocator must also be provided in order to suitably bound the WCETs of transactions into software transactional memory.

1 Introduction

With the advent of multicore systems, the transactional memory (TM) concept has attracted much interest from both academy [1, 2] and industry [3] as it eases programming and avoids the problems of lock-based methods. By supporting the ACI (Atomicity, Consistency and Isolation) properties of transactions, TM relieves the programmer from dealing with locks to access resources. More important, it avoids the severe problems of lock-based methods such as deadlock situations and priority inversions. While lock-based methods systematically block all accesses to shared resources, transactional memory allows several transactions to access resources in parallel. A transaction is either aborted when a conflict is detected, or committed in case of successful completion. Conflicts are handled with non-blocking synchronization which offers a stronger guarantee of forward progress.

There are three kinds of implementations for transactional memory: hardware-based memory (HTM) [1, 4], software-based memory, denoted as software transactional memories (STM) [2, 5, 6, 7] and hybrid schemes (HyTM) that combine both hardware and software supports [8, 9]. HTM researchers mainly focus on implementation with less attention to performance. On the contrary, STM researchers take care about performance issues on TM, and several policies [10, 11] have been proposed to manage conflicts between transactions.

While software transactional memories are widely studied for numerous and various purposes, they have not yet been studied for real-time systems. However, we believe that the advantages of transactional memory can also be brought to real-time systems. Thus, we propose to study how to adapt it to soft real-time systems. For this purpose, we aim to identify which parts of STM cause WCET variations. It is often claimed that transaction rollback times are the main cause of unpredictability in execution times. However, the recent STMs are usually dynamic memory based. We show in this paper that STM memory allocators require more consideration than rollback times in order to bound the execution time of transactions. Furthermore, we show that transaction rollback times also depend on the time latencies of the underlying operating system. This is why we focused on selecting the best task scheduling policy minimizing the rollback times.

To the best of our knowledge, this paper is the first to study the WCET variation of STM based on a real implementation. The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the real-time scheduling of both tasks and transactions and presents the STM used in our experiments. Section IV presents both the issues identified for adapting STM to satisfy real-time constraints and their implementation. Section V gives an experimental analysis under several real-time scheduling policies of tasks and shows the impact of memory allocator on the STM. Finally, Section VI draws the main conclusions and discusses future work.

2 Related Work

Schoeberl *et al.* [12] propose a real-time HTM which uses the late conflict detection (*i.e* the conflict between transactions is detected on a commit). The transaction is either rollbacked on a conflict or aborted on context switch. The number of retries of the transaction is bounded and integrated into the WCET analysis. This

bound assumes one atomic region per thread period and allows having hard real-time constraints. However, we are interested by soft real-time STM, and the HTM presented by the authors assume that all critical sections resources need to be known.

Brandenburg *et al.* [13] compare wait-free and lock-free algorithms with spin-based and suspension-based synchronization mechanisms. They conduct experiments[1] using the real-time operating system LITMUS^{RT}. The four approaches are compared on the basis of both schedulability and tardiness bounds, by evaluating their respective overheads with respect to job release, scheduling and context-switching. One of the major conclusions of this work is that non-blocking algorithms are generally preferable for small, simple shared objects. Among non-blocking approaches, the authors conclude that wait-free algorithms are preferable to lock-free algorithms. Regarding scheduling policies, they show that, unlike partitioned-EDF (P-EDF), global-EDF (G-EDF) policy does not scale for lock-free algorithms when the access to shared objects occurs at high frequency.
The wait-free algorithms are primarily of interest in hard real-time transactions [14]. However, implementing a wait-free-based STM is very difficult since fair access to memory is usually not guaranteed.

Riegel *et al.* [15] deal with time-based transactional memory that uses time to reason about the consistency of the data accessed by transactions and the order in which transactions commit. Usually, implementations like [16, 17] rely upon shared counters which can quickly become bottlenecks as the number of concurrent threads grows.
Riegel *et al.* [15] show how a time base can affect transactional memory performance. They rely on experiments[2] which compare the use of a shared integer counter with that of a MMTimer which is a real-time clock with an interface similar to the High Precision Event Timer widely available in x86 machines. Their main observation is that this enhanced hardware support can ensure a much better clock synchronization than mechanisms that require communication via shared memory. As part of their work, the authors introduce the *Real-Time Lazy Snapshot Algorithm* (LSA-RT) which is a timestamp-based algorithm using a real-time clock. Moreover it uses a helper mechanism to help committing transactions to complete. However, the authors consider only throughput, and not WCET of transactions. Furthermore, they consider the time-based STM performance without tacking into consideration the impact of the operating system in which their STM is performed.

Yoo *et al.* [18] describe a scheduler for transactional memory. The authors compare their *adaptive transaction scheduler* to the traditional *Contention Manager* (CM). In CM-based STMs [19, 11], the transaction that encounters a conflict, consults its CM. When the CM retries the denied object, it typically employs an exponentially backoff scheme with a retry interval expanding exponentially to a maximum limit until success. Thus, a CM can decide to abort a certain transaction, but does not deal with *when* to resume an aborted transaction. In contrast, the scheduler presented by the authors, specially deals with *when* to resume the aborted transaction which is an important notion in a real-time context. However, the authors do not deal with any real-time constraints in their paper.

[1] The hardware platform used was a four 32-bit Intel Xeon(TM) processors running at 2.7 GHz

[2] using a 16-processor partition of an SGI Altix 3700 and a ccNUMA machine with Itanium II processors

3 Theoretical Background

3.1 Real-Time Task Model

We consider the scheduling of a sporadic task system τ on $m \geq 1$ processors. For each task $\tau_i \in \tau$ we associate a set of jobs $J = \{j_1, j_2, ..., j_n\}$. Task τ_i is characterized by a set of parameters r_i, C_i, P_i which respectively represent the task release, its execution requirement in the worst-case, and its minimal period of activation. At time $r_i + (k-1)P_i$ and for $k \geq 1$, a kth job is released, receives C_i units of processor time and should complete by its absolute deadline $d_i = r_i + kP_i$. The weight (or processor utilization) for a task τ_i on processor m is defined by $u_{i,m}= C_i/P_i$. We assume that at any time, a processor executes at most one job, and a job is executed at most on one processor.

Scheduling of tasks. On multiprocessor systems, two alternative paradigms for scheduling collections of tasks are considered: *partitioned* and *global* scheduling. In the partitioned approach, the tasks are statically assigned to processors and are always executed on a single processor. Each processor has its own scheduling queue of tasks which is independent of other processors and the migration of jobs or tasks on other processors is not allowed. Feasibility analysis under the partitioned paradigm which is comparable to a *bin-packing* problem, is NP-Hard. Indeed it consists in placing k objects with different sizes in m boxes which respectively represent the tasks and the processors in our case. First-Fit and Best-Fit algorithms and their variants [20] are usually used to assign tasks to processors with an appropriate condition in accordance with the schedulability analysis. In contrast, under the global scheduling approach, inter-processor migrations are allowed. A single queue and only one policy are applied to tasks. A known result for uniprocessors is that the scheduling algorithm Earliest Deadline First (EDF) is optimal [21]. Unfortunately, EDF is *not* optimal on multiprocessors either under the partitioned or the global approaches [22] , called respectively P-EDF and G-EDF. Another class of scheduling algorithms, which differs from the previous ones, gathers the *Pfair* algorithms (namely PD and PD^2) [23]. These are based on the idea of *proportionate fairness* and ensure that each task is executed with uniform rate.

Tasks are broken into quantum-length subtasks and time is subdivided into a sequence of subintervals of equal lengths called *windows*. A subtask must execute within the associated window and migration is allowed for each subtask. With respect to feasibility, the authors in [23] proved that a periodic task set with $r_i = 0$ has a Pfair schedule on m processors iff:

$$\sum_{\tau_i \in \tau} \frac{C_i}{P_i} \leq m \quad (1)$$

In order to make our experimental evaluation, as complete as possible, we select one algorithm in each class of scheduling (*i.e.* P-EDF, G-EDF and PD^2). Although the PD^2 algorithm is used to schedule hard real-time tasks on multiprocessors, we choose to include it in our study so as to cover all kinds of real-time applications.

3.2 Real-Time Transactions

Like real-time tasks, real-time transactions are classified according to the criticity of their deadlines: *hard*, *soft* or *firm*. The hard[3] class is rarely considered. Most studies assume the scheduling of transactions either in soft[4] or firm[5]classes.

Scheduling of transactions. The scheduler of transactions in database systems embeds a *concurrency control* protocol, which is in charge of resolving the conflicts between transactions when they occur, in order to maintain database consistency. In real-time database systems, not only database consistency should be satisfied, but transactions must also meet their deadlines [24]. To our knowledge, no real-time concurrency control policies are specially designed for software transactional memories.

3.3 Fraser's STM

FSTM [25] is a dynamic lock-free object based STM. It has been implemented as a C library. FSTM employs a recursive helping and an enforced global total order for transactions to ensure that despite contention, at least one process is making progress. The object is the basic unit of concurrency. Each object is pointed by an *object header* which contains the current version of the object (see Fig. 1.). The object header is pointed by an object handle which keeps the old and new references to the object. In case of a successful commit, the object header is updated with the new data block object. The transaction descriptor embodies both read-only and read-write lists. When a transaction accesses an object, the procedure is similar for both read-only and read-write accesses. The data structures described above are thus created according to the type of access. A *shadow copy* of the object is also created in the case of a read-write access and remains private until the transaction commits.

[3]System cannot tolerate the missing of deadlines.
[4]The transaction could be accepted even if it misses its deadline.
[5]Missing the deadline causes to abort the transaction.

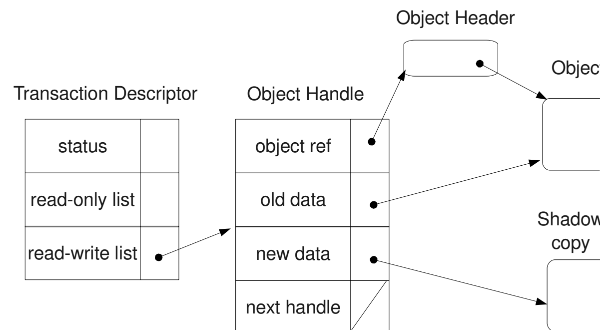


Fig. 1. Fraser's STM data structures

The commit phase is divided into three phases. The first phase is the acquire phase. The transaction attempts to acquire ownership of all objects on its read-write list in a canonical order. The transaction that attempts to acquire ownership of the object, performs a CAS (Compare And Swap) operation on the object header, to replace the pointer to the object by a pointer to its transaction descriptor. If the content of the object header points to a more recent object, the transaction will then abort. However, if the object is owned by another transaction then the obstruction is helped to completion. The second phase is the read phase. It checks whether each read-only object has not been updated since it was opened. If all objects are successfully acquired or checked then the transaction attempts to commit successfully. In the last phase, all acquired objects are released and if the transaction commits then each old object is replaced by its corresponding shadow copy (*i.e.* the new object).

4 Introducing Real-Time into STM

We aim to implement a real-time STM with soft constraints by minimizing the execution time jitter of transactions. In order to make STM suitable for soft real-time, not only the rollback times should be taken into consideration, but also both the scheduler of transactions and that of the operating system. Therefore, we propose to analyze which parts of STM cause execution time variations. Static memory approaches as proposed in the first implementation of STM [2] could be a good candidate to bound the execution time of transactions, but only basic real-time applications are involved in this case. It therefore contradicts the transactional memory concept which is rather intented for complex applications. In our study, we are interested in taking into consideration the dynamic allocation of memory since most of the recent STM implementations integrate a garbage collector. However, the dynamic allocation of memory in real-time context, is usually avoided because considered as an unbounded part. To summarize, we have to face orthog-

onal constraints while considering complex real-time applications using dynamic memory-based STMs.
As a solution, we choose to enhance Fraser's STM because its scheduler is based on the recursive helping between transactions. The helping mechanism appears more suitable for soft real-time. Indeed, a transaction with a low priority can help a transaction with higher priority and then at least one transaction will make progress. Moreover, FSTM dynamically creates and deletes objects in memory. Other implementations of STM like DSTM [19] are not considered here. Indeed, DSTM is an obstruction-free based implementation which provides the weakest guarantee to make progress. Consequently, it is not suitable for real-time systems.

4.1 Implementation

Intuitively, the underlying operating system (OS) has to be considered since transactions are executed within threads. That is why we use a real-time operating system (RTOS) named LITMUS^{RT} [6] [26]. Designed to run on top of a symmetric multiprocessor (SMP) architecture, it implements all the real-time task scheduling algorithms described in Section 3.1. LITMUS^{RT} is based on the Linux operating system (kernel version 2.6.24). The proposed schedulers are implemented as plugin components that can be selected from Linux user-space. In order to manipulate both tasks and synchronization mechanisms from Linux user-space, system calls are gathered within a C library. For all these reasons, LITMUS^{RT} becomes an excellent (perhaps the only) candidate to study the behavior of FSTM on multiprocessor systems, under a panel of advanced real-time scheduling policies.

We use the TLSF (Two-Level Segregate Fit)[7] [27] memory allocator to show the impact of object's allocation within our WCET analysis. TLSF is based on an algorithm that has a constant cost $\Theta(1)$. It solves then the problem of the worst case bound, thus maintaining the efficiency of the allocation and deallocation operations. Therefore, TLSF allows the reasonable use of dynamic memory in real-time applications.

4.1.1 Integration into LITMUS^{RT} library

Under LITMUS^{RT}, a real-time task is initially created as a standard linux thread (using the standard *pthread* library) before being effectively started. Then, it initializes the real-time environment and specifies the real-time parameters of the task, namely C_i and P_i. Thereby, the thread sporadically releases its jobs by calling the job function every P_i units of time.
To summarize, FSTM and the LITMUS^{RT} library have been combined by creating real-time threads within FSTM. We performed this integration so as to support both non real-time threads and real-time tasks.

[6] http://www.cs.unc.edu/∼anderson/litmus-rt
[7] http://rtportal.upv.es/rtmalloc/

4.1.2 Integration of TLSF library

TLSF is a C library. We integrated it into FSTM by replacing all the allocation and deallocation functions by those provided by TLSF. The memory pool which is used by TLSF is created at initialization time by the classical *malloc* function. Note that the TLSF's initialization is done before the creation of real-time threads.

5 Experimental Evaluation

5.1 Evaluation Context

We present here the experiments we performed to evaluate FSTM with respect to WCETs. Firstly, we describe the hardware and software configurations we use for our experimental evaluation, as well as the STM benchmarks we consider. Secondly, we report comparative results allowing us to select the best scheduling policy among Linux and LITMUS^{RT} operating systems. Finally, we study the dynamic memory allocator impact on FSTM.

Hardware context. The hardware platform used in our experiments is a two 32-bit multicore Intel Core(TM)2 Duo T7500 processors running at 2.20GHz with 4MB L2 cache, and 3.5GB of main memory. During all experiments, the multicore option has been enabled, and the cpu frequency for each core has been fixed at 2194MHz.

Software context. We have compiled the LITMUS^{RT} kernel for the above hardware platform and used it on top of an Ubuntu 8.04 hardy Linux distribution. The system has never been overloaded during the experiments neither under Linux (*i.e* only the test application has been launched), nor under LITMUS^{RT}.

Real-time task parameters. For each real-time task, we fixed $C_i = 20ms$ and $m = 2$; the parameter P_i being determined according to Equation 1. Thus, in all cases, we consider processors under heavy loads. The impact of the variation of these parameters is not considered in this paper, and we defer its consideration for future work.

STM benchmark. The experiments performed by Fraser [25] for the performance evaluation of STMs are about 10 seconds of duration. Fraser considers that this duration is pretty sufficient to stabilize the data into the cache, since after 10 seconds the same values are repeated. During the 10s of test, the evaluated STM performs a series of three operations: readings, writings and deletes over the shared objects organized as red-black trees or skip lists. The proportion of each operation performed is given as an input parameter of the benchmark. Fraser thinks that 75% of reads and 25% of writes and deletes well reflect a real situation.
For our experiments we used only red-black trees. Each experimental test lasts 10 seconds and operations are composed of 75% of reads namely lookup and 25% of writes

and deletes namely update and delete respectively. Shared resources are highly contended, with 2^4 maximum keys for red-black trees. Note that we have slightly modified this benchmark in order both to adapt it to the real-time context and to make our measurements.
Unlike classical STMs in which performance evaluation usually uses the average number of transactions per success and per time duration, we use other parameters for our real-time evaluation. These are described below.

5.2 Performance Metrics

Transaction WCET jitter. We measure the execution time of the three operations usually performed by a transaction (i.e. lookup, update and remove). The transaction WCET for each operation corresponds to a mean value and is obtained over all launched threads for a test of 10 seconds duration. This test is repeated 10 times. The jitter is then the variation of the WCET observed between each test. To perform these measurements, we recover the current processor ticks by calling the assembly instruction *rdtsc*. Each operation time is obtained by subtracting the processor ticks value at the end of operation to that at its start time. However, this method to get the ticks value at user-level does not work. Indeed, if a transaction starts on one core and migrates to another core, then the execution of the transaction becomes invalid since the clockticks of the cores are not synchronized.
We have proposed an alternative solution (see Algorithm 1) which consists in adding the core identity to the context of the thread. This is done by calling the assembly instruction *cpuid*[8]. Secondly, we make sure that the CPUID is corresponding to the *rdtsc* (see line 6) as the instructions are not atomically executed.
If task migration occurs more than 2 times during the test then we stop the retries (line 7). According to the state in which we perform the test, either we abort the program at start time of transaction operation (line 9) or consider the test as a bad one at the end of operation (line 11). At the end of the experiment, if the number of transactions that have experimented bad test is up to 1% of the total number transactions, then the experiment is manually restarted.
Note that we have measured the time duration of Algorithm 1. which is 0.5μs. Thus, the worst case execution path of this algorithm is 2μs (*i.e*, 2×0.5 at the starting time of the transaction operation, plus 2×0.5 at the end of the operation). Therefore, the WCET has a precision within the interval $[1, 2]\mu$s.

Time variation factor. As the experiment that gives the WCET of transactions is repeated 10 times, we obtain 10 values of WCET for each one of the three operations of the transactions. For each operation value, we compute its mean $\overline{x}$ and its standard deviation σ. Let the time variation factor $V = \frac{\overline{x}}{\sigma}$. The variation facor V is then a ratio which provides information on the variation degree of the WCET of transactions over 10 experiments.

Rollback time ratio. This parameter is measured once and the experiment is not repeated (*i.e* 10 seconds of duration only). We define for each thread, the rollback-time ratio $roll_i$ of its transactions. For each operation O_i of the transaction, the parameter $roll_i$ is defined as follows: $roll_i = \frac{\sum^n RollbackTime(O_i)}{\sum^n Duration(O_i)}$. The global rollback-time R we consider for our experiments is then:

$$R = \frac{\sum^N roll_i}{N} \tag{2}$$

where N is the number of threads.

[8]The id assigned by the APIC is at the 25-bit in our case

Algorithm 1 Transaction operation measurement

1: $init\ RetryCPU \Leftarrow 2$
2: $T_j.coreID \Leftarrow CPUID()$
3: **repeat**
4: $\quad RetryCPU \Leftarrow RetryCPU - 1$
5: $\quad T_j.RTSched_j.r_j \Leftarrow ReadProcessorTicks()$
6: **until** $T_j.coreID = CPUID()$ **Or** $RetryCPU = 0$
7: **if** $RetryCPU = 0$ **then**
8: $\quad$ **if** $state = TransactionStarting$ **then**
9: $\quad\quad Abort()$
10: $\quad$ **else**
11: $\quad\quad BadTest \Leftarrow BadTest + 1$
12: $\quad$ **end if**
13: **end if**

5.3 Results

5.3.1 OS's impact

In this experiment, we intent to show how the underlying operating system can impact on the rollback times of transactions. Results are presented in Figures 2, 3 and 4. Note that the average number of transactions is around of 7×10^6 for each case.

We can see that the parameter R is constant and less than 0.25% for the three policies, namely Linux, G-EDF, and PD^2. This value can be practically ignored since in each policy it still remains constant for an increasing number of threads.
We observe that for the Pfair policy (see Fig. 3), R is reduced. This is because Pfair does not scale due to its important migration cost. Indeed, the migration cost increases the effective duration time of the thread and thereby that of transactions. Transaction rollbacks rarely occur and then are less likely to be impacted by the migrations. In fact, the values used for computing R − not presented here for readability − show that the rollback time is not reduced, but only the duration of transactions is increased. The same phenomenon can be slightly observed with G-EDF (see Fig. 4) since the G-EDF ratio of migrations is usually lesser than that of Pfair.
On the contrary, Fig. 5 shows that R is almost null. In this case, the duration of transactions is relatively reduced thanks to the minimal preemption and overheads induced by P-EDF. These overheads, mainly caused

by task migrations, are avoided under P-EDF. More important, the preemption time per task is reduced since a task is only preempted by those of its own queue. Thus minimizing rollback times.
Therefore, this experiment shows that under FSTM, rollback times do not make up the major part of the transaction duration. In addition, according to their weak impact, rollback times can be ignored when doing the WCET analysis for soft real-time constraints. Furthermore, for the reasons mentioned above, we choose the P-EDF policy for the rest of our experiments.

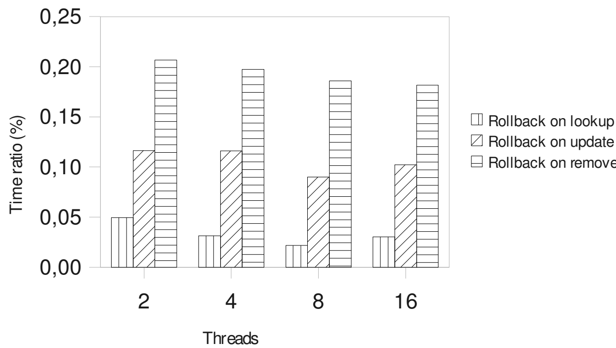


Fig. 2. Rollbacks under Linux

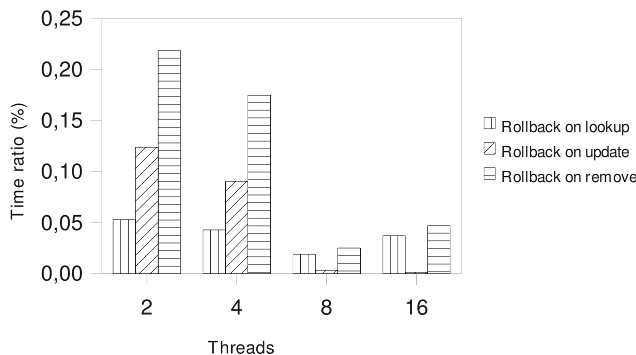


Fig. 3. Rollbacks under PD^2

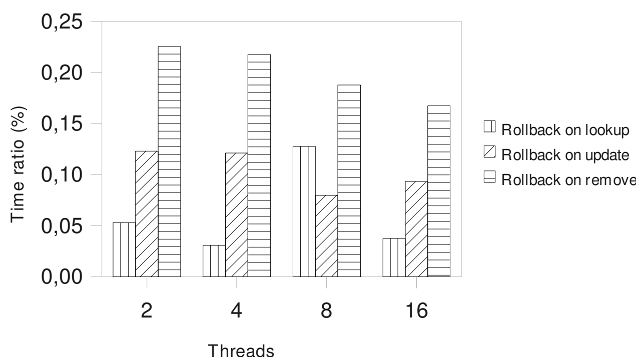


Fig. 4. Rollbacks under G-EDF

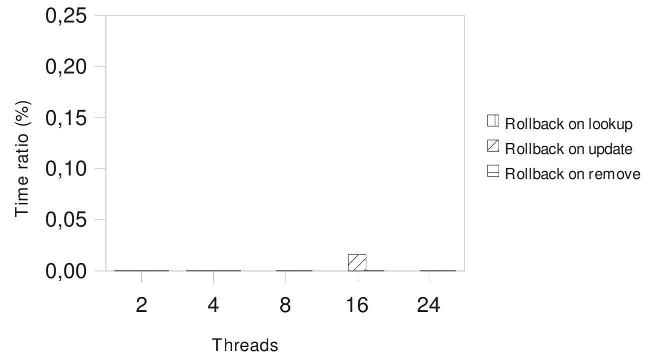


Fig. 5. Rollbacks under P-EDF

5.3.2 Dynamic memory allocator's impact

Since rollbacks do not impact significantly on the duration of transactions, we attempt here to show which part has really a detrimental effect, considering the P-EDF policy (selected from the previous experiment). We compare the results obtained using the classical memory allocator *malloc* with that of TLSF, on the basis of the V parameter defined above.

Fig. 6. shows that the duration of transactions has an important jitter for the three operations. Although P-EDF is used, FSTM suffers from important time latencies that characterize the execution environment at each program launch. FSTM uses a garbage collector that we have configured to be in minimal mode. Indeed, the normal mode often causes the program to abort due to a chunk imposed not only to deaden the cost allocation but also to increase the per-cacheline pointer density. However, we noticed that this mode of garbage collector configuration impacts on the total memory used by FSTM, but not on the V parameter.
The real reason of this variation is demonstrated on Fig. 7 and Fig. 8. When TLSF is used instead of the classical memory allocator *malloc*, the WCET of transactions is bounded with almost the same value. Indeed, the maximum variation that is reached using *malloc* is around 160% versus 8% when using TLSF.
This shows that FSTM could satisfy soft real-time constraints provided a bounded memory access is performed (i.e. using a constant-time dynamic memory allocator like TLSF).

6 Conclusion

We believe that the advantages of transactional memory can also be brought to real-time systems. Thus, we studied the possibility of introducing soft real-time into STMs by analyzing the WCET of transactions. To our knowledge, such study has not been attempted before.
The main results of our study are summarized hereafter: **(i)** P-EDF reduces the rollback times of transactions; **(ii)** For soft real-time constraints, the rollback times could be ignored within FSTM when doing the WCET analysis;

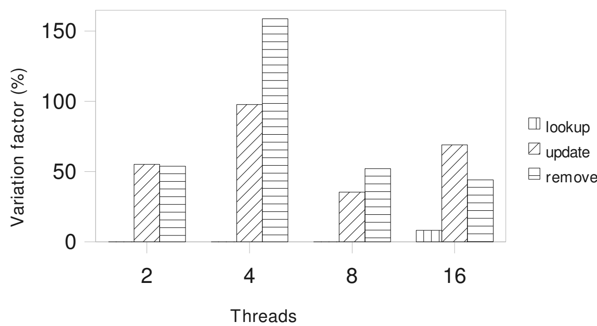


Fig. 6. WCET jitter using classical *malloc* (P-EDF)

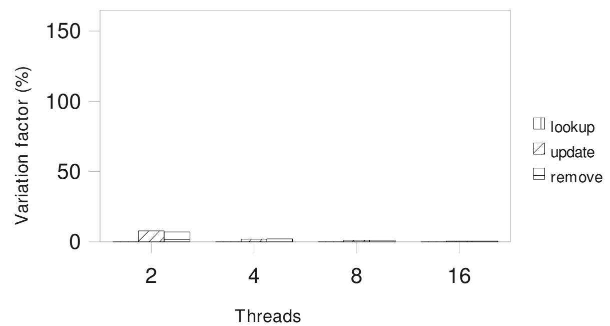


Fig. 7. WCET jitter using TLSF (P-EDF)

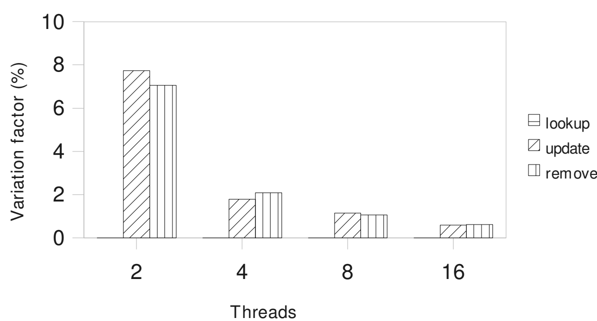


Fig. 8. Zoom on Fig. 7

(iii) FSTM could greatly satisfy soft real-time constraints provided memory accesses are bounded.

Now that we have bounded time in STM, many directions are then possible for future work. First, in this study we only dealt with the duration of transactions. It would be interesting to study the impact of STM on the number of deadline violations when scheduling real-time transactions. Secondly, in our experiments, we arbitrarily fixed the parameters of the real-time tasks. It would be also interesting to evaluate the impact of the processor load. Finally, we would like to formalize the interaction between the real-time scheduler of tasks and that of transactions.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *proc. the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[2] N. Shavit and D. Touitou, "Software transactional memory," in *proc. the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1995, pp. 204–213.

[3] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc r processor," *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory." in *HPCA*. IEEE Computer Society, 2005, pp. 316–327.

[5] R. Ennals, "Softawre transactional memory should not be obstruction-free," Intel Research Cambridge, Tech. Rep., 2006.

[6] K. Fraser and T. Harris, "Concurrent programming without locks." *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007.

[7] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime." in *PPOPP*, J. Torrellas and S. Chatterjee, Eds. ACM, 2006, pp. 187–197.

[8] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory." in *PPOPP*, J. Torrellas and S. Chatterjee, Eds. ACM, 2006, pp. 209–220.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory." in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 336–346.

[10] W. N. Scherer III and M. L. Scott, "Contention management in dynamic software transactional memory," in *proc. the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[11] W. N. S. III and M. L. Scott, "Advanced contention management for dynamic software transactional memory." in *PODC*, M. K. Aguilera and J. Aspnes, Eds. ACM, 2005, pp. 240–248.

[12] M. Schoeberl, B. Thomsen, and L. L. Tomsen, "Towards transactional memory for real-time systems,"

Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Research Report 19/2009, 2009.

[13] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2008, pp. 342–353.

[14] J. H. Anderson, R. Jain, and S. Ramamurthy, "Implementing hard real-time transactions on multiprocessors," in *RTDB*, 1997, pp. 247–260.

[15] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 221–228.

[16] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii." in *DISC*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 194–208.

[17] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott, "Conflict detection and validation strategies for software transactional memory." in *DISC*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 179–193.

[18] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems." in *SPAA*, F. M. auf der Heide and N. Shavit, Eds. ACM, 2008, pp. 169–178.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III, "Software transactional memory for dynamic-sized data structures." in *PODC*, 2003, pp. 92–101.

[20] D. Johnson, "Fast algorithms for bin packing," *Journal of Computer ans Systems Science*, vol. 8, no. 3, pp. 272–314, 1974.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[22] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[23] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[24] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," in *VLDB*, 1988, pp. 1–12.

[25] K. Fraser, "Practical lock freedom," Ph.D. dissertation, Cambridge University Computer Laboratory, 2003, also available as Technical Report UCAM-CL-TR-579.

[26] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmus rt : A testbed for empirically comparing real-time multiprocessor schedulers." in *RTSS*. IEEE Computer Society, 2006, pp. 111–126.

[27] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: A new dynamic memory allocator for real-time systems," in *ECRTS*, 2004, pp. 79–86.

Contract based management of the memory resource

Ismael Ripoll†, Patricia Balbastre†, Miguel Masmano†, Alfons Crespo†, and Alan Burns‡*
† Instituto de Informática Industrial, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia, Spain
‡ Deparment of Computer Science
University of York, Heslington, York, YO10 5DD, UK

Abstract

Resource reservation has been widely used in many real-time systems to guarantee the proper access to the system resources. Despite that being the memory a key resource, it has attracted little attention in the specific area of real-time systems. In order to use dynamic memory in real-time systems, two fundamental problems have to be settled: allocation and deallocation in bounded time, and the fragmentation problem.

Recent research results have removed the unbounded timing behaviour of the dynamic memory allocation. TLSF is a fast and constant time memory allocator. Although the fragmentation is still an open research problem, we present a deep and comparative analysis showing that it has several characteristics in common with the well-known WCET analysis.

In this paper, we present a contract based framework for handling dynamic memory in real-time systems. The framework provides both: i) timing guarantee for dynamic memory allocation and deallocation operations, and ii) spatial guarantee by using a flexible contract negotiation model.

1 Introduction

Nowadays, computers are included as components in many kinds of systems. We can find them in automotive, aerospace, robotics, home systems, toys, etc. Most of these embedded systems need to interact with the environment and operate under real-time constraints. Moreover, the wide field of applications requires an important engineering and programming effort to add flexibility and adaptability to these systems. To fulfil these requirements the complexity of embedded software increases.

A basic property that differentiates embedded systems from other general software systems is their execution environment under resource constraints. These resource-constrained applications are related to CPU time, memory usage, I/O, network bandwidth, energy, space, etc.

Resource-aware computing tackles this challenge by managing them dynamically, and thus, optimising the use of these resources. Resource managers are the software components responsible for performing this task. In real-time systems, some resource-management related topics such as CPU time deliverance and the relationship between CPU and power consumption have been deeply studied providing consolidated theoretical foundations. However, other topics such as memory management have been completely ignored. Like CPU, memory is a fundamental part of the embedded system, nevertheless, real-time systems allocates statically the existing memory at load time, most of the times, overestimating the real necessities in memory of the applications. However, actual applications seldom behaves statically but they are usually composed by a set of static regions (code and data) which do not vary during the program's execution and dynamic ones (stack and heap) which grow/shrink according to application needs. The stack can be considered as a limited region not being particularly affected by a static deliverance of the memory, however, the absence of a heap prevents applications to use dynamic structures. As a direct consequence of this, embedded systems designs tend to require more memory than they really would need if this resource was dynamically allocated.

Moreover, the problem of resource management is that different resources cannot be handled independently. Currently real-time operating systems are including some sort of resource manager. This is an emerging trend, and more research on techniques and mechanisms to use and control the system resources (application isolation) are needed.

This work is motivated by the necessity of including memory resource management in a real-time resource management framework (FRESCOR). Dynamic storage allocation is considered one of the keys to add flexibility and adaptability to the application programming. For this rea-

*This work was supported by FRESCOR and the Spanish Government Research Office (CICYT) under grant THREAD (TIC2005-08665-C03)

son, it becomes increasingly desirable. However, dynamic memory has seldom been used in real-time applications due to its unbounded nature.

Recently, a new real-time allocator (TLSF, Two-Level Segregate Fit)[16, 14] was proposed specifically designed to meet real-time requirements. This is the first allocator able to perform allocation/deallocation in constant-time, O(1), with a very high efficiency in term of time and space (fragmentation). The use of this allocator opens new options to consider dynamic storage allocation in real-time applications due to its bounded response time.

Although, the proposed model is general and could cover the memory of the programs in execution, we have only considered the memory usage related to dynamic memory. Static memory as a unique resource is a well-known bin-packing problem. When memory is not static or more than one resource is considered, the problem is not so trivial. Static memory jointly with CPU allocation has been considered in [7]. It proposes a way to analyse, in a multiprocessor platform with limited resources, the computing capacity at each processor and the amount of local memory for a set of tasks.

The approach presented in this paper is being developed in the FRESCOR project [8] where other resources such as CPU and network are being developed by other research groups which will be integrated at the last stages of the project.

1.1 Contributions and outline

In this paper we propose a memory resource reservation model for flexible embedded systems requiring dynamic memory. The static memory to execute the applications (program code, static data and stack) is not considered in this work.

The main contributions of this paper are:

- A vision of the memory as a resource in the same way that other resources are considered.
- A memory model and a memory reservation architecture being able to manage the spare memory of the system.
- An acceptance test for memory contracts and a memory reclaiming mechanism of the memory associated requests.

As far as the authors know, this is the first work dealing with quality of service related to memory management of the application.

This paper is organised as follows: section 2 presents the most common misconceptions about dynamic memory in real-time systems. In section 3 the parallelism between CPU and memory management is established. In section 4, the memory model is presented. Section 5 characterises the states of the memory and proposes an acceptance test for dynamic memory requests. The proposed acceptance test is evaluated in section 6. Section 7 summarises the existing works on resource reservation and dynamic memory in real-time systems. Finally, section 8 presents a summary of the results of this work and outlines future work on this topic.

2 Dynamic memory misconceptions

2.1 Bounded time allocation

Before the TLSF allocator [**?**] was presented, the use of dynamic storage allocation (DSA) was avoided in hard real-time applications. In [18] and [6], authors provide arguments to avoid its use due to the unbounded operations.

The only allocator with a bounded operation cost used in real-time applications was the well known Binary-buddy (or any other of the "buddy" family allocators). Being H the size of the heap, this allocator has an $O(log_2(H))$ cost on both, allocation and deallocation, but causes a large internal fragmentation (close to 50%).

Another generally accepted idea [4] is that there is a trade-off between space and time efficient allocators. In other words, a fast and bounded allocation can only be achieved at the cost of increased wasted memory due to "fragmentation".

These misconceptions were analysed by Johnstone et al. [11], and concluded that **it is possible to design space efficient allocators using well known allocation policies** (best-fit or good-fit), and those policies can be implemented using fast mechanism (segregated lists).

The publication of the TLSF allocator changed some of the assumptions taken from granted about how a DSA works. The TLSF memory allocator implements all dynamic memory operations (malloc, free, realloc) in constant time regarding the stated of the pool. The performance of the TLSF is not affected by the fragmentation or by the amount of different free blocks sizes. Also, the average time of the TLSF is close to the fastest allocators (DLmalloc[12]).

2.2 Memory fragmentation

P. Wilson et al. [23] define the fragmentation as *the inability to reuse memory that is free*. This definition focuses only of the final result of the allocation and deallocation process. It is also interesting to note that fragmentation depends both, on the current memory state, and on the future application requests.

The fragmentation problem was largely studied since the very beginning of the computer science. But, in many cases

the results of a researcher fell in contradiction with the previous ones. Zorn et al. [24] discovered a very important fact when defining a standard set of synthetic workload for evaluating DSAs. They tried to define a reference set of synthetic models which reproduce the allocation/deallocation sequence behaviour of a group of selected applications. They worked with both, models that were used by other researchers, and original models. Even the most accurate and complex mathematical model they were able to roughly approximate the behaviour of real program. The main conclusion was that most of the previous results should be reconsidered or even invalidated.

M.S. Johnstone and P.R. Wilson[11] analysed real applications and current policies and concluded that the fragmentation “problem” is really a problem of poor allocator *implementations*, and that for these programs well-known policies suffer from almost no true fragmentation. In addition, very good implementations of the best policies are already known.

We agree with the ideas of M.S. Johnstone and P.R. Wilson [11] about the nature and real impact of the fragmentation. Therefore, **the fragmentation problem can be bounded in most real applications, which effectively enables the use of dynamic memory**. In any case, further research should be carried out to fully understand and seize the problem, using formal and analytical methods rather than simulations and practical experiments.

The TLSF allocator follows all the policies shown by M.S. Johnstone et al. which reduce fragmentation (immediate coalescing, good fit and reuse blocks which has been releases recently). Those policies were implemented using a clever set of segregated list ranges which both, can be implemented using O(1) algorithms and causes only a 3% worst case internal fragmentation.

3 Processor versus memory

Fragmentation vs. execution time: The memory fragmentation problem has many similarities with the WCET analysis. In both cases, the analytical estimation of the worst case value is hard to find and quite pessimistic.

Currently, the worst case fragmentation (WCF) is only known for small set of allocation policies [20]: best-fit and first-fit. As far as the authors know, there are little WCF analysis about the family of allocation policies known as good-fit, which are ones used on the allocators that also show constant time operation (TLSF and Half-fit). Both, the conclusions about the practical fragmentation of Johnstone at al. [11], and the fact that external fragmentation can be reduced by increasing internal one [19][1], make us optimistic about the possibility to see advances in the WCF for real-time systems.

Physically allocatable resources: In a conventional system, a process can be preempted and suspended for a long time. When later resumed, the final logical result will still be correct. Also, the processor can be run as long as required to solve a given problem (tasks do not have deadlines). In a conventional system, the processor is a very flexible and dynamic resource.

One of the main differences between conventional and real-time systems is that the processor is managed as a statically allocated resource. The real-time scheduler is responsible of allocating to each task the amount of resources, $\frac{C_i}{P_i}$, granted to it. On a periodic system, a fraction of the total processor time is allocated to each task. The periodic nature of most real-time systems, jointly with the scheduler, transforms the processor into a bounded resource: the processor cannot be over 100% utilisation (or less depending on the scheduling policy).

More similarities can be found when comparing the memory management with the CPU reservation based servers. For instance, a Periodic Server is invoked with a fixed period to spend a server’s capacity. This capacity is consumed by the tasks ready to use or, if there are not tasks ready, it is idled away. So, the capacity is preserved.

The memory manager has a capacity (maximum amount of memory) that it can provide to its associated tasks. This capacity persists along its execution and never is replenished (renewed). From this point of view, it is a *non-renewable resource*. The application, or the associated tasks, have the responsablity of usign the resource in the proper way to allocate memory blocks and freeing them.

Feasibility analysis: Contrarily to the processor feasibility analysis, the analysis of the total amount of memory required to run an application is quite straightforward. Considering that the application does not use dynamic memory, the total application required memory can be obtained from the compiler and estimating the amount of stack memory[2] used. These two tests, for processor and memory, are done off-line during the development phase.

Spare capacity: In most cases, the system has more resources than those strictly needed by the application: the processor is not fully utilised and there are free memory. A lot of research has been done trying to take advantage of the processor spare capacity by using aperiodic and bandwidth servers. The basic idea is to use the extra capacity to improve the quality of the system response but without jeop-

[1]For example, an extreme case occurs when the allocator rounds-up all block requests to a one single size. In this case, no external fragmentation happens.

[2]Which basically depends on the number of nested function calls and the local variables.

ardising the correct execution of the hard real-time tasks. A server is an abstract entity used by the scheduler to reserve a fraction of CPU-time to a task.

4 Memory model

This section details the memory model and the underlying architecture which supports the memory management.

4.1 Memory resource manager

The dynamic memory pool is managed by the memory resource manager (MRM) that is the layer in charge of negotiating the memory reservation requests (*contracts*) between applications and the system. The contract has to be negotiated between both, application and MRM, to guarantee the availability of the resources. As a consequence of a successful negotiation, the MRM creates a memory resource controller (R)[3] that will monitor and control the use of the resource. The application, as owner of the resource, binds one or more tasks to it. An application can negotiate several contracts with the MRM. Also, a task can be bounded to several granted resources.

The MRM is defined as a set of resources $\Upsilon = \{R^1, R^2, \cdots, R^n\}$, and a memory pool $\Omega = (M_{tot}, M_{FR}, M_{CR})$.

4.2 Memory resources

A memory resource controller (R^k) is the component that monitors and controls the use of the granted memory. Each R is characterised by the following parameters:

$$R^k = (R^k_{min}, R^k_{max}, R^k_{imp}, R^k_{stab}, R^k_{at}, R^k_B, R^k_{clive}, R^k_{mlive})$$

where R^k_{min} R^k_{max} are the minimum and maximum amount of memory that the task needs to operate properly. The value of the granted budget will be in the range of these values; R^k_{imp} is the absolute fixed importance; R^k_{stab} is the duration relative to the time at which the request is made, during which the memory assigned to the application must not change, R^k_{at} specifies the arrival time of the negotiated and accepted contract, R^k_B is the budget of memory granted in the negotiation; R^k_{clive} is the *live memory* as the sum of all the currently allocated memory blocks to this resource and R^k_{mlive} is the maximum value reached by the current live memory.

The first four parameters are provided in the contract negotiation, the rest are required by the resource controller to monitor and control the resource usage.

[3]It corresponds to a *virtual resource* controller but we avoid the use of the term *virtual* because of virtual memory has a well-known meaning in memory management

In case that an application does not require dynamic memory, the only parameter needed by the MRM is R_{min} which corresponds to the static amount of memory needed by the application. As a consequence, $R^k_B = R^k_{min}$, and the rest of parameters are not required. Note that the static memory model is a subset of our proposal.

The R_{max} parameter can be estimated from the dynamic data needed by the application. If the dynamic memory is stored in a buffer until a second thread remove it, it can be considered that the minimal application requirements are achieved with a minimum buffer size, but the larger buffer size the better results are obtained. The R_{max} (R_{min}) parameter will be the sum of the static memory and the maximum (minimum) buffer size.

A resource R^i is said to be *eligible* at time t_{now} if its stability time has expired: $R^i_{at} + R^i_{stab} \leq t_{now}$.

Let Υ^k_- be the subset of elegible resources with lower importance than R^k:

$$\Upsilon^k_- = \{R^i \ / \ (R^i_{imp} < R^k_{imp}) \wedge (R^i_{at} + R^i_{stab} \leq t_{now})\}$$

Similarly, Υ^k_+ is the subset of elegible resources with higher importance than R^k:

$$\Upsilon^k_+ = \{R^i \ / \ (R^i_{imp} > R^k_{imp}) \wedge (R^i_{at} + R^i_{stab} \leq t_{now})\}$$

4.3 Memory pool characterisation

The memory pool is characterised by three parameters $\Omega = (M_{tot}, M_{FR}, M_{CR})$. M_{tot} is the total amount of "heap" system memory (memory used to attend dynamic memory requests). All the dynamic memory used by applications will be managed using a single memory pool. In other words, when a resource is created, no memory is reserved (or allocated) from the heap to create a new subheap. The resource operates as a proxy to the single system pool, implements the access policy and enforces strict resource isolation, by accounting how much memory was granted initially, and how much memory can still be requested to it. M_{tot} is not entirely available for applications. Some amount of this memory will be reserved for different purposes. Specifically:

$$M_{tot} = M_u + M_{FR}$$

where:

M_u is the effective memory guaranteed to be used by the dynamic storage allocation.

M_{FR} is the memory reserved to deal with fragmentation. This parameter mainly depends on two factors: the efficiency of the dynamic memory allocator and the behaviour of the application (mutator[4]). As explained

[4]The term "*mutator*" is used in the area of dynamic memory to refer to the application that uses (allocates and frees) memory.

in section 2.2, there are some allocating policies that are known to produce less fragmentation; the TLSF allocator has been shown to be among the most efficient allocators. Experimental results [16, 14] show that no real application has caused more than 15% of total fragmentation (both internal and external). As a rule of thumb, we suggest to reserve $FR = 20\%$ of the memory, so $M_{FR} = \frac{M_u \cdot FR}{100}$. In any case, it is recommended to analyse the exact memory needs of the applications and adjust this parameter accordingly.

A fraction of the M_u can be reserved to be used when the system is under a high demand of memory, to attend new incoming contract negotiations. Let CR be the percentage of memory reserved for run time contract negotiation. For convenience, let $M_{CR} = \frac{M_u \cdot CR}{100}$.

At any time, the remaining memory M_r that can be granted to incoming contracts can be calculated as:

$$M_r = M_u - \sum_{1 \le i \le n} R_B^i - M_{CR}$$

Negative values of M_r mean that the system is using memory from M_{CR} and, consequently, it is in a stressed situation. Figure 1 shows graphically how the memory pool is managed.

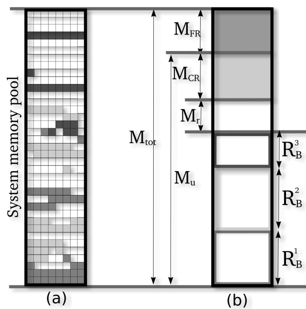


Figure 1. Memory pool parameters.

Figure 1.*a)* represents the memory pool with the used and free blocks distributed all along the memory. Figure 1.*b)* draws the memory abstraction as it is offered by the MRM.

4.4 Memory access protocol

In order to use dynamic memory, the application has to follow a three-step protocol (sketched in Figure 2): negotiation, binding and usage.

In the first step, the application negotiates the use of the resource (contract). If it succeeds, the memory resource controller R^k is created and the application tasks can be associated to it. After the binding step, the tasks can use the memory allocating (malloc) and deallocating (free) operations. These operations are redirected to the memory allocator (TLSF) which receives the invocations of all memory resource controllers. TLSF uses a unique memory pool for all these memory allocations.

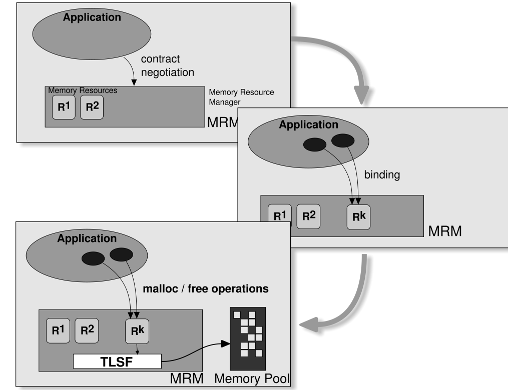


Figure 2. Memory access protocol

The goal of the contract negotiation is to determine the amount of memory granted to the resource, that is, its budget (R_B^k). The contract negotiation will succeed if $R_{min}^k \le R_B^k \le R_{max}^k$.

Three aspects have to be considered during the negotiation: i) the amount of memory to be assigned, ii) which R^i are eligible for reducing their budgets and iii) how much memory can be reclaimed from each one. Follows a formalisation of these ideas, which will be used in the acceptance test.

The amount of memory to be assigned will be the maximum requested when there is enough memory or it can be obtained from less important memory resources. If more important memory resources are reclaimed, then the *goal* is to satisfy the minimum memory requested.

The R^i candidates to reduce its budget are the eligible ones. Regarding the amount of memory reclaimed from each R^i, tree levels of aggressiveness reclaiming can be defined:

1. Memory that never has been used. It corresponds to the difference between the budget and the maximum live memory: $R_B^i - \max\{R_{min}^i, R_{mlive}^i\}$
2. Memory that is not currently being used. It corresponds to the difference between the budget and the current live memory: $R_B^i - \max\{R_{min}^i, R_{clive}^i\}$.
3. All the extra memory. The difference between the budget and its minimum: $R_B^i - R_{min}^i$.

Figure 3 shows an example of the dynamic memory evolution in a memory resource (R^k). It plots the amount of

memory allocated by R^k as result of dynamic memory allocations and deallocations. At the end of the stabilisation time, this resource is eligible for reclamation. At t_{now}, a new contract is under negotiation. At this time, the three reclamation levels are drawn in the figure.

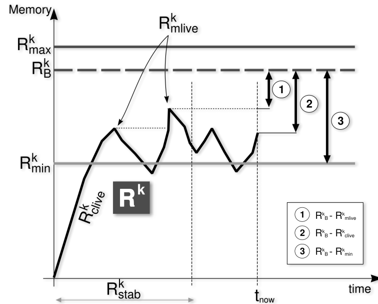


Figure 3. Dynamic memory used by R

If the third level is applied, then the current live memory may be larger than the new budget. There are several actions that can be done to overcome this transient overload situation:

Do nothing: But no allocation can be done (only deallocation) until the live memory will be below the budget. The system will be in an unsafe state until the live memory is reduced, which is not acceptable on a real-time system.

Inform the application: Ask to the application to deallocate memory. As the application has the knowledge about the semantics of the data, it can properly deallocate the most convenient blocks.

Deallocate memory: The MRM releases the last allocated blocks. This is not acceptable, because it breaks the basic rules of the program execution.

In this paper, only the two first levels will be considered for memory reclaiming.

5 Acceptance test

5.1 Characterisation of the memory states

In this section, we analyse the reachable states of the memory taking into account the contract requirements, the current memory status and the reclaiming capacity applied. Figure 4 shows different cases that can occur comparing the available and the requested memory.

In case 1 there is enough available memory to fulfil the request needs, so the memory requested is granted. Cases 2, 3 and 4 perform a memory reservation request that cannot

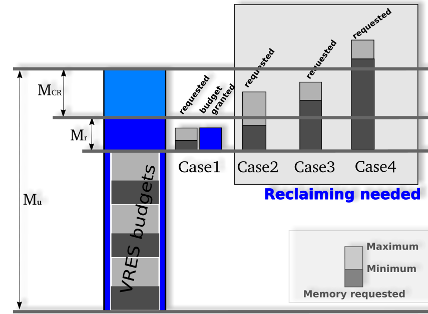


Figure 4. Negotiation with space memory attributes.

be granted directly because it involves not only the remaining memory but the reserved memory (M_{CR}) or exceeds the memory capacity. These cases require a reclaiming process on other memory resources in order to recover memory that is not used.

Figure 5 shows the three initial situations or states (cases 2, 3, and 4) and the final memory states reachable depending on the amount of memory reclaimed to the other resources.

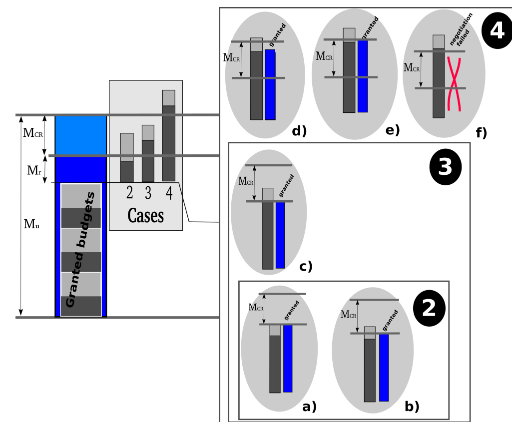


Figure 5. Final states after the reclamation phase for cases 2, 3 and 4

a) During the reclaiming phase from lower important resources, the memory available reaches the maximum requested ($M_r > R^k_{max}$). It is granted (Goal = R^k_{max}).

b) The reclaimed memory to lower or equal important resources is enough to grant the minimum ($M_r > R^k_{min}$) but not sufficient to grant the maximum ($M_r < R^k_{max}$). Before reducing to higher important resources, this amount is granted. (Goal = $[R^k_{min}, R^k_{max}]$)

c) The reclaimed memory is the minimum requested reducing memory from lower, equal and higher important

resources ($M_r = R^k_{min}$). (Goal = R^k_{min}).

d) After the reclaiming phase, it could not obtain enough memory ($M_r > R^k_{min} > M_r + M_{CR}$). To grant the minimum memory requested, it uses the reserved memory.

e) After the most aggressive reclaiming phase, the amount of memory obtained was the minimum using all the available (remaining + reserved).

f) After the reclaiming phase, it could not obtain enough memory ($M_r + M_{CR} < R^k_{min}$). In this case, the memory contract cannot be granted and the **negotiation fails**.

Case 2 can reach states *a)* and *b)*. Case 3 can additionally arrive to state *c)*. Finally, Case 4 can reach any of the states.

5.2 Acceptance algorithm

The reclaiming phase is in charge of recovering memory already assigned to an existing resource. It is easy to test if there is enough memory: if $R^k_{max} \leq M_r$, the negotiation succeeds and the granted budget is R^k_{max}. If not, it will reclaim memory from other resources, and the budget goal will be reduced as the MRM reclaims from more resources. The goal is that a resource cannot have its maximum memory if higher important resources do not have it. The reclaiming process consists of four reclaiming steps. Listing 1 sketches the actions done during the contract negotiation (the functions that carry out the reclamation are explained below).

Listing 1. Pseudo-code of the acceptance test

```
1   function acceptance_memory_test(R^k) is
2   begin
3      -- Phase 0: Use remaining memory.
4      --          Budget goal is : R^k_max
5      if (M_r >= R^k_max) then
6         return(GRANTED);
7      end if ;
8      -- Phase 1: Reclaim memory never used from lower
9      -- imp resources . Budget goal is : R^k_max
10     M_r += reclaim_memory_from_Υ^k_-(R^k);
11     if (M_r >= R^k_max) then
12        return(GRANTED);
13     elsif (M_r > R^k_min)
14       return(GRANTED);
15     end if ;
16     -- Phase 2: Reclaim memory never used from higher
17     -- imp resources. Budget goal is : R^k_min
18     M_r += reclaim_memory_from_Υ^k_+(R^k);
19     if (M_r >= R^k_min) then
20        return(GRANTED);
21     end if ;
22     -- Phase 3: Use reserved memory.
23     -- Budget goal is: R^k_min
24     M_r += M_CR;
25     if (M_r >= R^k_min) then
26        return(GRANTED);
27     end if ;
28     -- Phase 4: Reclaim memory not currently being
29     -- used from lower imp resources .
30     -- Budget goal is: R^k_min
31     M_r += reclaim_live_memory_from_Υ^k_-(R^k);
32     if (M_r >= R^k_min) then
33        return(GRANTED);
34     end if ;
35     return(FAILED);
36  end acceptance_memory_test;
```

This test performs an analysis of the memory state and determines whether the memory requested can be granted or not. The reclaiming phase is performed with different levels of aggressiveness or depth.

During the first two phases, the system is considered to be unstressed because 1) there is free memory not allocated to any resource or; 2) because the memory previously reserved to less important resources has been never used by those resources, therefore we assume that the application made an overestimation of the maximum memory, and that memory can be reclaimed with no damage.

If the algorithm cannot attend the new contract using the memory obtained from the first two phases, then we consider that the system is stressed, or it is close to. In this case, the amount of memory that will be given to the new resource R^k is not longer the maximum requested, but the minimum.

A particular case may occur if at the end of phase 1 there is not enough memory to attend the maximum memory requested, but there is more memory than the minimum requested. In this case, the new budget will be a value in the range $[R^k_{min}..R^k_{max}]$.

This algorithm ends when enough memory is found or after the fourth phase. If the algorithm ends successfully, then the variable M_r contains, at least, the minimum memory requested in the new contract. In this case, the new resource is created, and the remaining memory as well as the budget of all affected resources are updated accordingly. If the algorithm fails (not enough memory), the state of the resources remains unchanged. It is implemented as a transaction: system and resource data are copied in a scratchpad area, the acceptance test works on the scratchpad area, and then, only if the algorithm ends successfully the results are committed to the real system and resource data. Otherwise, the scratchpad data is discarded.

The next pseudo-code details the reclamation phases 1, 2 and 4:

```
1  procedure reclaim_memory_from_Υ^k_−(R^k) is
2      foreach R^i ∈ Υ^i_−
3          M_r+ = R^i_B − max(R^i_mlive, R^i_min);
4          exit when M_r ≥ R^k_max;
5      end for
6  end reclaim_memory_from_Υ^k_−;
```

```
1  procedure reclaim_memory_from_Υ^k_+(R^k) is
2      foreach R^i ∈ Υ^i_+
3          M_r+ = R^i_B − max(R^i_mlive, R^i_min);
4          exit when M_r ≥ R^k_min;
5      end for
6  end reclaim_memory_from_Υ^k_−;
```

```
1  procedure reclaim_live_memory_from_Υ^k_−(R^k) is
2      foreach R^i ∈ Υ^i_−
3          M_r+ = R^i_B − max(R^i_clive, R^i_min);
4          exit when M_r ≥ R^k_min;
5      end for
6  end reclaim_live_memory_from_Υ^k_−;
```

6 Evaluation

This section presents the evaluation of the memory resource controller. Next sections detail the evolution of two scenarios when different resources are defined. These scenarios are representatives of the results obtained with different set of resources. The scenarios presented in this section intend to show how the algoritm work and the evolution of the memory assigned to the different resources. It is not relevant the sizes showed in the scenario ($M_u = 6000$), that is relevant is the relation between the total amount of memory and the memory requested by the resources. Finally, an evaluation of different sets of scenarios is presented.

6.1 Scenario 1

This scenario shows the memory negotiated by five resources with usable memory of $M_u = 6000$ bytes and a memory reservation of the 15% (M_{CR}= 900). Each resource R^i has the following characteristics:

	R^k_{at}	R^k_{imp}	R^k_{min}	R^k_{max}	R^k_{stab}
R^1	10	1	1500	1800	400
R^2	10	5	800	1400	300
R^3	450	3	500	700	300
R^4	600	4	500	1400	300
R^5	950	2	500	800	300

This scenario produces a result that is plot in figure 6. In this figure, the granted budget (straight line) and the evolution of the mallocs and frees is plotted for each resource. As it can be seen, no reclamation is needed for resources R^1, R^2 and R^3 because there is enough memory for granting the maximum amount requested. However, at time 600, there is not enough memory to satisfy R^4 request. Specifically, this situation corresponds to case 2 (figure 5). When R^4 arrives at time 600, R^1 and R^2 are eligible. As R^2 has a higher importance (5) than R^4 (4) and the memory available is not enough (100 bytes remaining, 900 bytes reserved) the algorithm enters in phase 1 (reclaim_memory_from_Υ^k_-). In this case, the budget of R^1 (lower importance) is reduced (R^1 decreases from 1800 to 1500). When R^5 arrives, it requests its maximum memory (800), which cannot be granted with the remaining memory. This situation corresponds with case 3 (figure 5), since $M_r + M_{CR}$ does not grant the R^5_{max} but do grant R^5_{min}. The reclamation phase tries to recover memory from phase 1 (at time 950, only R^1 is an eligible resource with lower importance than R^5). However, R^1 already has decreased its budget in a previous reclamation (to grant memory to R^4 at time 600), so a second reclamation phase is executed (reclaim_memory_from_Υ^k_+). After this phase, the minimum memory can be granted to R^5. Note that R^5 could not obtain its maximum memory request because it implies to reduce the amount of memory to higher importance resources.

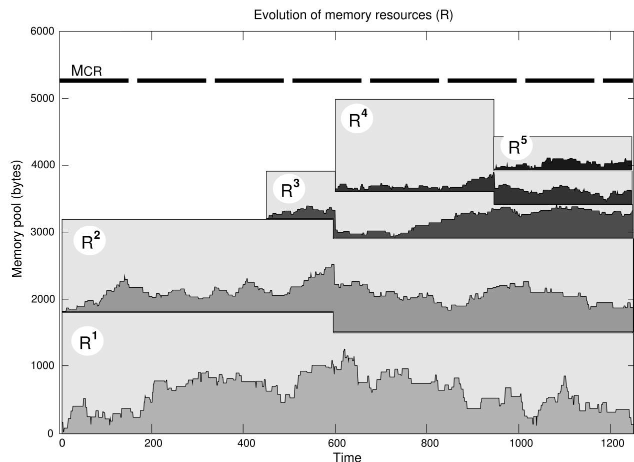


Figure 6. Scenario 1.

6.2 Scenario 2

In this scenario, R^5 has greater maximum and minimum requested memory so the total minimum memory of all resources is higher than the total usable memory. This situation corresponds to case 4 (figure 5). The graphical response of this scenario is plot in figure 7. This scenario

coincides with scenario 1 until time 950. At this time there is no remaining memory and a reclamation phase is started. However, the reclaimed memory from R^2, R^3 and R^4) is not enough to fulfil the request and the algorithm enters in phase 3 of the acceptance test, using 150 bytes of the reserved memory (M_{CR}). Note that using the reserved memory implies to assign the minimum memory.

	R^k_{at}	R^k_{imp}	R^k_{min}	R^k_{max}	R^k_{stab}
R^5	950	2	1950	2500	300

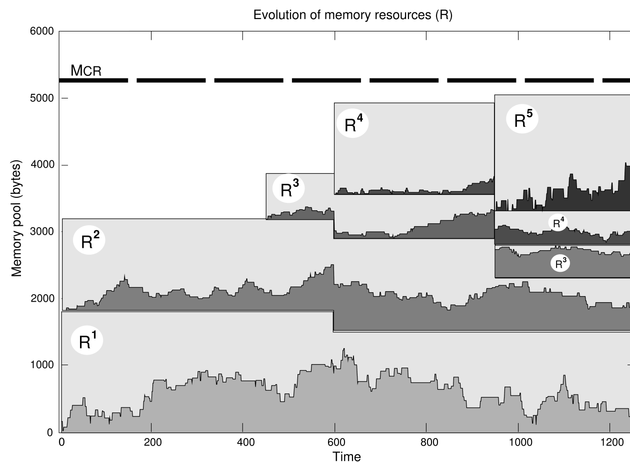


Figure 7. Scenario 2.

7 Related work

Resource reservation in real-time systems is a topic that has received the attention of researchers for more than twenty years. When considering the CPU usage, resource-based algorithms have been developed to characterise the timing requirements and processor capacity reservation requirements for real-time applications ([17, 10, 9, 3, 1, 2]).

In [22], the authors pointed out the needs of resource reservation in many application domains, such as, aerospace, multimedia, and real-time control systems. They stated a set of propositions related to the operating systems services to provide quality of service to applications. They also claimed for a more efficient memory management and a definition of the memory services interfaces at RTOS level.

The problem of task partitioning on multiprocessors, considering both CPU and memory as a resource, is formalised in [7]. However, the memory is considered as a static resource, so tasks cannot vary their memory assignment. This work proposes techniques for simultaneously considering constraints due to several resources: the computing capacity at each processor, and the amount of local memory available.

As far as the authors know, the first paper considering the dynamic memory management in real-time systems is [13]. In this paper, it is proposed to control the memory allocation and deallocations considering that tasks request memory in a periodic fashion. A feasibility test for memory is also proposed. This paper does not defined any memory reclamation and adjustment algorithm. The control of the memory was based on the knowledge about the periodic allocation and deallocation performed by the tasks.

In [21] a memory reservation mechanism (container) to monitor and control the use of resources (CPU time and resident memory) in Linux systems is proposed. The proposed mechanism isolates the memory behaviour of a group of tasks from the rest of the system. It can be used to isolate greedy applications by limiting the amount of memory, execution of virtual machines, etc. This mechanism will be included in the new version of the Linux kernel.

Our proposal could have similarities with the elastic task model proposed by Buttazzo et al [5]. This paper states that: *"whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilisation of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request."*. The MRM reclaims and reduces the granted memory to existing resources in order to accommodate a new incoming resource.

8 Conclusion and future work

This paper presents a novel memory reservation framework, jointly with an acceptance test that redistributes the available memory to improve the overall system performance.

The use of dynamic memory in real-time systems was very limited due to the unbounded nature of the basic allocation and deallocation operations. The situation changed when the TLSF algorithm was presented by Masmano et al. [15]. The TLSF is a fast constant time, $O(1)$, allocator.

The other source of indeterminism, which seriously limits the use of DSA in real-time, is the memory fragmentation problem. We summarise the main results about fragmentation, and conclude that although it is still an open problem, it is not a real problem for practical applications. The fragmentation problem is comparable with the worst case execution time (WCET) analysis. In both cases, the theoretical worst case are very pessimistic compared with the real observed fragmentation

Contrary to general belief, a detailed comparison between how the processor is scheduled and how the memory can be used in real-time systems, shows that both kinds of resources have more similarities than differences.

In the second part of the paper, a memory model for real-time applications, and a contract based framework for

managing spare memory is presented. The proposed acceptance test resembles the elastic task model, in the sense that the acceptance test distributes the spare memory among the memory resource controller that can use it. Different reclaiming memory strategies are analysed and used to adjust the available resources.

References

[1] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Journal of Real-Time Systems*, 29(2-3):131–155, 2005.

[2] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. P. Gutiérrez, and M. Trimarchi. Fsf: A real-time scheduling architecture framework. In *IEEE Real Time Technology and Applications Symposium*, pages 113–124, 2006.

[3] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22(1-2):49–75, 2002.

[4] A. Borg, A. Wellings, C. Gill, and R. K. Cytron. Real-time memory management: Life and times. *Euromicro Conference on Real-Time Systems*, 0:237–250, 2006.

[5] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *IEEE Real-Time Systems Symposium*, pages 286–295, December 1998.

[6] S. Feizabadi. Dynamic memory management in a resource-constrained real-time utility accrual environment. In *PhD Dissertation Proposal*, 2004.

[7] N. Fisher, J. H. Anderson, and S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *IEEE Real Time Technology and Applications Symposium*, 2005.

[8] FRESCOR. Framework for Real-time Embedded Systems based on COntRacts, 2007. FP6/2005/IST/5-034026 European Research Project. (http://www.frescor.org).

[9] C. Hamann, J. Loser, L. Reuther, S. Schonberg, J. Wolter, and H. Hartig. Quality-assuring scheduling: Using stochastic behavior to improve resource utilization. In *22nd IEEE Real-Time Systems Symposium*, pages 119–128, 2001.

[10] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 480–491, 1998.

[11] M. Johnstone and P. Wilson. The Memory Fragmentation Problem: Solved ? In *Proc. of the Int. Symposium on Memory Management (ISMM'98), Vancouver, Canada*. ACM Press, 1998.

[12] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[13] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 201–210, 2007.

[14] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40(2):149–179, 2008.

[15] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *16th ECRTS*, pages 79–88, Catania, Italy, July 2004. IEEE.

[16] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings. Implementation of a constant-time dynamic storage allocator. *Softw., Pract. Exper.*, 38(10):995–1026, 2008.

[17] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical report, Pittsburgh, PA, USA, 1993.

[18] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. In *14th ECRTS*, page 41, 2002.

[19] B. Randell. A Note on Storage Fragmentation and Program Segmentation. *Communications of the ACM*, 12(7):365–372, 1969.

[20] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 1977.

[21] B. Singh and V. Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Linux Symposium*, 2007.

[22] L. Steffens, G. Fohler, G. Lipari, and G. Buttazzo. Resource reservation in real-time operating systems - a joint industrial and academic position. In *ARTOSS'03*, pages 25–30, 2003.

[23] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Int. Workshop on Memory Management*, volume 986 of *LNCS*, pages 1–16. Springer-Verlag, 1995.

[24] B. Zorn and D. Grunwald. Evaluating Models of Memory Allocation. *ACM Transactions on Modeling and Computer Simulation*, pages 107–131, 1994.

Design Optimization

An FPTAS for Interface Selection in the Periodic Resource Model*

Nathan Fisher
Department of Computer Science
Wayne State University
fishern@cs.wayne.edu

Abstract

The periodic resource model *of Shin and Lee [19] provides a flexible, simple framework for designing compositional real-time systems. Each component in the periodic resource model has an interface which specifies the* period *and* capacity *of the resource used to schedule the component. Unfortunately, the best-known exact algorithms for determining the interface parameters for a component in the periodic resource model potentially require exponential or pseudo-polynomial time. In this paper, we obtain an FPTAS for the problem of selecting an interface period given a capacity-determination algorithm. We also apply our approach to obtain an FPTAS for the problem of selecting both a period and capacity for components consisting of sporadic task systems. Our algorithms obtain interface parameters with interface bandwidth at most $(1 + \epsilon)$ times the optimal for any $\epsilon > 0$.*

1 Introduction

Compositional analysis for real-time systems has recently received considerable research attention due to its well-known benefits of reducing system-design complexity. For component-based systems, reduction in system complexity is typically achieved via *component abstraction.* Component abstraction hides the complexity and internal details of each component from developers of other components and only exposes information necessary to use the component via an interface. Numerous compositional frameworks have been proposed to support the design and analysis of compositional real-time systems (for a non-exhaustive list see [1, 7, 10, 19]). In each of these compositional frameworks, a component expresses its computational requirements to the system via a *real-time interface.* One important attribute of a real-time interface is the *interface bandwidth.* The interface bandwidth simultaneously quantifies the fraction of the total system resource supply that a component C will require to meet its real-time constraints and the component C's "interference" on the resource supply provided to other system components. Thus, a fundamental goal in the design and analysis of compositional real-time systems is the *minimization of real-time interface bandwidth* (MIB-RT).

The *periodic resource model* [19] is a simple, yet flexible real-time compositional framework. A periodic resource, denoted by $\Gamma = (\Pi, \Theta)$, guarantees that a component C executed upon resource Γ will receive at least Θ units of execution (not necessarily contiguous) between successive time points $\{t \equiv t_0 + \ell\Pi \mid \ell \in \mathbb{N}\}$, given some initial resource start-time t_0. The parameters Π and Θ are respectively referred to as the *period* and *capacity* of the periodic resource Γ. It will be assumed throughout the paper that Π is a positive integer while Θ may be a real number. The ratio $\frac{\Theta}{\Pi}$ represents the interface bandwidth of resource Γ. A system-level scheduling algorithm allocates the processor time among the different periodic resources that share the same processor, such that each resource receives (for every period) aggregate processor time equivalent to its capacity. A subsystem's tasks are then hierarchically scheduled by a subsystem-level scheduling algorithm upon the processing time supplied to resource Γ. If the system-level scheduling algorithm is earliest-deadline-first (EDF), then it is known (e.g., see [19]) that periodic resources $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ can successfully guarantee the capacity parameters to their respective components, if and only if, the total system bandwidth does not exceed one; i.e., $\sum_{i=1}^{m} \frac{\Theta_i}{\Pi_i} \leq 1$. From the aforementioned condition, it is clear that minimizing the interface bandwidth of each resource increases system schedulability.

In this paper, we specifically consider the problem of determining the optimal choice of period parameter (i.e., Π) for resource Γ, given a component C and a *capacity-determination algorithm* $\mathcal{A}$. A capacity-determination algorithm returns the minimum-schedulable capacity for a given component C to meet all deadlines upon a periodic resource with a fixed period Π. Such algorithms have already

*This research has been supported by a Wayne State University Faculty Research Award.

been devised for sporadic tasks; e.g., see [9, 12, 19]. Let $\Theta^{\mathcal{A}}_{\min}(\Pi, C)$ be the value returned by capacity-determination algorithm $\mathcal{A}$ for a given C and Π. Furthermore, let us consider only integer values of Π in the range $\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ as possible periods. The optimal choice period for the MIB-RT problem is:

$$\Pi^*(\mathcal{A}, C) \stackrel{\text{def}}{=} \arg\min_{\Pi \in \mathbb{N}^+} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \mid \Pi_{\text{lower}} \leq \Pi \leq \Pi_{\text{upper}} \right\}. \quad (1)$$

Let OPT denote the optimal capacity-determination algorithm. The problem of exactly determining $\Pi^*(\mathsf{OPT}, C)$ for the period resource model (and slightly more general models) has been extensively studied by Easwaran [8]. However, each of the proposed methods has runtime dependent of the difference between Π_{lower} and Π_{upper}. If this difference is large, determining $\Pi^*(\mathcal{A}, C)$ may be prohibitively expensive from a time-complexity perspective, especially if the capacity-determination algorithm $\mathcal{A}$ also has significant computational complexity. Such a large computation cost prohibits exact algorithms from being used for "on-the-fly" computation of interfaces.

§Our Contributions. Our main objective is to find an approximation to $\Pi^*(\mathcal{A}, C)$ with bounded deviation from the optimal solution to MIB-RT. In this paper, we present an approximation scheme with the following guarantee:

> Given a component C, capacity-determination algorithm $\mathcal{A}$, range of possible period values $\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$, and an accuracy parameter $\epsilon > 0$, if our algorithm returns $\widehat{\Pi}$ for the given parameters, then $\frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A},C),C)}{\Pi^*(\mathcal{A},C)} \leq \frac{\Theta^{\mathcal{A}}_{\min}(\widehat{\Pi},C)}{\widehat{\Pi}} \leq (1+\epsilon) \cdot \frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A},C),C)}{\Pi^*(\mathcal{A},C)}$. Furthermore, our algorithm runs in time polynomial in the representation of C, Π_{lower}, Π_{upper}, $\frac{1}{\epsilon}$, and the complexity of $\mathcal{A}$.

In other words, our algorithm is a fully-polynomial-time approximation scheme (FPTAS) [20] for the MIB-RT problem (with respect to the computation complexity of $\mathcal{A}$). The $(1 + \epsilon)$ factor is called the *approximation ratio* of the produced solution. We also give an FPTAS for determining both the period and capacity of a component consisting of sporadic tasks.

§Organization. The remainder of the paper is organized as follows. Section 2 briefly describes prior related research on the MIB-RT problem. Section 3 presents our proposed approximation scheme for period selection (with respect to any given capacity-determination algorithm) and proves its associated approximation ratio. Section 4 shows how the approximation scheme given in the Section 3 may be used to obtain an overall approximation scheme (with respect to the optimal capacity-determination) for components consisting of sporadic real-time tasks.

2 Related Work

The MIB-RT problem has previously been studied for the periodic resource model and the *explicit-deadline periodic resource* (EDP) model [9] where each component is represented by a *sporadic task system* [16]. Easwaran et al. [9] also obtain exact solutions to MIB-RT in this context (i.e., if the bandwidth provided by the system to component C is less than the exact solution to MIB-RT, then some real-time constraint will be violated for C). This solution is based upon exact schedulability techniques for uniprocessor real-time systems [4, 14], which may be computationally expensive. Easwaran [8] also explored period selection in the EDP model (which can trivially be applied to the periodic resource model); however, the worst-case complexity of the proposed algorithms could be potentially exponential in the number of tasks in the component. Shin and Lee [19] have obtained $O(n)$-time, sufficient solutions to MIB-RT for the periodic resource. The advantage of this approach is that bandwidth allocation may be determined quickly for a component C. However, our previous analysis [11] of Shin and Lee's linear-time algorithm showed that, for a fixed resource period, there exists sporadic task systems that would cause the algorithm to return a bandwidth that is a factor of 1.5 greater than the optimal bandwidth. (It is also shown that the most that the returned bandwidth could exceed optimal is by a factor of 3). As an intermediate solution between computationally-expensive, exact solutions and efficient, inexact solutions, Fisher and Dewan [12] obtain an FPTAS for capacity-determination in the periodic resource model with fixed resource periods. In this paper, we remove the assumption of fixed resource periods.

A number of other results on MIB-RT for different compositional models exist. For components scheduled by fixed-priority on *temporal partitions*, Lipari and Bini [15], developed an exact, pseudo-polynomial-time algorithm for MIB-RT. While Almeida and Pedreiras [3] developed sufficient, polynomial-time bandwidth allocation techniques for fixed-priority scheduling upon temporal partitions. Recently, researchers have also focused on characterizing components by processor-demand curves which describe the minimum amount of processing that a component requires over any interval. For example, Wandeler and Thiele [21] proposed the concept of *interface-based design* which uses real-time

calculus [6] to compute demand curves and service curves for each component in a compositional real-time system. In another demand-based model, Albers et al. [1] have developed parametric algorithms for MIB-RT (without known approximation ratios) for the *hierarchical event stream model*.

3 An Algorithm for Selecting the Interface Period

In this section, we describe an algorithm for selecting a "near-optimal" interface period for a component C, with respect to a given capacity-determination algorithm $\mathcal{A}$. However, to guarantee that the approximation ratio of our proposed algorithm is not too large, we need to formally restrict the types of capacity-determination algorithms that we consider. Informally, we will only consider algorithms where the capacity grows with increasing periods. The formal definition is given in the following.

Definition 1 *For any component C, an algorithm $\mathcal{A}$ is a* **monotonically non-decreasing capacity-determination algorithm** *over $\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$, if for all $\Pi_1, \Pi_2 \in \{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ such that $\Pi_1 \le \Pi_2$, it must be that $\Theta^{\mathcal{A}}_{\min}(\Pi_1, C) \le \Theta^{\mathcal{A}}_{\min}(\Pi_2, C)$.*

Note that this definition does not place a constraint on the types of problems that can be solved by our period-selection algorithm, as all known capacity-determination algorithms [9, 12, 19] for the periodic resource model possess this property. Section 4 will formally show that the algorithm of [12] is monotonically non-decreasing in Π. In the remainder of this section, we will briefly describe our algorithm (Section 3.1) and prove its correctness (Section 3.2).

Algorithm 1 SelectInterface$(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$.

Require: Component C, resource-capacity determination algorithm $\mathcal{A}$, positive integers Π_{lower} and Π_{upper}, and positive real number $\epsilon : 0 < \epsilon \le 1$.

Ensure: $\frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*_{\mathcal{A}}, C)}{\Pi^*_{\mathcal{A}}} \le \frac{\Theta^{\mathcal{A}}_{\min}(\widehat{\Pi}, C)}{\widehat{\Pi}} \le (1+\epsilon) \cdot \frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*_{\mathcal{A}}, C)}{\Pi^*_{\mathcal{A}}}$ where $\Pi_{\text{lower}} \le \Pi^*_{\mathcal{A}} \le \Pi_{\text{upper}}$.

```
1:  Π_last ⇐ Π_lower
2:  Θ_last ⇐ Θ^A_min(Π_last, C)
3:  Θ_upper ⇐ Θ^A_min(Π_upper, C)
4:  Π̂ ⇐ Π_last
5:  Θ̂ ⇐ Θ_last
6:  while (1 + ε)Θ_last ≤ Θ_upper do
7:      Perform a binary search over Π_last to Π_upper for
        largest Π s.t. (Θ =def Θ^A_min(Π, C)) ≤ (1+ε)Θ_last.
8:      Π_last ⇐ Π + 1
9:      Θ_last ⇐ Θ^A_min(Π_last, C)
10:     if Θ/Π > 1 then
11:         return "C not schedulable."
12:     end if
13:     if Θ/Π < Θ̂/Π̂ then
14:         Θ̂ ⇐ Θ
15:         Π̂ ⇐ Π
16:     end if
17:     if Θ_last/Π_last < Θ̂/Π̂ then
18:         Θ̂ ⇐ Θ_last
19:         Π̂ ⇐ Π_last
20:     end if
21: end while
22: if Θ_upper/Π_upper < Θ̂/Π̂ then
23:     return Γ̂ = (Π_upper, Θ_upper)
24: else
25:     return Γ̂ = (Π̂, Θ̂)
26: end if
```

3.1 Algorithm Description

The period-selection algorithm, SelectInterface$(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$, is simple; the algorithm evaluates $\Theta^{\mathcal{A}}_{\min}(\Pi, C)$ for select values of Π between Π_{lower} and Π_{upper} and returns the Π and $\Theta^{\mathcal{A}}_{\min}(\Pi, C)$ with the minimum $\frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi}$. The values of Π are selected based on an accuracy parameter ϵ. Pseudocode for SelectInterface$(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ is given below.

Lines 1 through 5 initialize the first choice for Π (equal to Π_{lower}), the corresponding minimum capacity (as determined by $\mathcal{A}$), and other bookkeeping variables. The *while* loop of Lines 6 and 21 iterate through successive choices of Π that have capacity ratios (i.e., the ratio between the capacity of Π and the capacity for the previous choice of Π) of at most $(1+\epsilon)$. To find the next choice of Π, we use a *binary search* (Line 7) over the range of remaining values of Π that have not been selected. The binary search returns the largest Π such that the capacity of Π is no more than $(1+\epsilon)$ times the capacity of the previously-selected value of Π. Π is incremented (Line 8) within the while loop to ensure that the next capacity-value for Π is more than $(1+\epsilon)$ times the previous. Finally, the while loop terminates when our returned capacity exceeds the capacity of Π_{upper}. The values $\widehat{\Theta}$ and $\widehat{\Pi}$ maintain the interface parameters of the resource with the minimum capacity of all values of Π that have been evaluated. The algorithm returns the minimum-bandwidth interface over all evaluated choices of Π.

3.2 Proof of Correctness

We now must show that SelectInterface$(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ returns an interface with bandwidth no more than a factor of $(1+\epsilon)$ greater than the optimal

bandwidth. That is, we need to show that $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ has an approximation ratio equal to $(1+\epsilon)$. The first result that we give towards this goal is a lower bound on the bandwidth for any contiguous range of periods $\{\Pi_i, \ldots, \Pi_j\}$.

Lemma 1 *Consider any* $\Pi_i, \Pi_j \in \{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ *where* $\Pi_i \leq \Pi_j$ *Given any monotonically non-decreasing capacity-allocation algorithm* $\mathcal{A}$ *and component* C*, the following is true.*

$$\frac{\Theta^{\mathcal{A}}_{\min}(\Pi_i, C)}{\Pi_j} \leq \min_{\Pi \in \{\Pi_i, \Pi_{i+1}, \ldots, \Pi_j\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\}. \quad (2)$$

Proof: Observe that, since $\mathcal{A}$ is monotonically non-decreasing over $\{\Pi_i, \ldots, \Pi_j\}$, $\Theta^{\mathcal{A}}_{\min}(\Pi_i, C) \leq \Theta^{\mathcal{A}}_{\min}(\Pi, C)$ for all $\Pi \in \{\Pi_i, \ldots, \Pi_j\}$. Also, $\frac{1}{\Pi_j} \leq \frac{1}{\Pi}$ is trivially true for all $\Pi \in \{\Pi_i, \ldots, \Pi_j\}$. Equation 2 follows from these two observations. ■

We now show that if we select period values from $\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ such that consecutive choices from this domain are either adjacent (i.e., the values are different by one) or have computed capacities different by at most a multiplicative factor of $(1+\epsilon)$, then the minimum bandwidth resulting from these choices is at most a factor of $(1+\epsilon)$ greater than the optimal minimum bandwidth.

Lemma 2 *Consider monotonically non-decreasing capacity-allocation algorithm* $\mathcal{A}$*, component* C*, and real number* $\epsilon > 0$*. Let* $\{\Pi_1, \ldots, \Pi_m\}$ *be any (ordered) subset of*$\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ *such that* $\Pi_1 = \Pi_{\text{lower}}$*,* $\Pi_m = \Pi_{\text{upper}}$*, and, for all* $i : 1 \leq i < m$*, either*

$$\Pi_{i+1} = \Pi_i + 1, \quad (3)$$

or

$$\Theta^{\mathcal{A}}_{\min}(\Pi_i, C) \leq \Theta^{\mathcal{A}}_{\min}(\Pi_{i+1}, C) \leq (1+\epsilon) \cdot \Theta^{\mathcal{A}}_{\min}(\Pi_i, C) \quad (4)$$

is true. Then, the following inequalities holds:

$$\begin{aligned} &\frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)} \\ &\quad \leq \min_{\Pi \in \{\Pi_1, \ldots, \Pi_m\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\} \\ &\quad \leq (1+\epsilon) \cdot \frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)}. \end{aligned} \quad (5)$$

Proof: There are two cases to consider:

1. $\Pi^*(\mathcal{A}, C) \in \{\Pi_1, \ldots, \Pi_m\}$; or
2. $\Pi^*(\mathcal{A}, C) \notin \{\Pi_1, \ldots, \Pi_m\}$.

For Case 1, it obvious that $\min_{\Pi \in \{\Pi_1, \ldots, \Pi_m\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\}$ equals $\frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)}$; Equation 5 follows.

For Case 2, there must exist adjacent values $\Pi_i, \Pi_{i+1} \in \{\Pi_1, \ldots, \Pi_m\}$ such that $\Pi_i \leq \Pi^*(\mathcal{A}, C) \leq \Pi_{i+1}$. Furthermore, $\Pi_{i+1} \neq \Pi_i + 1$; otherwise, $\Pi^*(\mathcal{A}, C)$ would equal either Π_i or Π_{i+1}, and Case 1 would apply. By Lemma 1,

$$\begin{aligned} \frac{\Theta^{\mathcal{A}}_{\min}(\Pi_i, C)}{\Pi_{i+1}} &\leq \min_{\Pi \in \{\Pi_i, \ldots, \Pi_{i+1}\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\} \\ &= \frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)} \\ \Rightarrow \frac{(1+\epsilon)\Theta^{\mathcal{A}}_{\min}(\Pi_i, C)}{\Pi_{i+1}} &\leq \frac{(1+\epsilon)\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)} \\ \Rightarrow \frac{\Theta^{\mathcal{A}}_{\min}(\Pi_{i+1}, C)}{\Pi_{i+1}} &\leq \frac{(1+\epsilon)\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)} \\ &(\text{by Equation 4}). \end{aligned} \quad (6)$$

The final inequality implies

$$\begin{aligned} &\min_{\Pi \in \{\Pi_1, \ldots, \Pi_m\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\} \\ &\leq \frac{(1+\epsilon)\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)}. \end{aligned}$$

Obviously,

$$\begin{aligned} &\frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A}, C), C)}{\Pi^*(\mathcal{A}, C)} \\ &\quad \leq \min_{\Pi \in \{\Pi_1, \ldots, \Pi_m\}} \left\{ \frac{\Theta^{\mathcal{A}}_{\min}(\Pi, C)}{\Pi} \right\}, \end{aligned}$$

by definition of $\Pi^*(\mathcal{A}, C)$. From the preceding two inequalities, Equation 5 of the lemma follows. ■

Finally, we use Lemma 1 and 2 to show that $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ is an approximation scheme with approximation ratio $(1+\epsilon)$. Furthermore, we quantify the running time of the algorithm in the following theorem.

Theorem 1 $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ *has an approximation ratio of* $(1+\epsilon)$ *(where* $0 < \epsilon \leq 1$*) for period selection with respect to any monotonically, non-decreasing capacity-allocation algorithm* $\mathcal{A}$*. Furthermore, the algorithm has time complexity equal to*

$$O\left(\chi^{\mathcal{A}}(C) \cdot \lg\left(\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right) \cdot \lg\left(\Pi_{\text{upper}}\right) \Big/ \epsilon\right), \quad (7)$$

where $\Theta_{\text{lower}} \stackrel{\text{def}}{=} \Theta^{\mathcal{A}}_{\min}(\Pi_{\text{lower}}, C)$*,* $\Theta_{\text{upper}} \stackrel{\text{def}}{=} \Theta^{\mathcal{A}}_{\min}(\Pi_{\text{upper}}, C)$*, and* $\chi^{\mathcal{A}}(C)$ *equals the time complexity of the capacity-determination algorithm* $\mathcal{A}$ *given a component* C*.*

Proof Sketch: Let $\{\Pi_1, \ldots, \Pi_m\}$ be the ordered set of values that variables Π_{last} and Π (set in Line 7) are assigned throughout the execution of $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$. Obviously, $\{\Pi_1, \ldots, \Pi_m\} \subseteq \{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$ such that $\Pi_1 = \Pi_{\text{lower}}$ and $\Pi_m = \Pi_{\text{upper}}$. Furthermore, it is easy to verify that subsequent values of $\Pi_i, \Pi_{i+1} \in \{\Pi_1, \ldots, \Pi_m\}$ satisfy either Equation 3 (see Line 8) or Equation 4 (see Line 7) of

Lemma 2. Thus, by Equation 5 of Lemma 2, the $\widehat{\Gamma} = \left(\widehat{\Pi}, \widehat{\Theta}\right)$ computed by Lines 4, 5, 13, 17, and 22 is at most $(1+\epsilon) \cdot \frac{\Theta^{\mathcal{A}}_{\min}(\Pi^*(\mathcal{A},C),C)}{\Pi^*(\mathcal{A},C)}$. This shows that $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$ has an approximation ratio of at most $(1+\epsilon)$.

For determining the complexity of $\mathsf{SelectInterface}(C, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \epsilon)$, observe that the running time is dominated by the *while* loop in Lines 6 through 21. The complexity of the *while* loop can (informally) be determined by

$$\begin{aligned} & (\text{Number of iterations of } \textit{while} \text{ loop}) \\ \times\ & (\text{Number of } \Pi \text{ values to be checked in } \textit{binary search}) \\ \times\ & (\text{Execution time to check value of } \Pi)\,. \end{aligned} \tag{8}$$

The number of iterations of the *while* loop can be determined by observing that Θ_{last} increases by at least a factor of $(1+\epsilon)$ upon every iteration of the *while* loop of Lines 6 to 21. Thus, the number of iterations is equal to smallest integer value of ℓ such that the following equation is true:

$$(1+\epsilon)^{\ell}\,\Theta_{\text{lower}} \geq \Theta_{\text{upper}}$$

Solving for ℓ,

$$\begin{aligned} \ell &= \left\lceil \log_{(1+\epsilon)}\left(\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right)\right\rceil \\ &= \left\lceil \frac{\ln\left(\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right)}{\ln(1+\epsilon)} \right\rceil \\ &\leq \left\lceil \frac{(1+\epsilon)\cdot\ln\left(\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right)}{\epsilon} \right\rceil \\ &= O\left(\left(\lg\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right)/\epsilon\right). \end{aligned} \tag{9}$$

The third equality above follows from the well-known identity $\frac{x}{1+x} \leq \ln(1+x)$ for all $x > -1$.

The *binary search* of Line 7 searches over the range $\{\Pi_{\text{last}}, \ldots, \Pi_{\text{upper}}\}$. This range contains at most Π_{upper} number of values. Thus, the number of values of $\Pi \in \{\Pi_{\text{last}}, \ldots, \Pi_{\text{upper}}\}$ that have to calculate $\Theta^{\mathcal{A}}_{\min}(\Pi, C)$ is

$$O\left(\lg(\Pi_{\text{upper}})\right). \tag{10}$$

Finally, the execution time for each calculation of $\Theta^{\mathcal{A}}_{\min}(\Pi, C)$ is equal to the execution cost of algorithm $\mathcal{A}$ for a given component, which is denoted $\chi^{\mathcal{A}}(C)$. Combining this observation with Equations 8, 9, and 10, implies the running time given in the lemma. ■

4 Interface Selection for Sporadic Task Systems

In this section, we explore an application of the algorithm proposed in the previous section to determine the minimum-bandwidth interface for a component consisting of sporadic tasks scheduled by EDF. We will show that we may, in fact, obtain an FPTAS for the MIB-RT problem in the context of sporadic tasks. The remainder of the section is organized as follows. In Section 4.1, we introduce notations and prior analytic results for the task, workload, and periodic resource models. In Section 4.2, we state the capacity-determination algorithm we use in deriving our FPTAS; the capacity-determination algorithm was previously proposed in [12]. In Section 4.3, we give a simple example to show that using Π_{lower} is not always the optimal choice for minimizing the interface bandwidth. In Section 4.4, we give a description of our FPTAS and prove its correctness.

4.1 Models and Notation

In this subsection, we present background and notation for the task model, workload functions, and periodic resource model that we use throughout the remainder of this section.

§Sporadic Task Model. A **sporadic task** $\tau_i = (e_i, d_i, p_i)$ is characterized by a *worst-case execution requirement* e_i, a *(relative) deadline* d_i, and a *minimum inter-arrival separation* p_i, which is, for historical reasons, also referred to as the *period* of the task. Such a sporadic task generates a potentially infinite sequence of jobs, with successive job-arrivals separated by at least p_i time units. Each job has a worst-case execution requirement equal to e_i and a deadline that occurs d_i time units after its arrival time. We will assume that task parameters are positive integers. Furthermore, obviously, $e_i \leq d_i$ and $e_i \leq p_i$ for any task τ_i; otherwise, cannot be scheduled to meet its deadline by any scheduling algorithm. A *sporadic task system* $\tau \stackrel{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$ is a collection of n such sporadic tasks. A useful metric for a sporadic task τ_i is the *task utilization* $u_i \stackrel{\text{def}}{=} e_i/p_i$. The system utilization is denoted $U(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i$.

The following lemma on system utilization will be useful in defining an FPTAS.

Lemma 3 *For any sporadic task system τ with positive integer parameters,*

$$U(\tau) \geq \frac{n}{p_{\max}} \tag{11}$$

where $p_{\max} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{p_i\}$.

Proof: By definition, $U(\tau)$ equals $\sum_{\tau_i \in \tau} \frac{e_i}{p_i}$. Equation 11 follows from observing that $e_i \geq 1$ and $p_i \leq p_{\max}$ for all $\tau_i \in \tau$. ■

§Workload Functions. For determining schedulability of a sporadic task system, it is often useful to quantify the maximum amount of execution that must complete over any given interval. For this purpose, researchers [5] have derived the *demand-bound function*, defined below.

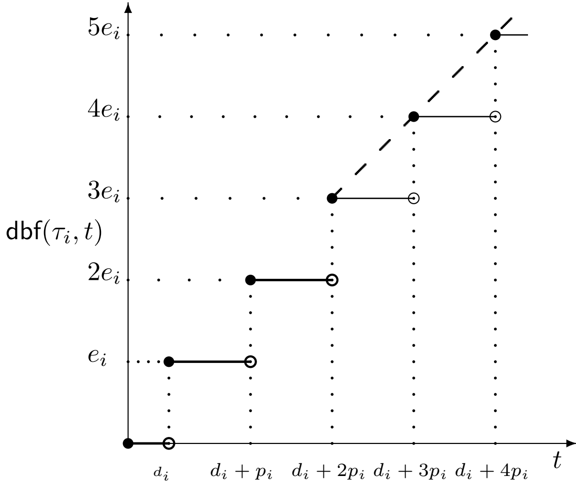


Figure 1. The step function denotes a plot of $\mathsf{dbf}(\tau_i, t)$ as a function of t. The dashed line represents the function $\widetilde{\mathsf{dbf}}(\tau_i, t, k)$, approximating $\mathsf{dbf}(\tau_i, t)$. $\widetilde{\mathsf{dbf}}(\tau_i, t, k)$ is equal to $\mathsf{dbf}(\tau_i, t)$ for all $t < d_i + (k-1)p_i$ (k equals three in the above graph).

Definition 2 (Demand-Bound Function) *For any $t > 0$ and task τ_i, the* **demand-bound function** *(*dbf*) quantifies the maximum cumulative execution requirement of all jobs of τ_i that could have both an arrival time and deadline in any interval of length t. Baruah et al. [5] have shown that, for sporadic tasks,* dbf *can be calculated as follows.*

$$\mathsf{dbf}(\tau_i, t) = \max\left(0, \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) \cdot e_i. \quad (12)$$

Figure 1 gives a visual depiction of the demand-bound function for a sporadic task τ_i. Observe from the above definition and Figure 1 that the dbf is a right continuous function with discontinuities at time points of the form $t \equiv d_i + a \cdot p_i$ where $a \in \mathbb{N}$. Let $\mathsf{DBF}(\tau, t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \mathsf{dbf}(\tau_i, t)$. It has been shown [5] that condition $\mathsf{DBF}(\tau, t) \leq t, \forall t \geq 0$ is necessary and sufficient for sporadic task system τ to be EDF-schedulable upon a preemptive uniprocessor platform of unit speed. Furthermore, it has also been shown that the aforementioned condition needs to be verified at only time points in the following *ordered* set (with elements are in non-decreasing order):

$$\mathsf{TS}(\tau) \stackrel{\text{def}}{=} \bigcup_{\tau_i \in \tau} \{t \equiv d_i + a \cdot p_i \mid (a \in \mathbb{N}) \wedge (t \leq P(\tau))\}. \quad (13)$$

where $P(\tau)$ is an upper bound on the maximum time instant that the schedulability condition must be verified at. For EDF-scheduled sporadic task systems on preemptive unit-speed processors, $P(\tau)$ is at most $\mathsf{lcm}_{\tau_i \in \tau}\{p_i\}$. The above set is known as the **testing set** for sporadic task system τ. For any $t_a \in \mathsf{TS}(\tau)$, $t_a \leq t_{a+1}$; if t_a is the last element of the set, we use the convention that t_{a+1} equals ∞. Also, we will assume that t_0 is equal to zero.

Albers and Slomka [2] proposed the following approximation to dbf to reduce the number of discontinuities (and, thus, points in the testing set).

$$\widetilde{\mathsf{dbf}}(\tau_i, t, k) \stackrel{\text{def}}{=} \begin{cases} \mathsf{dbf}(\tau_i, t), & \text{if } t < d_i + (k-1)p_i; \\ u_i \cdot (t - d_i) + e_i, & \text{otherwise.} \end{cases} \quad (14)$$

The main intuition behind $\widetilde{\mathsf{dbf}}(\tau_i, t, k)$ is that it "tracks" dbf for exactly k discontinuities (i.e., "steps"). After k discontinuities, $\widetilde{\mathsf{dbf}}(\tau_i, t, k)$ using a linear interpolation of the subsequent discontinuous points (with slope equal to u_i). The steps with the thick lines and the sloped-dotted line in Figure 1 correspond to $\mathsf{dbf}(\tau_i, t, 3)$. We will abuse notation slightly and use the convention that $\mathsf{dbf}(\tau_i, t, \infty)$ corresponds to $\mathsf{dbf}(\tau_i, t)$. Let $\widetilde{\mathsf{DBF}}(\tau, t, k) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \widetilde{\mathsf{dbf}}(\tau_i, t, k)$. Albers and Slomka show [2], for any fixed $k \in \mathbb{N}^+$, the condition $\widetilde{\mathsf{DBF}}(\tau, t, k) \leq t, \forall t \geq 0$ is sufficient for sporadic task τ to be EDF-schedulable upon a preemptive uniprocessor platform of unit speed. The ordered testing set of this condition is reduced to

$$\widetilde{\mathsf{TS}}(\tau, k) \stackrel{\text{def}}{=} \bigcup_{\tau_i \in \tau} \{t \equiv d_i + a \cdot p_i \mid (a \in \mathbb{N}) \wedge (a < k) \wedge (t \leq P(\tau))\}. \quad (15)$$

§Periodic Resource Model. Throughout this section, we assume that each component C is a sporadic task system. Let C be composed of a sporadic task system τ that is to be EDF-scheduled upon periodic resource $\Gamma = (\Pi, \Theta)$. (From now on, we use τ in the context of component C). We will now present some concepts that have been previously-introduced by researchers to determine whether τ will meet all deadlines when scheduled upon Γ.

Definition 3 (Supply-Bound Function) *For any $t > 0$, the* **supply-bound function** *(*sbf*) quantifies the minimum execution supply that a component executed upon periodic resource Γ may receive over any interval of length t. Shin and Lee [19] have quantified the supply bound function for an periodic resource in the following (using the notation of Easwaran et al. [9]):*

$$\mathsf{sbf}(\Gamma, t) = \begin{cases} y_\Gamma \Theta + \max(0, t - x_\Gamma - y_\Gamma \Pi), & \text{if } t \geq \Pi - \Theta \\ 0, & \text{otherwise.} \end{cases} \quad (16)$$

where $y_\Gamma = \left\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \right\rfloor$ *and* $x_\Gamma = (2\Pi - 2\Theta)$.

EDF-schedulability conditions for periodic resource Γ have been developed [9, 17, 18], as given in the following theorem.

Theorem 2 (from [9]) *A sporadic task system τ is* EDF-*schedulable upon an EDP resource $\Gamma = (\Pi, \Theta)$, if and only if,*

$$(\mathsf{DBF}(\tau,t) \leq \mathsf{sbf}(\Gamma,t),\ \forall t \leq P(\tau)) \bigwedge \left(U(\tau) \leq \frac{\Theta}{\Pi}\right) \quad (17)$$

where $P(\tau)$ equals $\mathsf{lcm}_{\tau_i \in \tau}\{p_i\} + \max_{\tau_i \in \tau}\{d_i\}$.

4.2 Capacity-Determination Algorithm

The algorithm that we consider for determining the minimum-capacity of a periodic resource (for a fixed period) is given by Algorithm 2. Informally, the algorithm works as follows: for each value t in the testing set $\widetilde{\mathsf{TS}}(\tau, k)$, MinimumCapacity determines the minimum capacity $\Theta_t^{\min}$ such that the $\widetilde{\mathsf{DBF}}(\tau, t)$ is "below" the supply-bound function $\mathsf{sbf}((\Pi, \Theta_t^{\min}), t)$. The algorithm is exact when k equals ∞, since $\widetilde{\mathsf{TS}}(\tau, \infty)$ equals $\mathsf{TS}(\tau)$. More details on the algorithm and a proof of correctness is available from [12]. Please note that the algorithm found in [12] is given for the more general explicit-deadline periodic resource (EDP) model [9]; the algorithm stated here has been adapted to for the periodic resource model.

Algorithm 2 MINIMUMCAPACITY(Π, τ, k).

Require: Sporadic task system τ, resource period Π (positive integer), and positive integer k.

1: $\Theta^{\min} \leftarrow U(\tau) \cdot \Pi$
2: **for all** $t \in \widetilde{\mathsf{TS}}(\tau, k)$ **do**
3: $\quad D_t \leftarrow \widetilde{\mathsf{DBF}}(\tau, t, k)$
4: $\quad \alpha_t \leftarrow \sum_{\tau_i \in \tau : t \geq d_i + (k-1)p_i} u_i$
5: $\quad \Theta_t^{\min} \leftarrow \infty$
6: $\quad$ **for** $\ell = \max\left\{1, \left\lfloor \frac{t}{\Pi} \right\rfloor - 1\right\}$ to $\left\lceil \frac{t}{\Pi} \right\rceil$ **do**
7: $\quad\quad \Theta_\ell^{\min} \leftarrow \max \left\{ \begin{array}{c} \alpha_t \Pi, \\ \frac{D_t - t + (\ell+1)\Pi}{\ell+1}, \\ \frac{D_t}{\ell}, \\ \frac{D_t + \alpha_t((\ell+2)\Pi - t)}{\ell + 2\alpha_t} \end{array} \right\}$
8: $\quad\quad \Theta_t^{\min} \leftarrow \min\{\Theta_t^{\min}, \Theta_\ell^{\min}\}$
9: $\quad$ **end for**
10: $\quad \Theta^{\min} \leftarrow \max\{\Theta^{\min}, \Theta_t^{\min}\}$
11: **end for**
12: **return** $\Theta^{\min}$

The following result (restated from [12]) shows that MinimumCapacity is an FPTAS for capacity-determination given a fixed period.

Theorem 3 (from [12]) *Given Π, τ, and $\epsilon > 0$, the procedure* $\mathsf{MinimumCapacity}\left(\Pi, \tau, \left\lceil \frac{1}{\epsilon} \right\rceil\right)$ *returns $\Theta^{\min}$ such that*

$$\Theta^*(\Pi, \tau) \leq \Theta^{\min} \leq (1 + \epsilon) \cdot \Theta^*(\Pi, \tau)$$

where $\Theta^*(\Pi, \tau)$ is the optimal minimum capacity required to EDF-*schedule τ upon a periodic resource with period of Π. Furthermore,* $\mathsf{MinimumCapacity}\left(\Pi, \tau, \left\lceil \frac{1}{\epsilon} \right\rceil\right)$ *has time complexity $O\left(\frac{n \lg n}{\epsilon}\right)$.*[1]

4.3 A Motivating Example

At this point in the paper, the reader may wonder, if a monotonically non-decreasing capacity-determination algorithm is used, does the minimum bandwidth occur for smallest possible value of Π (i.e., Π_{lower}) – eliminating the need for more complex period-selection algorithm? In this subsection, we show that such a period-selection algorithm is definitely required. We give a small motivating example to illustrate that the minimum bandwidth is not always achieved using the period Π_{lower}.

Consider a component consisting of a single sporadic task $\tau_1 \stackrel{\text{def}}{=} (e_1, d_1, p_1) = (1, 301, 1000)$. If the range of possible Π values is $\{80, 81, \ldots, 150\}$ and we use an exact capacity determination algorithm (i.e., MinimumCapacity with $k = \infty$), then we will find that the minimum capacity to successfully schedule τ_1 is equal to 0.5 for all $\Pi \in \{80, 81, \ldots, 100\}$ and 1.0 for all $\Pi \in \{101, 102, \ldots, 150\}$. Thus, the minimum bandwidth interface for τ_1 is $\Gamma = (100, 0.5)$ which has a bandwidth of 0.005. However, the approach of checking all values of Π would have required us to call MinimumCapacity a total of 71 times. Furthermore, observe that the minimum bandwidth interface did not have a period of Π_{lower}, but in fact occurred 20 units greater than Π. We expect that this example can be generalized so that the difference from Π_{lower} to the optimal period is arbitrarily large; thus, checking all values of Π for the minimum bandwidth interface is potentially very expensive. On the other hand, $\mathsf{SelectInterface}(\{\tau_1\}, \mathsf{MinimumCapacity}, 80, 150, .1)$ would call MinimumCapacity a total of nine times and, in fact, return the interface $\Gamma = (100, 0.5)$ for this example. Thus, for this particular example, a significant reduction in time complexity can be obtained with no loss of accuracy. The next subsection shows how we may obtain an FPTAS that guarantees the desired level of accuracy for the selected interface of any possible component.

4.4 An FPTAS for Interface Selection

In this subsection, we present the main result of the section: an FPTAS for interface selection for periodic resources scheduling sporadic tasks. Before we state our main result, we require two technical lemmas. The first lemma gives an upper and lower bound on the value returned by MinimumCapacity.

[1] Note that there is an error in the statement of this theorem in [12]. It incorrectly stated that the value $\max\left(1, \left\lfloor \frac{1}{\epsilon} \right\rfloor\right)$ should be used in the third argument to MinimumCapacity. However, the technical report [13] has corrected the value of the argument to $\left\lceil \frac{1}{\epsilon} \right\rceil$.

Lemma 4 *For a given* Π*, sporadic task system* τ*, and* $k \in \mathbb{N}^+ \cup \{\infty\}$*, if* $t \geq \widetilde{\mathsf{DBF}}(\tau, t, k)$ *for all* $t \in \widetilde{\mathsf{TS}}(\tau, k)$ *and* $U(\tau) \leq 1$*, then the capacity* $\Theta^{\min}$ *returned from* MinimumCapacity (Π, τ, k) *satisfies*

$$\Pi \cdot \frac{n}{p_{\max}} \leq \Theta^{\min} \leq \Pi \tag{18}$$

Proof: Line 1 of MinimumCapacity sets $\Theta^{\min}$ to the minimum default value of $U(\tau) \cdot \Pi$. Thus, $\Theta^{\min} \geq U(\tau) \cdot \Pi$. By Lemma 3, $\Theta^{\min} \geq \Pi \cdot \frac{n}{p_{\max}}$.

To see the upper bound on $\Theta^{\min}$, consider Line 7 when ℓ equals $\lceil \frac{t}{\Pi} \rceil$. The value $\alpha_t \cdot \Pi \leq U(\tau) \cdot \Pi$, by Line 4; $\alpha_t \cdot \Pi \leq \Pi$, since $U(\tau) \leq 1$ by supposition. The value $\frac{D_t - t + (\ell+1)\Pi}{\ell+1}$ is at most Π since $D_t \leq t$. The value $\frac{D_t}{\ell}$ is also at most $\frac{D_t \cdot \Pi}{t}$ since $\ell = \lceil \frac{t}{\Pi} \rceil \geq \frac{t}{\Pi}$; since $t \geq D_t$, $\frac{D_t}{\ell} \leq \Pi$. Finally, the value $\frac{D_t + \alpha_t((\ell+2)\Pi - t)}{\ell + 2\alpha_t}$ is increasing in α_t when ℓ equals $\lceil \frac{t}{\Pi} \rceil$; thus, $\frac{D_t + \alpha_t((\ell+2)\Pi - t)}{\ell + 2\alpha_t} \leq \frac{D_t + (\ell+2)\Pi - t}{\ell+2}$ which is at most Π since $t \geq D_t$. We have, thus, shown that $\Theta^{\min}_\ell \leq \Pi$ when ℓ equals $\lceil \frac{t}{\Pi} \rceil$. Line 8 sets $\Theta^{\min}$ to the minimum value of $\Theta^{\min}_\ell$ for any $t \in \widetilde{\mathsf{TS}}(\tau, k)$. The inequality $\Theta^{\min} \leq \Pi$ follows. ■

The second technical lemma states that MinimumCapacity is monotonically, non-decreasing.

Lemma 5 *For a given* τ *and* $\epsilon > 0$, MinimumCapacity *is monotonically, non-decreasing over* $\{\Pi_{\text{lower}}, \ldots, \Pi_{\text{upper}}\}$.

Proof Sketch: The lemma follows from observing that the values of $\Theta^{\min}$ and $\Theta^{\min}_\ell$ (set in Lines 1 and 7, respectively) are monotonically, non-decreasing in Π. ■

Finally, we give the FPTAS for the MIB-RT with respect to sporadic tasks. The theorem below uses both SelectInterface and MinimumCapacity to obtain the FPTAS.

Theorem 4 *Given sporadic task system* τ *and accuracy parameter* $\epsilon : 0 < \epsilon \leq 1$*, if* $t \geq \widetilde{\mathsf{DBF}}(\tau, t, k)$ *for all* $t \in \widetilde{\mathsf{TS}}(\tau, k)$ *and* $U(\tau) \leq 1$*, then the procedure* SelectInterface$(\tau, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \frac{\epsilon}{3})$ *where* $\mathcal{A}$ *equals* MinimumCapacity $(\cdot, \tau, \lceil \frac{3}{\epsilon} \rceil)$ *returns* $\widehat{\Gamma} = \left(\widehat{\Pi}, \Theta^{\mathcal{A}}_{\min}(\widehat{\Pi}, \tau)\right)$ *such that*

$$\begin{aligned} &\frac{\Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau)}{\Pi^*(\mathsf{OPT}, \tau)} \\ &\leq \frac{\Theta^{\mathcal{A}}_{\min}(\widehat{\Pi}, \tau)}{\widehat{\Pi}} \\ &\leq (1+\epsilon) \cdot \frac{\Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau)}{\Pi^*(\mathsf{OPT}, \tau)}. \end{aligned} \tag{19}$$

Furthermore, the above algorithm has time complexity that is polynomial in the number of tasks n*,* $1/\epsilon$*, and the number of bits required to represent* Π_{upper} *and task system* τ*'s parameters.*

Proof Sketch: Let $\{\Pi_1, \ldots, \Pi_m\}$ be the set of values that Π and Π_{last} are set to throughout execution of SelectInterface$(\tau, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \frac{\epsilon}{3})$. Consider adjacent values $\Pi_i, \Pi_{i+1} \in \{\Pi_1, \ldots, \Pi_m\}$ such that $\Pi_i \leq \Pi^*(\mathsf{OPT}, \tau) \leq \Pi_{i+1}$. Let $\widehat{\Theta}_i$ and $\widehat{\Theta}_{i+1}$ equal the values determined by MinimumCapacity $(\cdot, \tau, \lceil \frac{3}{\epsilon} \rceil)$ evaluated at Π_i and Π_{i+1}, respectively. If $\Pi_{i+1} = \Pi_i + 1$, then $\Pi^*(\mathsf{OPT}, \tau)$ is equal to either Π_i or Π_{i+1}; w.l.o.g., assume that $\Pi^*(\mathsf{OPT}, \tau)$ equals Π_i. By MinimumCapacity's approximation ratio, $\widehat{\Theta}_i \leq \left(1 + \frac{\epsilon}{3}\right) \Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau) \leq (1+\epsilon)\, \Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau)$. Equation 19 follows in this case.

In the case that $\Pi_{i+1} \neq \Pi_i + 1$, we know that MinimumCapacity is monotonically, non-decreasing in Π (by Lemma 5). Thus, the binary search of SelectInterface$(\tau, \mathcal{A}, \Pi_{\text{lower}}, \Pi_{\text{upper}}, \frac{\epsilon}{3})$ ensures that $\widehat{\Theta}_{i+1} \leq \left(1 + \frac{\epsilon}{3}\right) \widehat{\Theta}_i$. The approximation ratio of MinimumCapacity also guarantees that $\widehat{\Theta}_i \leq \left(1 + \frac{\epsilon}{3}\right) \Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau)$. Thus,

$$\begin{aligned} &\widehat{\Theta}_{i+1} \\ &\quad \leq \left(1 + \tfrac{\epsilon}{3}\right)^2 \Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau) \\ &\quad \leq (1+\epsilon)\Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau), \end{aligned}$$

since $\epsilon \leq 1$. The above inequality and $\Pi_{i+1} \geq \Pi^*(\mathsf{OPT}, \tau)$ implies that

$$\frac{\widehat{\Theta}_{i+1}}{\Pi_{i+1}} \leq (1+\epsilon) \frac{\Theta^{\mathsf{OPT}}_{\min}(\Pi^*(\mathsf{OPT}, \tau), \tau)}{\Pi^*(\mathsf{OPT}, \tau)}$$

Equation 19 follows from the fact that $\frac{\widehat{\Theta}_{i+1}}{\Pi_{i+1}} \geq \frac{\Theta^{\mathcal{A}}_{\min}(\widehat{\Pi}, \tau)}{\widehat{\Pi}}$.

The time complexity of the approach follows from Equation 7 of Theorem 1. By Theorem 3, $\chi^{\mathcal{A}}(\tau)$ is $O(n \lg n/\epsilon)$. The second term of Equation 7 ($\lg \left(\frac{\Theta_{\text{upper}}}{\Theta_{\text{lower}}}\right)$) is upper bounded by ($\lg \left(\frac{\Pi_{\text{upper}}}{n/p_{\max}}\right)$) according to Lemma 4; this term is $O\left(\lg(\Pi_{\text{upper}}) + \lg(p_{\max})\right)$. Thus, the entire time complexity of the approach of the Theorem is

$$O\left(n \lg n (\lg^2 \Pi_{\text{upper}} + \lg \Pi_{\text{upper}} \lg p_{\max})/\epsilon^2\right)$$

which is polynomial in the number of tasks, number of bits to represent both the task parameters and Π_{upper}, and $1/\epsilon$. ■

5 Conclusions

In this paper, we propose an approximation algorithm for the minimization of interface bandwidth (MIB-RT) problem in a real-time compositional framework, the periodic resource model. We first propose a general algorithm for determining the interface parameter, given a capacity determination algorithm. This approach is general and can apply to

any component task model. Next, we explore interface selection for components consisting of entirely sporadic tasks, and propose an algorithm based on a previous capacity-determination algorithm [12]. Our algorithm returns bandwidth that is at most a factor of $(1 + \epsilon)$ greater than the optimal minimum bandwidth, for any $\epsilon > 0$. Furthermore, it is shown that our algorithm is an FPTAS as it has time complexity that is polynomial in the number of tasks in the sporadic task system, the number of bits to represent the task parameters, the number of bits to represent the maximum task period Π_{upper}, and the term $1/\epsilon$. Previous work [8] has shown that exact algorithms for MIB-RT problem on periodic resources may require pseudo-polynomial or exponential time. Thus, our results may provide a significant reduction in the time necessary to determine the minimum-bandwidth interface parameters.

In future work, we hope to explore interface selection in the presence of shared global resources; we also would like to study the effect of overheads in the choice of interface parameters. We believe that our the central idea of our FPTAS proposed in this paper is general enough to extend to these more practical and complex settings.

References

[1] K. Albers, F. Bodmann, and F. Slomka. Advanced hierarchical event-stream model. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 211–220, Prague, Czech Republic, July 2008. IEEE Computer Society.

[2] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 187–195, Catania, Sicily, July 2004. IEEE Computer Society Press.

[3] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 95–103, New York, NY, USA, 2004. ACM.

[4] S. Baruah, R. Howell, and L. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[5] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[6] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10190, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Z. Deng and J. Liu. Scheduling real-time applications in an Open environment. In *Proceedings of the Eighteenth Real-Time Systems Symposium*, pages 308–319, San Francisco, CA, December 1997. IEEE Computer Society Press.

[8] A. Easwaran. *Compositional Schedulability Analysis Supporting Associativity, Optimality, Dependency and Concurrency*. PhD thesis, Computer and Information Science, University of Pennsylvania, 2007.

[9] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the IEEE Real-time Systems Symposium*, Tuscon, Arizona, December 2007. IEEE Computer Society.

[10] X. A. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 26–35. IEEE Computer Society, 2002.

[11] N. Fisher. Approximation algorithms for compositional real-time systems: Trading bandwidth for speed-of-analysis. In *Proceedings of the Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Barcelona, Spain, December 2008. IEEE Computer Society Press.

[12] N. Fisher and F. Dewan. Approximate bandwidth allocation for compositional real-time systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, Dublin, Ireland, July 2009. IEEE Computer Society Press.

[13] N. Fisher and F. Dewan. Approximate bandwidth allocation for compositional real-time systems. Technical report, Department of Computer Science, Wayne State University, 2009. Available at `http://www.cs.wayne.edu/~fishern/papers/PRM-Approx-TR.pdf`.

[14] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989*, pages 166–171, Santa Monica, California, USA, Dec. 1989. IEEE Computer Society Press.

[15] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of the EuroMicro Conference on Real-time Systems*, pages 151–160, Porto, Portugal, 2003. IEEE Computer Society.

[16] A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[17] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society, 2003.

[18] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 57–67. IEEE Computer Society, 2004.

[19] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, 7(3), April 2008.

[20] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapur-Tokyo, 2001.

[21] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 80–89, New York, NY, USA, 2005. ACM.

An approach for improving Fault-Tolerance in Automotive Modular Embedded Software*

Caroline Lu[1,2,3]
[1] RENAULT Technocentre
1, Avenue du Golf
78288 Guyancourt Cedex
caroline.lu@renault.com

Jean-Charles Fabre[2,3], Marc-Olivier Killijian[2,3]
[2] CNRS ; LAAS ; 7 avenue du colonel Roche,
F-31077 Toulouse, France;
[3] Université de Toulouse ; UPS, INSA, INP, ISAE ;
{jean-charles.fabre, marco.killijian}@laas.fr

Abstract

Error detection and error recovery mechanism must be carefully selected in automotive embedded applications mainly because of limited resources and economical reasons. However, major safety concerns, brought by new customer services (i.e. chassis control), motivate the automotive industry to search for new means for improving robustness in operation. The challenge is to study a "low-cost", portable and flexible dependability solution. The guiding principle is to rigorously control what/when information is essential to get, and what/when instrumentation is necessary, to perform fault-tolerance. The paper proposes an approach to develop a defense software, as an external customizable component, based on observation and control mechanisms provided by current standard in the automotive industry.

1. Introduction

Improving software fault-tolerance is a common interest for aeronautics, railway and automotive software-based systems. However, the automotive context meets more stringent economical constraints and resources limitations, due to higher volume of vehicle production and lower criticality of vehicle functions compared to avionics. A "lightweight" solution for fault tolerance is studied.

To optimize online verification means to avoid exhaustive systematic information storage and checks, thanks to preliminary safety analyses. Identifying, at first, major critical data and control flows of application software enables to perform selective verification. The drawback of such an application-specific approach would be a lack of adaptability and portability for reuse of safety mechanisms, if they are not well organized and coordinated.

Our approach favors reuse by applying the "separation of concerns" principle [1] to realize customizable defense software. The defense software, implements a fault-tolerance strategy and is separated from functional software. Both software parts interact with each other only through an instrumentation interface. The error monitoring strategy is application-specific and derived from safety analysis, whereas the instrumentation interface between functional and defense software is as generic as possible. This way, if functional software evolves, the interface may evolve but the strategy and defense software remain unchanged. On the other hand, if defense software has to evolve due to arbitrary change (addition or removal of a strategy) the interface may be adapted, without any change on functional software. Feasibility of the presented framework and robustness improvements has been experimented with several prototypes. Efficiency of the defense software is evaluated by fault injection.

The aim of the work reported in this paper is to present the approach and the principle of the fault-tolerance framework. It addresses particularly interaction errors between application software and lower software layers. Section 2 precisely describes the context of the work, in terms of automotive software architecture, fault model and safety requirements. The fault-tolerance framework and a corresponding development cycle are proposed in Section 3. The architectural solution is discussed according to two main aspects: design of defense software (Section 4) and instrumentation interface (Section 5). Finally, Section 6 gives an idea of early implementation issues.

2. Automotive software context

New major standards are emerging in the automotive landscape and will probably influence

* This work has been partially supported by the SCARLET project financed by ANR (the French science foundation, ground transportation research programme PREDIT) focused on robustness of executive software in critical automotive applications.

dependability of tomorrow's embedded software. The first one, AUTOSAR [2], standardizes complex automotive software, structuring it in modules and abstraction levels. We focus particularly on the interaction between application and basic software modules. Another standard, ISO-26262 [3], aims at promoting functional safety measures, at each step of the development cycle of a product.

2.1. Fault model

An automotive embedded system may fail in operation due to either physical faults (hardware, EMC, etc.) or residual bugs from design or development phase of the software development process [4].

Physical faults are modeled as permanent and transient bit-flips and stuck-at in the code and data memory segments. This kind of fault is always possible due to the aggressive environment of automotive applications and the increasing complexity of the hardware components and system architecture.

Bugs during design may occur due to non respected rules for design (MISRA), bad temporal design (sizing, execution order, etc.), bad resource sizing, bad data usage (wrong choice of data for usage, wrong handling of a data, etc.) or non expected modes. Bugs at development phase are likely to happen during manual coding, because of misinterpretation of specifications, coding errors, compiler or linker's default.

A new growing trend is automated code generation. Then scaling or configuration of tools may be wrong (it is enhanced by software complexity). The adaptation to the generator's constraints may be uncomplete (e.g. multiplication of boolean values is not optimal and not supported by all generators). Generator's defaults (especially if the tool is not certified) may lead to errors on the generated code. The integration phase takes a major role in the context of component-based systems and the use of *Components-Of The-Shelf* (COTS) as black boxes most of the time. At the integrator level, there may be again misinterpretation of specifications, coding errors (of glue code), bad scaling of global data, use of bad module version or configuration, and compiler or linker's default.

Actually, the statistical distribution of fault and their diversity are not the major interests from the application software viewpoint. All these faults result in transient or permanent failures on the functional data and control flow. Application-level faults are easier to translate into customer effect, and can be evaluated depending on levels of potential threat or undesirable event to people.

2.2. Automotive safety constraints

A given automotive embedded system is described by a set of specification documents and/or models. They are derived from functional and mechatronics levels to application software design requirements. Safety analysis identifies a list of "unwanted system events (USE)" at application software level. These USE can be potentially safety-critical or not. In the first case, the customer may be endangered, whereas in the second one leads at worst to a dissatisfaction of the customer. Safety barrier must be designed to avoid both these types of unwanted events.

The selection of safety properties from these specifications is defined case-by-case for a given project, taking into account economical, hardware and software sizing constraints.

About potentially critical unwanted system events, the ISO26262 standard defines four safety levels called ASIL (Automotive Safety Integrity Level). They are graded from A to D level with a respectively increasing criticality. Each level is given a set of requirements within which safety methods and mechanisms are listed with graduated recommendation. Therefore, the proposed protection framework enables focusing on highly critical (ASIL C-D) functions and/or information only.

3. Framework overview

The proposed fault-tolerant architecture relies on "computational reflection" [5]. Basic concepts and the overall methodology are given before describing the defense software design and implementation.

3.1. Reflective Principle

The reflection paradigm [6] for fault-tolerance purpose relies on the ability of a system to check and to correct itself in a separate abstraction level. In practical terms, the software architecture (Figure 1) is clearly divided into two parts (functional and defense software) that interact together via an interface [7, 8]. The defense software has enough knowledge of the structure and expected behavior of functional software, to control it. To apply this principle to a given functional software, the main activities concern the definition of:

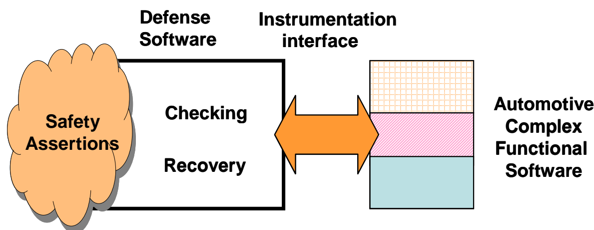
- Safety Assertions (Sections 3.5);
- Defense software (Section 4);
- Instrumentation interface (Section 5).

The fault-tolerant architecture corresponds to the defense software and the instrumentation interface. The defense software detects errors by checking safety properties and performs recovery using generic instrumentation and infrastructure functionalities.

The idea is similar to other industrial solutions to improve system robustness and safety in railways and aeronautics applications. In the electronic interlocking system Elektra [9] a two-channel-approach (notion of safety bag) performing specification diversity is used

for detecting software design faults. Airbus command and control systems rely on the notion of self-checking component composed of command and monitoring computers, in the series A320 to A380 [10]. However, such architectural solutions are not viable for the automotive industry for the time being, due to strong constraints on resources. A lightweight solution may be less robust than these systems.

Figure 1. Reflective System.



3.2. Framework

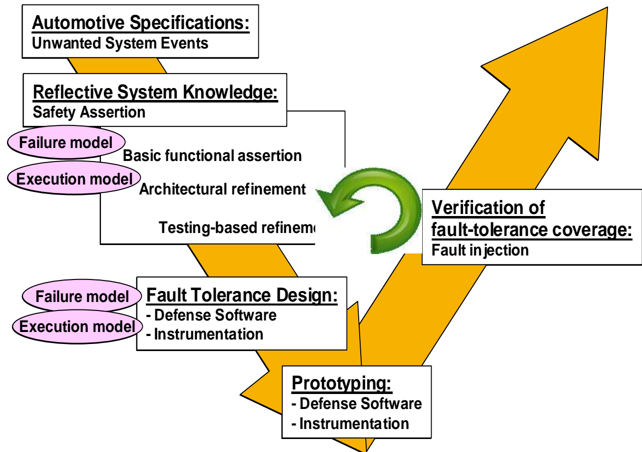
Following the reflective principle, fault-tolerance relies on the knowledge (a model) a system has of itself and safety properties. The accuracy of the knowledge determines the ability of the system to control its state and behavior. This is why a few refinement steps, using a top-down approach, may be necessary to improve fault tolerance.

Figure 2 describes the main parts of the overall framework. Automotive unwanted system events are translated into basic safety assertions, according to a functional "failure model", described in Section 3.3. At this stage, fault-tolerance is still designed at the application software level, ignoring the execution support. To deal with the real embedded system, a first refinement step of the safety assertions aims to take into account the software architecture and underlying infrastructure. It relies on a simple "execution model" (Section 3.4) of the system in operation.

Then, for each safety assertion, the corresponding defense software and instrumentation can be defined and implemented (Sections 4 and 5). In our approach, verification of fault-tolerance coverage is performed by fault injection. Depending on the results, a new refinement of safety assertions can be carried out, and fault-tolerance software design adapted accordingly. The process can be repeated iteratively until the expected fault-tolerance coverage is reached.

The next sections introduce briefly the two concepts of "failure" and "execution" models that are used, as a support to face the diversity of automotive applications. The "failure model" is an input of the definition of fault tolerance mechanisms. Regarding implementation, an "execution model" is essential to analyze the impact of faults and related errors on the system behavior. For this purpose, we do not need complex formalisms. We just need a simplified representation of complex systems, highlighting specific concerns. From this model, we can factorize automotive safety needs into a limited number of categories, for which generic protection mechanisms can be defined and developed.

Figure 2. Overall framework.



3.3. Functional failure model

At application software level, we structure the failure model into two parts: data flow and control flow. Failures can impact both data and control: data often accompany the control or when data moves, control can be activated. The considered classification is not orthogonal. From the control flow viewpoint, the first question is the internal scheduling of computation steps within a software component, whereas data flow mainly means reasoning about data properties, availability, transformation and latency. The user can arbitrarily identify a data failure, a control failure or both, depending on his major concern.

Critical control flow failures.

They fall into 3 categories. The first one targets ***control events***, which impact directly or indirectly, the activation or termination of execution of a treatment. Another type is a defect on the ***sequence of execution***, either at the application level or at lower levels. The last category of control failure affects the ***execution time*** (deadline, timing evolution, periodicity, etc.).

Critical data flow failures.

Value and timing defects can be separated. Data include variables or exchanged messages, as inputs or outputs of software modules. A ***value*** may be faulty within a correct range or out of range. We may also have complex requirements on the values of a set of data (symbolic expression or equation). If functional ***timing*** constraints are explicitly given on data, we relate them to data communication instead of time of execution.

3.4. Execution model

The difficulty to combine a classical state-transition graph with the considered failure model, to highlight potential error sources, leads to introduce a dedicated simple representation. The objective is to describe the runtime behavior of a system as a sequence of "scheduled entity". The "scheduled entities" are triggered by events and generate triggering events. They are also data consumers and producers.

The "scheduled entity" generic expression gathers two viewpoints. For the operating system, "scheduled entities" correspond to "tasks". However, for the designer of the application, implementation issues, including tasks mapping, are generally unknown. For example, if applications are developed originally with Simulink tools, functional requirements specify the sequence of execution of "application level" connected boxes. Such "application level" boxes of the Simulink model become application-level functions after code generation. Consequently, these application level functions are also considered as "scheduled entities". The mapping of a function within a task is the job of the integrator.

Control flow of a scheduled entity relates to the control events starting or stopping its execution. These events are produced by the environment or other entities. In parallel, *data flow* of a scheduled entity corresponds to the input data it consumes and the output data it produces during its execution. Interactions (and error propagation sources) through the software architecture at runtime are therefore based on data exchange and control events, and potential interweaving of scheduled entities.

The granularity level of the "failure" and "execution" models enables to deal with different automotive applications, from air-conditioning to torque control modules.

3.5. Design steps and refinement process

An example is given now, showing the way a real automotive unwanted system events is used in order to define the "reflective system knowledge" (Figure 2).

Basic functional assertion example.
Preliminary safety analysis of an automotive system identifies a list of "unwanted system events (USE)" (Section 2.3). These USE are the requirements, so to speak initial or basic functional assertions. For example, a realistic USE on an automotive transmission module could be the following:

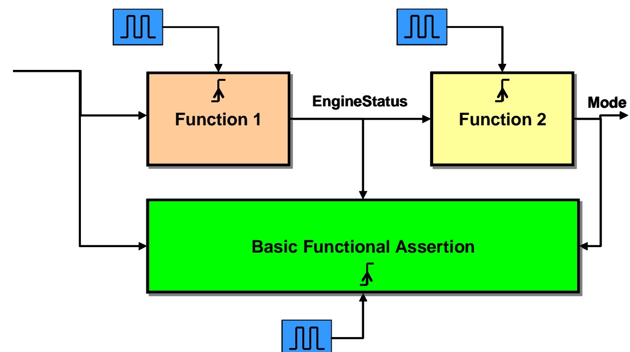
> *"The system is blocked (more than 1 second) in mode A, while the engine status is equal to 2, whereas it should switch to mode B".*

Figure 3 shows a Simulink-like model of the automotive application ("Function1", "Function2"), which is targeted by the USE. Other functions, inputs and outputs of the real system have been hidden to keep the example simple. In the automotive context, "Function1" belongs to the static control module that interprets driver commands and environment measures. "Function2" is a part of the dynamic system control that computes the engine torque set-point. Figure 3 also shows the defense software that implements the "basic functional assertion" to be verified and that receives application critical data as inputs.

At a coarse abstraction level, our defense software is limited to a module that verifies the basic assertion, and is eventually able to switch the system in a predefined degraded mode. To perform the verification, it is mandatory to dive into low-level details to solve the following issues:

1) how/when to catch and store the information required to perform the verification (run the executable assertions with all parameters fixed).
2) how/when to perform the verification within the control flow of the system (thanks to the execution model) and when triggering the error recovery.

Figure 3. Example at functional level.



In this example, the considered type of fault is either the loss of a control event or a wrong data event. Both types of fault may lead to a miss of the change of mode. To derive the executable assertion from such high-level analysis, a refinement taking into account the underlying software architecture is necessary.

Architectural refinement example.
To perform the refinement, low-level implementation details are needed, either from underlying executive layers or from the communication services

For the given example, the input event of "Function1" (Figure 4) is implemented by the value recorded by a hardware sensor. It is transmitted to "Function1" every 10ms by a periodic task, reading the value of the sensor. The output of "Function2", called "mode" must be consistent with the sensor value and

the engine status, according to the USE. An error may happen when a corrupted data is read by the sensor.

After a careful analysis of the way these functions have been implemented (i.e. mapped to OS or middleware objects, connected to each other and to the external world, the execution profile and which core parameters have been considered), a refined version of the assertion can be expressed, for instance:

> *"At the end of each sensor task, the mode is consistent with the value of the sensor, while the engine status is equal to 2".*

This refinement process enables the initial USE related assertion to be expressed in computing terms. It is worth noting that the final assertion depends on the implementation of the functions. It enables to identify:

- the required information to perform the check (e.g. sensor value, mode, engine status);
- the logical and/or numerical expression or the algorithm corresponding to perform the check ;
- where the check of the assertion has to be performed in the execution flow (e.g. end of sensor task);

Refinement evaluation.
After prototyping the defense software for the selected assertions, an evaluation phase can be started. We perform verifications by a fault injection technique, based on (i) the considered failures (cf. failure model) and (ii) the USE. Control flow failures are realized by the insertion of system calls in the program, whereas data flow is disturbed by selected communication service calls.

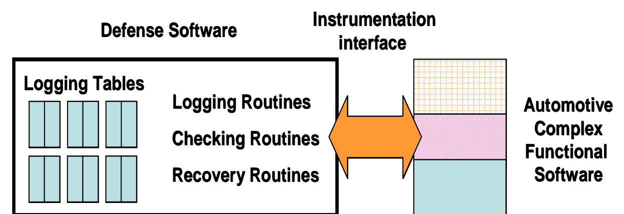
If the fault-tolerance coverage rate and residual failure modes do not match the expected results, then a refinement is needed. This involves diving into deeper analysis of the implementation, deeper knowledge of the software architecture or revising some steps in the assertion refinement process.

Finally, the refined assertions can be included into the defense software. This is done naturally since the defense software architecture is adaptable by construction thanks to the reflective approach. Then the "refinement-evaluation process" starts again until the expected fault-tolerance coverage rate is obtained.

4. Defense Software

The defense software (Figure 4) is organized around logging tables and three types of services that control (i) information logging ("*logging routines*"), (ii) error detection ("*checking routines*"), and (iii) error recovery ("*recovery routines*"). This application-related part of the fault-tolerant architecture is specified from a given set of selected safety properties.

Figure 4. Defense software organization.



4.1. Logging strategy and architecture

Logging or tracing are mechanisms often needed for debugging and diagnosis issues. Amount of tracing information depends on the objectives of the user: defect analysis with possibility to reproduce the failure scenario (extended tracing), defect analysis only to remove a bug (local tracing), performance profiling to determine where the system spends its execution time (selective tracing), etc.

The capture of software trace induces significant execution slowdown. Classical automotive applications (e.g. Body Control Module), within an ECU, may exchange several thousand of data, and may be controlled by several dozens of tasks that include both application and infrastructural tasks. Reducing overhead requires either sacrificing details or using hardware extensions.

As a result, the logging strategy has to select rigorously the *necessary and sufficient* critical information to get at runtime, according to fault-tolerance concerns. Then, the structure of storage is a major factor to reduce timing access to information for online error detection. Actually, software architecture for logging must be specified to favor reuse, adaptability with the diversity of automotive applications, and portability on different platforms.

The proposed logging strategy is derived from the model of execution from section 2. In order to supervise data and control flow, the system should record a history of task switches, application-level functions entries and exits, some system calls and data communication. Then, information logging consists in storing only events that belong to a critical flow. This preliminary selection (resulting in less than a hundred of critical data and a few critical tasks, for example) enables stored information to be redundant and diversified. It provides multiple viewpoints, like OS/application or data/control flow, that are of high interest for improving fault-tolerance. Instrumentation to catch information is detailed in section 5.

The logging architecture is organized into several "**bracket-tables**" that are updated and used at runtime. Any storage table can be considered as an opening or closing "bracket". Each logging table has an associated table because information they store is symmetric. For

instance, when information regarding the start of a task is stored in a table ("opening bracket"), the information regarding the end of the same task is stored in an associated table ("closing bracket"). The number of tables depends on the number and the complexity of safety properties to be protected by the defense software. Tables should be kept small to reduce scanning of information by error detection routines. The "bracket-tables" are sorted into 4 categories:

- ***The execution trace from OS viewpoint:*** when a critical task starts execution, an "opening-bracket-table" entry is filled basically with the task identifier and a timestamp. The "closing-bracket-table" stores the same type of information, when the task ends (not when it is preempted). The length of the table depends on the complexity of the safety properties. For example, if a periodical sequence of execution must be verified, the length is that of the sequence. This category contains at most one couple of bracket-table (opening/closing).
- ***The execution trace from application viewpoint:*** when a critical application-level function starts, an "opening-bracket-table" entry is filled basically with the function identifier, the task identifier in which the function runs and a timestamp. The associated "closing-bracket-table" stores the same type of information, when the function ends (not when it is preempted). As above, the length of the table depends on safety properties. This category contains at most one couple of bracket-table that may replace (if functions are considered more meaningful than tasks) or be redundant with the preceding table providing the OS viewpoint.
- ***The control event trace:*** when an activation event (that impacts directly or indirectly the activation of a task) happens, an "opening-bracket-table" entry is filled basically with parameters that characterize the event, the current running task identifier, and a timestamp. The "closing-bracket-table" stores the same type of information, when the termination event occurs. This category may group several bracket-tables because there are several types of control events.
- ***The data event trace:*** when a critical data is written, an "opening-bracket-table" entry is filled basically with the data, the function identifier that produces the data, the task identifier in which it runs and a timestamp. The "closing-bracket-table" stores the same type of information, when the data is read. This category may group several bracket-tables because there are several types of data communication services.

Each table is associated with a dedicated routine ("logging routine") that uses preferably existing infrastructure services to get information ("basic sensor services", Section 5.2). When these services are not available, data are collected through the routine parameters, or by additional instrumentation ("hooks", Section 5.1).

4.2. Error detection strategy

Once application-specific safety properties are specified, the corresponding error detection routine is developed as an executable assertion. An assertion is verified at runtime within a corresponding "checking routine". When an error is detected, the checking routine triggers a "recovery routine" (Section 4.3).

Safety properties may address critical data flow, control flow or both. The knowledge of defense software, about the behavior of functional software, and more particularly the critical flows, is gathered into the logging tables (that must be trusted). Checking routines rely on the information stored in these tables to verify assertions. The structure of logging tables is given in section 4.1. The content of tables is equal to the information that is needed to check safety assertions and recover from errors. As a result, this content (which event, when, how many) is determined from the safety assertions.

Assertions are pre- or post- condition at a verification point. The verification point depends on the safety property and if error detection "as soon as possible" is expected or not. For example, it can be the beginning of execution, a waiting point within a task, or the reception or emission of a control event, etc.

According to 3.3 that describe the main types of failures, the corresponding types of assertions address 1) control events, 2) sequences of execution, 3) timing constraints of execution, and 4) values or 5) timing constraints on data exchange.

Table 1. Basic reference to logging tables.

Assertion with:	Logging tables
Control event	Control event trace
Sequence of execution	Execution trace
Timing constraints of execution	Execution trace
Value constraints on data	Data event trace
Timing constraints on data	Data event trace

The analysis of an example gives an idea of how to derive the checking routine from an assertion such as:

> *"The acknowledgement of reception of Message 1, notified to Task 1, at latest 2ms after Message 1 has been sent, allows Task1 to activate Task 2, else Task 3 must be activated".*

"The acknowledgement of reception of Message 1" and the "activation of Task 2" are control events. "At

latest 2ms after Message 1 has been sent" is a timing constraint on data. The pseudo-code of the checking routine (called before the execution triggering of Task2) for this assertion is given in Figure 6.

Figure 6. Checking routine.

```
Check_P1 {
If {
/* Check in the control event trace of "ActivateTask" system call to find activation of Task 2 by Task1 */
        LogTable_ActivateTask.ActivatedTaskID[i] == Task 2;
        LogTable_ActivateTask.RunningTaskID[i] = = Task 1;
        LogTable_ActivateTask.Return[i] = = ok;
/* Check in the control event trace of "DataAcknowledgement" service call to find notification of reception of Message 1 to Task 1 */
        LogTable_DataAck.TaskID[j] = = Task 1;
        LogTable_DataAck.MessageID [j] = = Message 1;
        LogTable_DataAck.Return[k] = = ok;
/* Check in the date event trace of "DataSent" service call to find time of emission of Message 1 by Task 1 */
        LogTable_DataSent.TaskID[k] = = Task 1;
        LogTable_DataSent.MessageID [k] = = Message 1;
        T1= LogTable_DataSent.Time[k];
/* Check the timing contraint */
        T2 = LogTable_DataAck.Time[j];
        T1 – T2 < 2ms;
}
/* An error is detected */
Else    Recov_P1( );
}
```

4.3. Error recovery strategy

Error recovery from application and from infrastructure viewpoints have complementary advantages and limitations. To reduce error latency and improve efficiency the core idea is to use infrastructure recovery controlled by application level consideration.

Degraded modes of automotive applications are generally very sophisticated at the system and application level. Once an error is detected the application is turned into a safe state. About data flow, degraded data is usually known to recover from invalid values. Missed timeout or acknowledgement of data exchange may lead to new communication requests or use of degraded values again. About control flow, apart from reset, recovery actions are limited, it consist in a change of working mode or application-level functions inhibition.

At the infrastructure level, recovery actions on control flow are also basic: reset, terminate and restart a task or a set of OS objects. Recovery actions are difficult to take without knowledge of the application. Killing and restarting an air conditioning, an airbag or a torque control module has not same impact. From the application level, the support of execution is not supposed to take alone uncontrolled recovery decisions that could leave the system in an unexpected state. It is worth noting that infrastructure services represent a collection of software actuator (Section 5.3), which can improve fault-tolerance.

In the proposed fault-tolerant architecture, each "checking routine" is associated with one or more "recovery routines". Recovery routines are calling available executive services ("basic actuator services", Section 5.3) and update logging tables, if necessary.

The recovery action depends on the detected error. Going back to the example given in section 4.2, an example of pseudo-code of the recovery routine would be:

```
Recov_P1 { /* Error: Task 2 must not be activated but Task 3 */
ActivateTask (Task3); }
```

Control flow error recovery.
If a lost control event has been detected, the logging tables should have stored the event so that we can chain the correct treatment (activate a task, wake up a waiting task, etc). Another option is to duplicate the event and make sure it is received. On the contrary, if a wrong or untimely control event has been detected, it should be cleared.

If an error in the sequence of execution is detected, two types of recovery can be considered. Usually, we can terminate or chain a task, to restart another task within a degraded mode, or the same task. Otherwise, the whole application software component can be stopped, reinitialized and restarted.

Data flow error recovery.
In case of data flow error, a first option is to update data values, either with a good value if the error detection part managed to save the correct value, or with old/default value. If timeout data reception or acknowledgement data emission is missed, logging routines could have saved the exchanged value and the recovery restores communication, or it repeats the communication call.

5. Instrumentation

Two types of software instrumentation are considered (Figure 6): hooks and basic services. Hooks are the means to tie up defense software to functional software, and to insert code. The possibility to generate hooks automatically is an advantage regarding development cost. Basic services play the role of software sensors and actuators. The availability and authorization of their use enables to limit intrusiveness especially to get information. All fault-tolerance

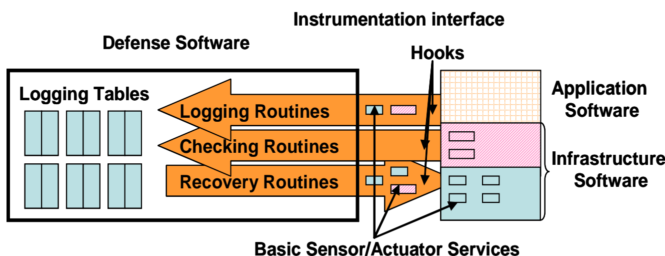
intelligence and control is contained in defense software described before. Now, instrumentation, as tools, intends to be generic, flexible and reusable.

5.1. Hooks

In C programming, hooks are entry points, with empty routines, located at selected places in the program. They are commonly used as debugging breaking points or exception treatments triggering. In the AUTOSAR OS specification [2], some hook functions are defined and implemented by the user. The operating system invokes them at specified times, such as tasks context switch, startup, shutdown, or detected errors.

These hooks are very convenient for "logging", "checking" or "recovery" routines belonging to the defense software. The insertion of a hook, at a selected place in the source code, is related to both a place in the architecture of the software system and a moment of execution at runtime.

Figure 6. Instrumentation organization.



Hooks must be placed on critical data and control flow. Considering the 4 types of traces (section 4.1), recording the execution from OS and application viewpoints, can be done with hooks set at the beginning and at the end of execution of critical tasks and application-level functions. Hooks are also set at the corresponding critical services calls to capture the control and data event traces.

The instant of execution to trigger the "checking routine" and thus the location of the hook, is crucial. For the "logging routine", the location of the hook is set where information is easier to capture.

Recovery routines are essentially triggered by checking routines, immediately after error detection. Nevertheless, an application can specify that after error detection the recovery has to be delayed to the end of task execution for example. In this case, a hook at the end of the considered task contains the recovery routine, which is activated only if quoted by the corresponding checking routine.

Several implementations of automatically generated hooks can be considered, especially "at" service calls. Hooks can be inserted just before or after the call instruction. Another solution is to add them within the service routine, at the beginning or the end. The difference is important with system calls, if the system supports the separation between user and supervisor mode. In the first case, the hook routine runs in user mode, whereas in the second case, it runs in kernel mode and has access to more information if needed (task priority, etc.).

Another implementation issue is the use of parameters or not at the hook interface. It is an alternative to the use of sensor services. For example, after a write-service, the data value may be collect as an input parameter of a hook, instead of using a read-service to get the value.

5.2. Basic sensor services

Types of information that contribute to describe execution, control event and data event traces are:

- ***Exchanged parameters (what):*** return notification, activated task, set event, activated alarm, exchanged data, etc.
- ***Current execution context (where):*** application-level function identifier, task identifiers (task state, priority, etc. if needed) or interrupt routine identifier, current mode, etc.
- ***Timestamp (when):*** e.g. counter register value

Executive support should give the possibility to the user to get this information through an observation interface. For example, via OSEK-VDX operating system standard interface [11, 2, 12], some information is reachable: the running task identifier ("GetTaskID"), the task state ("GetTaskState"), the current state of event mask of a task ("GetEvent"), alarm characteristics ("GetAlarmBase", "GetAlarm"), and the current mode ("GetActiveApplicationMode"). Autosar OS that can be considered as an extension of OSEK, has additional standardized interfaces: "GetISRID" to get the identifier of interrupt routines, "GetApplicationID" to get the identifier of a sort of partition (if the OS uses memory protection), and information about predefined scheduling tables ("GetScheduleTableStatus", "GetCounterValue", "GetElapsedCounterValue").

The observation interface of the Autosar operating system is rich enough, if the user does not need to check the task priority. What is missing, at higher level, is essentially an identifier for application-level functions, which is added manually otherwise.

5.3. Basic actuator services

Basic actuator services can be defined independently from a particular implementation, even if the reference studied architecture is that of Autosar standard. Practically, functional infrastructural services are used (Table 2), even if they are not designed to perform recovery. Ideally, a specialized recovery interface should be provided by the infrastructure and should be well controlled by the user. Referring to the

model of execution in section 2, actuator services can be structured into control actions and data actions.

Control flow actuators.
At the operating system level, actions on control flow concern the life cycle of tasks and can be classified into 3 categories:

- ***End of task execution:*** the idea is to terminate the erroneous current treatment.
- ***Start of task execution:*** the objective is for example to launch a degraded task if switch to degraded mode is decided; or to launch the expected task after error detection on sequence of execution; or else to launch again the same task from the beginning to re-execute the same treatment with right entries. The activation of a task may be synchronous or asynchronous.
- ***Suspension of task execution:*** the idea is to temporarily stop the current execution, to allow the execution of another action/task.

Table 2. Recovery actions with AUTOSAR.

Recovery action	Useful Autosar services
End of task execution	TerminateApplication, TerminateTask, ChainTask, CancelAlarm
Start of task execution	ActivateTask, ChainTask, RestartTask (with TerminateApplication), SetEvent, SetRelAlarm, SetAbsAlarm
Hang of task execution	- (difficult with a static priority based scheduling)
Production of data	Rte_Write, Rte_IWrite, Rte_IrvWrite, Rte_IrvIwrite, Rte_Send
Consumption of data	Rte_Read, Rte_IRead, Rte_IrvRead, Rte_IrvRead, Rte_Receive
Renewal of data request	Rte_Send, Rte_Call
Inhibition of data	- (no direct means)

Data flow actuators.
At the communication level, actions on *data flow* relate to actions on data value and on data timing occurrence:

- ***Production of correct or degraded data:*** the recovery strategy overwrites the preceding erroneous data, by the right one.
- ***Consumption of correct or degraded data:*** data consumption instruction is called another time to get the correct value which is has been updated by the recovery strategy.
- ***Renewal of data request:*** data production or consumption instruction is called another time, when timeout reception or acknowledgement of emission is missed.
- ***Inhibition or delay of data:*** when invalid or untimely data is received, the recovery strategy acts as a filter, to transmit only right data to the application.

The following section refines the description of defense software and instrumentation in the context of memory protection with kernel and user separation of modes and address space.

5.4. Protection of defense software

The protection of defense software is principally a matter of economical constraints. The more measures are taken to improve defense software, the more it is expensive. A "low-cost" solution is required, although all the proposed fault tolerance strategy relies on the robustness of defense software. The only design property of defense software that satisfies both opposite requirements is: the complexity of the defense software is considered much lower to that of the functional software by construction. Concerning enhanced validation process and hardware protection, it will depend case by case on available resources that are given to particular projects.

Enhanced validation process.
A rigorous development process, including verification methods, has to be performed. We use fault injection techniques (Section 3.5) to measure fault-tolerance coverage, and to detect remaining software errors of defense software.

When defense software is based on safety assertions that have a complex behavior (check of transitions that implies many data and control elements), the use of a formal language to implement these routines is to be considerate. Again, some automotive projects may not take this option for culture or economical reasons. In our work, we use the C language, respecting MISRA coding rules [12].

Hardware protection.
To strictly follow the principle of separation of functional and safety concerns promoted by the reflective approach, both software part should be spatially and timely separated. Taking the example of Elektra railway system [9], three processors operate functional services and three other processors supervise the functional part, in parallel. So many resources are still unaffordable in the automotive world. Instead, software redundant information logging is realistic, in the proposed architecture, if other resources and timing constraints are respected.

Simple separation of functional and defense software can be done by the use of hardware memory protection. Considering this particular context, hooks can be implemented in user space, for convenience of existing automatic code generation of hooks. The logging tables are the most critical data, so they must be stored in protected address space, separated from

functional part. Logging, checking and recovery routines, with the software sensor and actuator they contain, have to access the logging tables by reading or writing, so they also must be trustable. As these routines are called within hooks in user mode, that requires a switch from user mode to kernel mode.

6. Early implementation issues

We have developed several AUTOSAR software platforms, both on a virtual processor running on an UNIX machine and on a real embedded evaluation board. We use a Freescale evaluation board S12XEP100TM 16 bit microcontroller, with memory protection unit, and another S12XDP512TM, without memory protection. Our development environment is CodeWarriorTM from Freescale.

The AUTOSAR RTE is automatically generated by a software tool from Vector (Microsar RTE, DaVinci DeveloperTM 2.2). We worked both on several application components we synthesized, and on serial automotive software products we adapted to the AUTOSAR context. The safety properties we take as inputs are derived from real automotive requirements.

We use Trampoline [13], an open source operating system from IRCCYN, compliant to AUTOSAR OS.

Our current experiments show the feasibility of the approach to improve robustness on prototypes. We have compared protected and non-protected applications with similar hardware, by carrying out verification testing, using controlled fault injections that cause USE (Unwanted System Events). Protected applications perform fault tolerance of their failure model. However, the evaluation of robustness should be completed by comparison with other fault-tolerance solutions.

7. Conclusion

The automotive industry is facing increasing complexity of embedded software, error propagation and the need to meet robustness challenges, in spite of stringent economical constraints. As a representative context of tomorrow's automotive software, we chose to deal with the two emergent standards: AUTOSAR for modular multilayered software architecture, and ISO26262 about safety concerns.

The work reported in this paper shows an approach to develop customizable defense software, externally to the target system. The proposed fault-tolerant architecture is based on the classical separation of the functional implementation and that of the safety functions, using the interfaces (entry points) defined by AUTOSAR.

This approach is very attractive for the automotive industry since it enables to tailor defense mechanisms according to the needs on a case-by-case basis.

Feasibility study has been carried out on early implementations of synthetic AUTOSAR applications. Current work exemplifies in deep error detection and recovery mechanisms and focus on fault injection to evaluate the efficiency of the approach.

References

[1] C. Lopes, W. Hursch. "Separation of Concerns". Technical Report, College of Computer Science, Northeastern University, Boston, USA, Feb 1995.

[2] AUTomotive Open Standard ARchitecture, http://www.autosar.org

[3] ISO/CD 26262-6, "Road vehicles, Functional safety, Part 6: Product development: software level", 2008.

[4] R. Chillarege, IS. Bhandari, JK. Chaar, MJ. Halliday, DS. Moebus, BK. Ray, and MY. Wong, "Orthogonal defect classification-a concept for in-process measurements", *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[5] P. Maes, "Concepts and Experiments in Computational Reflection". *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, pp. 147-155, 1987.

[6] J. Voas, "A Defensive Approach to Certifying COTS Software", *Reliable Software Technologies Corporation*, Technical Report: RSTR-002-97-002.01, 1997.

[7] M. Rodriguez, J.C. Fabre, J. Arlat, "Wrapping real-time systems from temporal logic specifications". *European Dependable Computing Conference (EDCC-4, 2002)*, Toulouse (F), pp. 253-270, 2002.

[8] F. Taiani, J.C. Fabre, M.O. Killijian, "Towards Implementing Multi-Layer Reflection for Fault-Tolerance". *IEEE International Conference on Dependable Systems and Networks (DSN'2003),* San Francisco (CA, USA), pp. 435-444, 2003.

[9] H. Kantz, C. Koza, "The ELEKTRA railway Signaling-System: Field Experience with an Actively Replicated System with Diversity". Alcatel Austria AG., Vienna, Austria, 1995.

[10] P. Traverse, I. Lacaze, J. Souyris, "Airbus fly-by-wire: A total approach to dependability", 2004.

[11] "OSEK/VDX Operating system". *Technical report*, 2005.

[12] The Motor Industry Software Reliability Association, http://www.misra.org.uk

[13] J.L. Béchennec, M. Briday, S. Faucou, Y. Trinquet, "Trampoline : An Open Source Implementation of the OSEK/VDX RTOS Specification", *IEEE Int. Conf. on Emerging Technologies & Factory Automation (ETFA'2006)*, Prague, Czech Republic. pp. 62--69 (2006) – see : http://trampoline.rts-software.org –

A Data Oriented Approach for Real-Time Systems

Tanguy Le Berre, Philippe Mauran, Gérard Padiou, Philippe Quéinnec
Université de Toulouse - IRIT
2, rue Charles Camichel
31000 TOULOUSE, FRANCE
{tleberre,mauran,padiou,queinnec}@enseeiht.fr

Abstract

Distributed real-time systems often have to maintain the temporal validity of data. In this paper we present a modelling framework centered on data where a so-called observation relation represents and abstracts the interactions between variables. An observation is a relation between variables, an image and its sources, where the image values depend on past values of the sources. The system architecture is seen as a set of observation relations describing the flow of values between variables. The observation relations are parametrized with timed constraints that limit the time shift between the variables and specify the availability of timely sound values.

At this level of abstraction, the designer gives a specification of the system based on timed properties about the timeline of data such as their freshness, latency etc. We proceed to an analysis of the feasibility of such a specification and we formally analyze the correctness of an implementation with respect to a specification.

In order to prove the feasibility of an observation-based model, we build a finite state transition system which is bi-similar to the specification. The existence of an infinite execution in this system proves the feasibility of the specification. Possible implementations are described as a set of interacting components which control the flow of values in the system. A finite system is built to prove the correctness of the implementation by model checking.

1 Introduction

We propose a framework to specify and analyze the timed properties of distributed real-time systems. The architecture of a system is not described as a set of communicating tasks. It is rather described as a set of related variables and links between the values of the variables. The timed requirements of the system are expressed on these links and state that the values of a variable that are available in the system must be "timely valid". A value is valid if it based on values of other variables that are consistent with the environment and the user's requirements. The goal is to express the timed requirements regardless of the task and communication protocols We then check that these requirements are satisfied by the implementation.

This paper presents the formal definitions used to build and analyze our framework. This modelling framework is illustrated by a simple example, an automatic cruise control system. We first describe the underlying formal system. We then introduce the observation relation that is used to describe the architecture of the system as a set of links between the values of the variables. Based on these links, a set of timed properties is defined to specify the timed requirements. A system is specified by the architecture and the timed properties. We explain how the feasibility of the system is proven by using the specification to build a bi-similar finite state transition system which is then explored to search for possible executions. Finally, we show how to model an implementation and check its correctness with respect to the specification. Here also a dedicated state transition system is built.

2 Related Works

A typical approach to real-time systems is the specification of properties as characteristics of the tasks. A scheduling analysis is then performed to check the satisfaction of these properties. In the case of distributed systems, the scheduling analysis takes into account the properties of the communication protocol as in [9].

We depart from such an analysis by expressing the properties as state based properties on the system variables. The properties are not expressed on the tasks and so do not relate system events.

Most works where the properties are specified on the data belong to the field of databases. For example, the variable semantics and their timed validity domains are used in [10] to optimize database transaction scheduling. Our work stands at a higher level as we propose to give an abstract description of the system in terms of of data relations. Another work analyzes the propagation of value in real-time database and their timed correctness [3]. But they only give results as a synchronized set of period tasks. In [8], the authors define derived objects that are computed from a set of objects. The age of an object is defined by the ages of the objects used to compute it. Their goal is to find

a scheduling of a set of periodic preemptable transactions to maintain mutual consistency. They want to limit the dispersion of the ages of the set of objects used to compute a derived object. In this paper we want to check other timed properties such as the freshness of the used objects.

Similar works specify systems using temporal logic. In [2], OCL constraints are used to define the validity domain of variables. A variation of TCTL is used to check the system synchronization and prevent a value from being used out of its validity domain. This work also defines timed constraints on the relations between application variables, but these relations are defined using events such as message sending whereas our definitions are based on the history of the values of the variables.

In [7], an Allen linear temporal logic is proposed to define constraints between intervals during which state variables remain stable. As in our approach, it uses an abstraction of the data timelines in terms of stability intervals. However in [7] the constraints do not relate to real-time.

3 An Introducing Example

We introduce a system example used to illustrate our framework. This system is a simplified automatic car cruise control system. The goal of such a system is to control a vehicle by maintaining a steady speed chosen by the driver. The vehicle speed is controlled through a throttle actuator. A sensor is used to compute the vehicle's current speed and based on this speed and the speed chosen by the driver, the input of the throttle actuator is computed by the control system. The architecture of this simplified system is illustrated Figure 1. This system is a distributed system where the components communicate through a bus.

This system reacts to its environment. The evolution of the vehicle speed implies that each value submitted to the throttle actuator has a bound validity domain. Thus, there are timed requirements on the speed at which the system reacts. We informally give the timed requirements and properties on data and relations between data in such a system:

- the current speed is computed based on the wheel turns. So, a minimum duration between each computation is required to give a relevant speed;
- but the speed must be updated often enough to be consistent with reality;
- there is a minimum time between two updates of the desired speed ;
- due to the bus properties, there is a minimum communication time between the different components;
- the throttle actuator value must be consistent with the current value of the vehicle current speed and the desired speed.

We explain how our approach allows to formally define this system and its real-time properties. Each component uses and/or produces data. We use a relation called observation to specify the dependencies between variables.

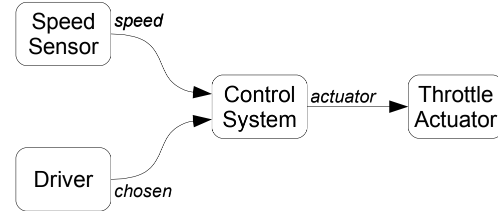


Figure 1. Cruise Control System

4 Formal Background

We give here the formal context and the definition of the properties used to define the observation relation and system timed properties.

4.1 State Transition Systems

Models used in this paper are based on state transition systems. A *transition system* S is a couple $(\Sigma, \rightarrow)$ where Σ is a set of *states* and $\rightarrow$ is a *transition relation*, i.e. a predicate on pair of states. A *state* is an assignment of values to variables. A *step* is a pair of states which satisfy the transition relation. An *execution* σ is any infinite sequence of states $\sigma_0\sigma_1 \ldots \sigma_i \ldots$ such that two consecutive states form a step. We note $\sigma_i \rightarrow \sigma_{i+1}$ the step between the two consecutive states σ_i and σ_{i+1}.

The system properties are expressed as *temporal predicates*. A *temporal predicate* is a predicate on executions; we note $\sigma \models P$ when an execution σ satisfies the predicate P. Such a predicate is written in linear temporal logic. A *state expression* e (in short, an expression) is a formula on variables; the value of e in a state σ_i is noted $e.\sigma_i$. The sequence of values taken by e during an execution σ is noted $e.\sigma$. A *state predicate* is a boolean-valued expression on states.

4.2 Introducing Time

We consider real-time properties of the system data. To distinguish them from (logical) temporal properties, such properties are called *timed* properties. Time is integrated in our transition system in a simple way, as described in [1]. Time is represented by a variable T taking values in an infinite totally ordered set, such as $\mathbb{N}$ or $\mathbb{R}^+$. The time domain is called $\mathbb{T}$. T is an increasing and unbound variable. There is no condition on the density of time and moreover, it makes no difference whether time is continuous or discrete (discussion in [6]). However, as an execution is a sequence of states, the actual sequence of values taken by T during a given execution is necessarily discrete. Note that we explicitly refer to the variable T to study time.

An execution can be seen as a sequence of snapshots of the system, each taken at some instant of time specified by the value of T. We require that "enough" snapshots are performed to catch each computation step. It means that no variable can have different values at the same time and so in the same snapshot. Any change in the system implies time passing.

Definition 1 *Separation. An execution σ is separated iff for any variable x:*

$$\forall\, i, j : T.\sigma_i = T.\sigma_j \Rightarrow x.\sigma_i = x.\sigma_j$$

In the following, we consider only separated executions. This allows to timestamp updates of variables.

4.3 Clocks

Let us consider a totally ordered set of values $\mathcal{D}$, such as $\mathbb{N}$ or $\mathbb{R}^+$. A clock is a (sub-)approximation of a sequence of $\mathcal{D}$ values. We note $[X \rightarrow Y]$ the set of functions with domain X and range contained by Y.

Definition 2 *A clock c is a function in $[\mathcal{D} \rightarrow \mathcal{D}]$ such that:*

- *it never outgrows its argument value:* $\forall\, t \in \mathcal{D} : c(t) \leq t$
- *it is monotonously increasing:* $\forall\, t, t' \in \mathcal{D} : t < t' \Rightarrow c(t) \leq c(t')$
- *it is lively:* $\forall\, t \in \mathcal{D} : \exists\, t' \in \mathcal{D} : c(t') > c(t)$

The predicate $clock(c)$ is true if the function c is a clock.

In the following, clocks are used to characterize the timed behavior of variables. They are defined on the indices of the sequence of states, to express a logical precedence.

4.4 Data Timeline

In order to state properties on the timed behavior of a variable x, we have to be able to characterise its timeline. We introduce a variable that refers to the last time this variable was updated. These are called the update instants $\hat{x}$. The goal is to capture the instant when the current value of x appeared, e.g. the beginning of the current occurrence. This referential can be either explicit or implicit. In the explicit case, the developer is responsible for giving its own variable $\hat{x}$. For example, when a variable is updated in a periodic way. In the implicit case, a formal definition of $\hat{x}$ is given based on the history of the values taken by x.

Definition 3 *For a separated execution σ and a variable x, the update instants of x is:*

$$\forall\, i : \hat{x}.\sigma_i = T.\sigma_{min\{j|\forall k\in[j..i]:\, x.\sigma_i = x.\sigma_k\}}$$

The timeline $\hat{x}$ is built from the history of x values. For a variable x, the update instant of x is defined as the value taken by the time T at the earliest state when the current value appeared and continuously remained unchanged up to the current state.

When x is updated and its value changes then the value of $\hat{x}$ is also updated. Conversely, if $\hat{x}$ changes then the value of x is modified. We consider in this paper that whenever a variable is updated, it is with a new value so that the update instants are equivalent to the modification instants.

The variable d_x is also defined to stand for the duration the current value of x is continuously kept.

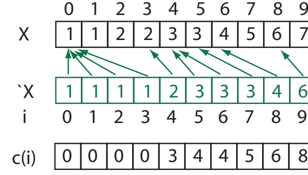


Figure 2. The Observation Relation

Definition 4 *For a separated execution σ and a variable x, the variable d_x is defined by:*

$$\forall\, i : d_x.\sigma_i = T.\sigma_{min\{j|\forall k\in[i..j[:x.\sigma_i=x.\sigma_k\ \wedge\ x.\sigma_i\neq x.\sigma_j\}} - \hat{x}.\sigma_i$$

These two variables give the timed characteristics of the current value of the variable.

5 Modelling the Data Flow

5.1 The Observation Relation

To give properties on the relations binding variables, we define an operator, the observation relation, on state transition systems as in [4]. The observation relation is used to abstract the dependency between values taken by different variables.

More precisely the observation relation binds two variables, the source x and its image y, and denotes that the history of the variable y is a sub-history of the variable x. The relation is defined by one couple $\langle source, image\rangle$ and the existence of at least a clock defining for each state which of the previous values of the source is taken by the image. This clock is used to define the time shift introduced by the observation. Figure 2 shows an example of an observation relation. The definition is:

Definition 5 *The variable y is an observation of the variable x in an execution σ: $\sigma \vDash y \prec x$ iff:*

$$\exists\, c \in [\mathbb{N} \rightarrow \mathbb{N}] : clock(c) \wedge\ \forall\, i : y.\sigma_i = x.\sigma_{c(i)}$$

This relation is used to abstract the communication in a distributed system. We extend this definition to a relation binding an image to a set of variables and a function.

Definition 6 *Given a function f and a set of variables $X = \{x_i | i \in [1..n]\}$, the variable y is an observation of the expression $f(X)$ in execution σ: $\sigma \vDash y \prec f(X)$ iff:*

$$\exists\, c \in [\mathbb{N} \rightarrow \mathbb{N}] : clock(c) \wedge \\ \forall\, i : y.\sigma_i = f(x_1.\sigma_{c(i)}, x_2.\sigma_{c(i)}, \ldots, x_n.\sigma_{c(i)})$$

In this case, all values of the inputs (X) are read at the same time, implying a synchronous behavior. Then the inputs are at the same node or the different nodes have to be perfectly synchronized. If they are not, additional observation relations are added to model the communication and the copy of the input to the computation node. With this definition, the basic observation is just a special case where $f = Identity$ and $card(X) = 1$.

Thus the observation can be used as an abstraction of communication in a distributed system as well as an abstraction of a computation:

```
- communications:
```

$$`speed \prec speed$$
$$`chosen \prec chosen$$
$$`actuator \prec actuator$$

```
- computation:
```

$$actuator \prec control(`speed, `chosen)$$

Figure 3. System Architecture

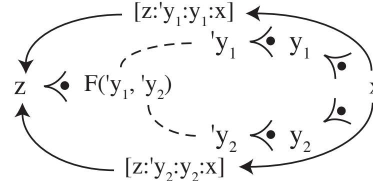


Figure 4. Path between Variables

- Communication consists in transferring the value of a local variable to a remote one. Communication time and lack of synchronization create a lag between the source and the image, which is modelled by $distant \prec local$.
- In state transition systems, a computation $f(X)$ is instantaneously computed. By writing $y \prec f(X)$, we model the fact that the computation takes time and that the value of y is based on the value of the inputs (X) at the beginning of the computation.

5.2 Example

We use the observation relations to describe the architecture of the example (see Figure 3). The distribution of the system is defined by the image variables $`speed$, $`chosen$ and $`actuator$. These variables are copies of the values of the variable $speed$, $chosen$ and $actuator$ sent through the communication bus.

The computation of the value of the $actuator$ variable is based on the values of the variables $`speed$ and $`chosen$. A $control$ function defines the computation of the variable $actuator$. This function is used in an observation relation binding the $actuator$ variable with the copies of the current speed and the chosen speed.

5.3 Path between Variables

Even if two variables are not directly related by any observation relation, they can be related by a set of observations. In the example, the value of $`actuator$ indirectly depends on the values of the variable $speed$. We want to be able to describe such an indirect dependency.

A set of observation relations defines an oriented graph where each variable is a node and observations are the edges that link the sources to the images. Given a set of observations, two variables are linked if there is a path between the nodes of these variables. Such a path represents the propagation of variable values through the system.

When none of the observations model a computation, there always exists a unique observation path between a given source and an image. But several observation paths can appear when a computation involves several source arguments. In figure 4, $F(`y_1, `y_2)$ has two images as input data, so two distinct observation paths have to be separately studied to verify time properties attached to the pair (z, x). Therefore, we define an observation path as a distinguishable sequence of variables. For example, the two paths between z and x are defined by the sequences $[z : `y_1 : y_1 : x]$ and $[z : `y_2 : y_2 : x]$.

The timed properties of the system are defined as properties on the propagation time of the values between two nodes. They express the time shifts that are introduced by the system architecture.

For each observation relation, the time shift is defined by the observation clock. The time shift along a path is defined by the composition of the observation clocks.

Definition 7 *Given the set of observation relations Obs that defines the architecture of the system and an execution σ, a path $p = [x_n : x_{n-1} : \ldots : x_0]$ between two variables x_n and x_0 defines a set of clock:*

$$\mathcal{C}lock(p).\sigma \triangleq \left\{ c_1 \circ c_2 \circ \ldots \circ c_n \;\middle|\; \begin{array}{l} \forall\, k \in [1..n], \exists\, f_k, X_{k-1} : \\ (x_k \prec f_k(X_{k-1})) \in Obs \\ \wedge\, x_{k-1} \in X_{k-1} \,\wedge \\ \forall\, i : x_k.\sigma_i = f_k(X_{k-1}.\sigma_{c_k(i)}) \end{array} \right\}$$

6 Timed Properties

The observation relations describe the system architecture. To complete our framework, we define the desired timed properties that specify the behavior of the variables and the relation between their timelines.

6.1 Timeline Properties

Timeline properties express the intrinsic necessity for a variable to have its value renewed often enough. That is to say, we bound the duration between two updates. In particular, this describes two behaviors: a *sporadic* variable keeps each value for a minimum duration and, on the contrary, an *alive* variable has to be updated often, no value can be kept longer than a given duration.

Definition 8 *The steadiness of a variable x is:*

$$\sigma \models x\{Steadiness(\delta, \Delta)\} \triangleq \forall\, i : d_x.\sigma_i \in [\delta, \Delta[$$

6.2 Relations Properties

We give timed properties on the propagation of values on a path between two variables with a set of predicates on the clocks $\mathcal{C}lock$ defined in the section 5.3

Definition 9 *Given a path $Path$ between two variables y and x, we define the parametrized relation between y and x defined by $Path$.*

$$\begin{array}{l} \sigma \models Path \left\{ \begin{array}{c} Predicate_1(\delta_1, \Delta_1), \\ Predicate_2(\delta_2, \Delta_2)... \end{array} \right\} \triangleq \\ \quad \exists\, c \in \mathcal{C}lock(Path).\sigma : Predicate_1(c, \delta_1, \Delta_1) \,\wedge \\ \qquad\qquad\qquad\qquad\qquad Predicate_2(c, \delta_2, \Delta_2) \,\wedge \\ \qquad\qquad\qquad\qquad\qquad \ldots \end{array}$$

$$
\begin{aligned}
Lag(c,\delta,\Delta) &\triangleq \delta \leq \hat{y}.\sigma_i - \hat{x}.\sigma_{c(i)} < \Delta \\
Latency(c,\delta,\Delta) &\triangleq \delta \leq T.\sigma_i - \hat{x}.\sigma_{c(i)} < \Delta \\
Shift(c,\delta,\Delta) &\triangleq \delta \leq T.\sigma_i - T.\sigma_{c(i)} < \Delta \\
Freshness(c,\delta,\Delta) &\triangleq \delta \leq T.\sigma_{c(i)} - \hat{x}.\sigma_{c(i)} < \Delta
\end{aligned}
$$

Figure 5. Predicates Characterizing The Link Between Two Variables

Such a relation is satisfied if among the clocks that bind the variables y and x, there is at least one clock that satisfies the predicates. Henceforth, for a relation between two variables y and x and a clock c, we use the predicates given in Figure 5:

- Predicate Lag is used to limit the time between the update of the source and the corresponding update of the image.
- Predicate $Latency$ quantifies the time elapsed since the appearance of the image's current value on the source and the current instant.
- Predicate $Shift$ bounds the time between the current instant and the instant when the current image value was taken on the source.
- Predicate $Freshness$ restricts the time intervals during which a source value is observable. The observation clock insures the source values are picked out during these intervals.

6.3 Example

We illustrate the timed properties on the example. The timed properties of the system are expressed on the observation relations of the system architecture definition (Figure 3) and are given Figure 6.

The properties of the variables $speed$ and $chosen$ are expressed using the steadiness. The variable $speed$ has the $Steadiness$ predicate parametrized by two bounds since it is not renewed too often in order to be significant and since it must still be updated often enough to be consistent with the real speed of the vehicle.

Due to the physical properties of the communication bus, the related observation relations are parametrized by a minimum bound on the $Shift$ predicate. These states that the value of a variable cannot be sent faster than the communication bus enables it. An upper bound is added in order to guarantee that the value available to the control system is consistent with the real value of the variables.

In order to give the timed requirements on the values used to control the vehicle speed, we express these requirements as parameters on the full processing chain. First we give requirements on the relation binding the value of variable $`actuator$ to the value of $speed$ through the system observation relations. The value of $speed$ is mostly valid when it has just been updated. We want the

```
- variables behaviours:
```
$$speed\ \{Steadiness(\delta_1,\Delta_1)\}$$
$$chosen\ \{Steadiness(\delta_2,+\infty)\}$$
```
- communications properties:
```
$$[`speed : speed]\ \{Shift(\delta_4,\Delta_4)\}$$
$$[`chosen : chosen]\ \{Shift(\delta_4,\Delta_4)\}$$
$$[`actuator : actuator]\ \{Shift(\delta_4,\Delta_4)\}$$
```
- complete processing chains:
```
$$[`actuator : actuator : `speed : speed]\ \{Latency(0,\Delta_5)\}$$
$$[`actuator : actuator : `chosen : chosen]\ \{Shift(0,\Delta_6)\}$$

Figure 6. System Timed Properties

total time elapsed between the appearance of this value and its use as the $actuator$ value to be short so we give a $Latency$ predicate to parameter this relation. In the relation binding the $`actuator$ and the $chosen$ variables, the value of the $chosen$ variables is always "timely correct" as it may not change during a cruise. The value used to produce the $`actuator$ value must be one that was taken recently by the $chosen$ variable. That is why we use the $Shift$ parameter.

6.4 Specification of a System

Finally, a specification is given by a couple $\langle Archi, Prop\rangle$. $Archi$ is a set of observation relations that describes the architecture of the system. $Prop$ is a set of properties. Some are intrinsic properties that define when the variable values are renewed; some are relation properties that are parametrized by a set of predicates and that define the relation between the values of two variables. We call $SPath$ the set of paths that are used to define the timed properties of the system.

7 Feasibility Analysis

The specification defines a state transition system where the timelines of the variables are restricted by the timed properties. The system is feasible if the specification defines at least one infinite execution. We build here a transition relation that defines a system equivalent to the specification.

7.1 Definition of the Analysis System

The transition relation of the system describes the behavior of the variables of the system with respect to their relations and timed properties. A variable transition relation describes the behavior of one variable. It defines which values can be used for an update and when an update occurs. We define the global transition relation of the system as the conjunction of variable transition relations.

Definition 10 *Given the variables defined by the architecture of the system $X = \{x_k | k \in [1..n]\}$ and the corresponding variable transition relations defined by the timed properties, $\rho = \{\rightarrow_k | k \in [1..n]\}$ the global transition relation is defined by:*

$$\sigma_i \rightarrow \sigma_{i+1} \triangleq T.\sigma_{i+1} = T.\sigma_i + 1 \ \wedge \ \bigwedge_{k=1}^{n} \sigma_i \rightarrow_k \sigma_{i+1}$$

Note that each global transition induces a time step. We now explain how the transition relation of each variable is built.

7.2 Interval of Validity

The timed properties of the specification limit the instants a value can be used to produce other variables values. For each value, two time intervals are defined: the possible update instants, i.e. when a new value can be assigned to the image; and how long this value can be kept. These intervals depend on the predicates that parametrize the relation between the variables. For example, the Lag predicate defines the possible update instants of the image and the $Shift$ or $Freshness$ predicates define the instants a value can be used. Each predicate defines an interval, all predicates must be satisfied so the timed property defines an interval that is the intersection of all predicate intervals.

These intervals also depend on the timed characteristics of the value that is used. The following type is introduced to store the characteristics of a value:

$$Value = \langle \mathbb{T}, \mathbb{T}, Path \rangle$$

It stores the timed characteristics of a value of the source that is propagated along a path $Path$. The two elements in the time domain $\mathbb{T}$ are the update instant of this value and the duration this value was continuously kept. These timed characteristics define when a value can be used.

When we consider a timed property between two variables, if we know the timed characteristics of a value, then we can define the intervals of instants where this value can be used to produce a new value of the image and when this value can be kept. We define two functions that give these intervals.

Definition 11 *A time property between two variables y and x along a path p defines an interval $UpdateValid$ when a value $v = \langle \hat{x}, d_x, p \rangle$ of x which appeared at the instant $\hat{x}$ and kept for a duration of d_x can be used to update y and replace a value updated at $\hat{y}$.*

$$UpdateValid(\langle \hat{x}, d_x, p \rangle, \hat{y}) \triangleq \left[\begin{array}{c} max \left(\begin{array}{c} \hat{y} + \delta_{Steadiness}, \hat{x} + \delta_{Lag} \\ \hat{x} + \delta_{Latency}, \\ \hat{x} + \delta_{Freshness} + \delta_{Shift} \end{array} \right), \\ min \left(\begin{array}{c} \hat{y} + \Delta_{Steadiness}, \hat{x} + \Delta_{Lag}, \\ \hat{x} + \Delta_{Latency}, \\ min(\hat{x} + d_x, \hat{x} + \delta_{Freshness}) + \Delta_{Shift} \end{array} \right) \end{array} \right]$$

It also defines an interval $ValueValid$ of the instants this value can be kept.

$$ValueValid(\langle \hat{x}, d_x, p \rangle, \hat{y}) \triangleq \left[\begin{array}{c} max \left(\begin{array}{c} \hat{x} + \delta_{Latency}, \\ \hat{x} + \delta_{Freshness} + \delta_{Shift} \end{array} \right), \\ min \left(\begin{array}{c} \hat{x} + \Delta_{Latency}, \hat{y} + \Delta_{Steadiness}, \\ min(\hat{x} + d_x, \hat{x} + \delta_{Freshness}) + \Delta_{Shift} \end{array} \right) \end{array} \right]$$

7.3 The History of Values

In an observation relation $y \prec f(X)$, the value of y depends on the values of the variables of X. So when building a new value for y, we must check that the values of these variables are correct. Moreover, y can be linked to other variables through X. So we must also know which values of other variables are used to build X value.

Definition 12 *Given an execution σ, a value v contains the timed characteristics about a path p in a state σ_i if we have:*

$$\begin{array}{rl} Charac(v, i, p).\sigma \triangleq & \exists\, p', \exists\, z : p = p' :: [z] \\ & \wedge\, \forall\, c \in \mathcal{C}lock(p).\sigma : \\ & \quad v = \langle \hat{z}.\sigma_{c(i)}, d_z.\sigma_{c(i)}, p \rangle \end{array}$$

The operator :: defines the concatenation of two sequences. Such a value stores the timed characteristics of the value of the source of the path that is used to set the current value of the path's image. This is the value of the source in the instants pointed by the clocks of the path. For a state σ_i, such a value is unique. We create a set with the timed characteristics of the sources of the paths that are used to set the value of a variable. We are only interested in the paths used to describe the timed properties of the specification.

Definition 13 *For a variable x and an execution σ, we define the set of values that are used to build the value of x in a state σ_i and that are linked to x through a set of paths $SPath$.*

$$\begin{array}{l} SrcCharac(x, i, SPath).\sigma \triangleq \\ \{v | \exists\, p \in SPath : \exists\, p' : p = [x] :: p' \wedge Charac(v, i, p').\sigma\} \end{array}$$

The evolution of a variable is bound to the recent evolution of other variables and so to the value of the other variables in previous states. A transition relation is a predicate on a pair of states that defines the behavior of a system. In order to define the transition relation of the system defined by the specification, an auxiliary variable, called history, is introduced. We consider an observation relation $y \prec f(X)$ and one of its observation clock c so that $y.\sigma_i = f(X.\sigma_{c(i)})$. The clock c is increasing so only the values taken by X between states $\sigma_{c(i)}$ and σ_i are used to compute the next value of y. The variable history $H_{y \prec f(X)}$ stores the values that are used to build the values of X in these states.

Definition 14 *Given an observation relation $y \prec f(X)$, and the set of paths $SPath$ that are used to describe the timed properties of the system and that link y to other variables, the variable $H_{y \prec f(X)}$ is defined by:*

$$\forall\, \sigma, \forall\, i : H_{y \prec f(X)}.\sigma_i \triangleq \left\{ \bigcup_{x \in X} SrcCharac(x, j, SPath').\sigma \;\middle|\; \begin{array}{l} c \in \mathcal{C}lock([y : x]).\sigma \\ \wedge\, x \in X \;\wedge\; j \in [c(i)..i] \end{array} \right\}$$

where

$$SPath' = \{[x] : p \mid x \in X \;\wedge\; ([y : x] :: p) \in SPath\}$$

History variable is a set of sets of values that gives the characteristics of the values that can be used to build the next value of the image. The values that are linked to the same value of X in the same instant are stored in the same set. A partial order relation that is based on the chronological order is defined on the set of values that are stored in the history.

Definition 15 *An order relation is defined between the set of values and for an execution σ.*

$$\begin{aligned}\forall \mathcal{V}_1, \mathcal{V}_2 : \mathcal{V}_1 < \mathcal{V}_2 \triangleq \quad & \exists\, i, j : i < j \;\wedge\; \exists\, x, \exists\, SPath : \\ & \mathcal{V}_1 = SrcCharac(v_1, i, SPath).\sigma \\ & \wedge\, \mathcal{V}_2 = SrcCharac(v_2, j, SPath).\sigma\end{aligned}$$

7.4 Variable Transition Relation

We define here the transition relation for a variable y image of an observation $y \prec f(X)$ that describes the evolution of y when time is increased.

There are two possible evolutions: the image is updated with a new value or the same value is kept. The possibility to use a value does not only depends on the value taken by the variables in X. It also depends on the values used to produce X value. Given the architecture of the system, we check that all values used to produce y value satisfy the specification. So we check the characteristics of the values in the history. The intervals defined in section 7.2 are used to define the variable transition relation.

Definition 16 *For each variable relation $y \prec f(X)$ in a specification, a variable transition relation is defined by:*

$$\forall\, \sigma_1, \sigma_2 : \sigma_1 \to \sigma_2 \triangleq$$
$$\left(\begin{array}{c}\left(\begin{array}{c}\hat{y}.\sigma_2 \neq T.\sigma_2 \;\wedge\; \forall\, v \in min(H_{y \prec f(X)}.\sigma_2) : \\ T.\sigma_2 \in ValueValid(v, \hat{y}.\sigma_1)\end{array}\right) \\ \vee \\ \left(\begin{array}{c}\hat{y}.\sigma_2 = T.\sigma_2 \;\wedge\; \forall\, v \in min(H_{y \prec f(X)}.\sigma_2) : \\ T.\sigma_2 \in UpdateValid(v, \hat{y}.\sigma_1)\end{array}\right)\end{array}\right)$$

The history only stores the values of the sources no older than the values that set the image current value (pointed by c). So $min(H_{y \prec f(X)})$ denotes the set of values currently used to define the image value. The state σ_2 is influenced by the state σ_1 through the definition of the history and the instant when the value of y in σ_1 was updated.

For a variable that is not the image of an observation relation, the dedicated transition relation is only defined by its intrinsic timeline property.

7.5 Reduction and Exploration of the System

We proceed to the analysis of the system defined by the global transition relation. We must explore the executions to prove the existence of an infinite execution and thus to prove the system feasibility. However the specification defines a system with an infinite number of states. There is no bound on the time T and other timed variables such as the update instants. So these variables can take an infinite number of values.

In order to perform a finite exploration of the states of an execution, we build a system bi-similar to the specification but where the variables take a finite number of values. This is possible if the shift between all timed variables is bounded. So this is possible if the specification states upper bounds on the time a value can be used by the system.

Given an observation $y \prec f(X)$, we bound the values of the timed characteristics that are used to check the validity of the value assigned to y. We look for a bound on all update instant stored in the history variable. All these values are in the interval between the update instants of the values at the beginning of the paths and the current time.

Proposition 1 *Given an observation $y \prec f(X)$ we have:*

$$\forall\, i : \forall\, \mathcal{V} \in H_{y \prec X}.\sigma_i, \forall\, \langle v_{\hat{x}}, v_{d_x}, p\rangle \in \mathcal{V} : v_{\hat{x}} \in [min_{src}, T.\sigma_i, v)]$$

where:

$$min_{src} = min\left(\left\{\hat{s}.\sigma_{c(i)} \;\middle|\; \begin{array}{c}\exists\, p \in SPath, \exists\, p' : \\ p = [y] :: p' :: [s] \,\wedge \\ c \in Clock(p).\sigma\end{array}\right\}\right)$$

We want to give a constant maximum size of this interval in all states. In a system where the relations between the variables and the sources are parametrized by a $Latency$ predicate, then the shift between all variables is bounded in all states by the most permissive $Latency$ predicate i.e. the one with the maximum upper bound. We restrict our analysis to such systems. If this property is not explicitly stated in the specification, then we use a set of propositions. Here are their principles:

- the latency that parameters an observation relation can be deduced from the combination of other predicates that parameters the relation such as $Steadiness$ and Lag;
- if along a path defined by a set of observations, all observations are parametrized by a latency predicate then so is the full path;
- if there are multiple sources all with a $Steadiness$ predicate parametrizing their behavior, and if there is a $Latency$ predicate binding one of this source to the image, then all are bound to the image with a $Latency$ predicate.

In the example, there is no upper bound on the $Steadiness$ predicate of the variable $chosen$. For the system to be analyzable, such a property must be added. A large bound must be chosen in order to not interfere with other properties.

Based on these properties and for such a system, we define a system where all values of the instants are stored modulo the length of an analysis interval denoted by L. L is chosen by the specification as a bound greater than the upper bounds on the variables $Steadiness$ and the paths $Latency$.

In the system defined by the specification, transitions are based on the time differences between the instants characterizing the variable timelines. These differences do not exceed the length L. Thus, for each state, if the value

of the time T is known and if the values of the other variables are known modulo L, then for each variable there is only one possible real value that can be computed using the value of T. Considering the clock values modulo this length does not add or remove any behavior of the original system.

Consequently the equivalence relation that is defined by the equality of timed variables modulo L is a bi-simulation. In this system, all timed variables have a finite number of values. So we obtain a system that is bi-similar to the system defined by the specification and that has a finite number of states.

In order to prove the system feasibility, we explore the executions that are defined by the finite system using a depth first search algorithm. A loop denotes an infinite execution. This proves the feasibility of the specification.

8 Verification of an Implementation

We explain here how to check an implementation. An implementation is correct if each of the behaviors it defines satisfy the specification. In other words, the implementation defines a set of executions that must be included in the executions defined by the specification.

8.1 Value Availability

In order to analyze an implementation, it must be modelled. An implementation defines how the values taken by each variable are transmitted through the architecture of the system. To check the timed properties of the implementation, we focus on the implementation properties that define the instants when a source value is available to the image.

Given an observation relation $y \prec f(X)$ and for each value taken by X, we define different states of availability. These states are the different steps between the instant when a value is assigned to the source and the instant when the image value is bound to this source value:

- initial (It): the value is currently assigned to the source;
- sent (St): the value has been stored to be later available to the image. For example a message has been created containing the source value or a component has read the inputs used for the computation;
- received (Rd): the value is available to the image. The message containing the value of the source has been received or the computation of the image new value is completed;
- delivered (Dd): the value of the source has been used to set the current value of the image.

Each value is in one and only one of the sent, received or delivered states but it may be both in the initial state and in another state. These states of availability do not exactly describe the different states of a value in an implementation. But the behavior of an implementation can be modelled with these states.

The history variable is split into different sets of values depending on the availability of each value. These sets are in fact sequence of sets of values. They are ordered with the order relation on sets of values (chronological order, the oldest one is the first in the sequence).

Definition 17 *Given an observation relation $y \prec f(X)$ and the paths $SPath$ used to describe the timed properties of y, the sequences that describe the states of availability of the values are defined by:*

$$\begin{array}{l}\forall\, \sigma, \forall\, i : \\ It.\sigma_i \triangleq \{ \bigcup_{x\in X} SrcCharac(x,i,SPath).\sigma \} \\ \wedge\, Dd.\sigma_i \triangleq \left\{ \begin{array}{l} \bigcup_{x\in X} SrcCharac(x,c(i),SPath).\sigma \\ \quad \mid c \in \mathcal{C}lock([y:x]).\sigma \end{array} \right\} \\ \wedge\, (It :: St :: Rd :: Dd).\sigma_i \subseteq H_{y\prec f(X)}.\sigma_i \end{array}$$

The sequences It and Dd are singletons. A value goes through the four states chronologically.

8.2 Model of the Specification

In order to check the satisfaction of the specification by an implementation, we give a model of the specification in the same semantic we use to model an implementation. Such a model is described by defining elementary transitions. An elementary transition relation model the evolution of the values states of availability in the observation relations of the system. These elementary transition relations are used to build the variable transition relation of the image of an observation. The variable transition relations are then used to build the global transition relation.

We use a semantic close to TLA [5] that is based on actions. An action is a predicate on two states, and an elementary transition relation is defined as a disjunction of actions. The actions give the different evolutions of the availability of a value and when these evolutions are allowed by the specification.

Definition 18 *Given a set of the actions $A = \{a_k | k \in [1..n]\}$ for the evolution of a value from one state of availability to the next one, we define an elementary transition relation $\rightarrow$*

$$\forall \sigma_i, \sigma_j : \sigma_i \rightarrow \sigma_j \triangleq \bigvee_{k=1}^{n} a_k.\sigma_i.\sigma_j$$

We now define the actions used to build a model of the specification. Except if it is stated by the action, all variables have the same value in both states of an action.

8.2.1 Sender

This elementary transition relation rules the evolution of the current value of the image to the sequence St. There are two actions: the value can be sent or not.

$$\begin{array}{l}\forall \sigma_i, \sigma_j : \\ \quad Send.\sigma_i.\sigma_j \triangleq St.\sigma_j = St.\sigma_i :: It.\sigma_j \\ \quad Idle.\sigma_i.\sigma_j \triangleq true \end{array}$$

The current value is sent when the value in the sequence It is added to the sequence St.

8.2.2 Receiver

This elementary transition relation rules the evolution of the values from the sequence St to the sequence Rd. Each value can be passed to the next sequence or lost.

$$\begin{aligned}&\forall \sigma_i, \sigma_j :\\ &Rcv(St_L, St_R).\sigma_i.\sigma_j \triangleq St.\sigma_i = Merge(St_L, St_R) :: St.\sigma_j\\ &\qquad\qquad\qquad\qquad \wedge Rd.\sigma_j = Rd.\sigma_i :: St_R\\ &Lose(St_L).\sigma_i.\sigma_j \quad \triangleq St.\sigma_j = St.\sigma_i \backslash St_L\end{aligned}$$

The function $Merge$ builds the ordered sequence union of two sequences. The actions Rcv passes the values St_R from St to Rd but lose the values in St_L. The actions are parametrized by sequence of values since the possible actions depend on the number of values in the sequence St.

8.2.3 Image

This relation decides which value is assigned to the image. It can keep the same value or take a new value that is in the sequence Rd. Some values of the sequence Rd can also be removed from this sequence. For an observation $y \prec f(X)$ we have:

$$\begin{aligned}&\forall \sigma_i, \sigma_j :\\ &Update(\mathcal{V}, Rd_L).\sigma_i.\sigma_j \triangleq \hat{y}.\sigma_j = T.\sigma_j\\ &\qquad \wedge Rd.\sigma_i = Merge([\mathcal{V}], Rd_L) :: Rd.\sigma_j\\ &\qquad \wedge Dd.\sigma_j = [\mathcal{V}]\\ &\qquad \bigwedge_{v \in \mathcal{V}} T.\sigma_j \in UpdateValid(v, \hat{y}.\sigma_i)\\ &Keep(Rd_L).\sigma_i.\sigma_j \triangleq Rd.\sigma_i = Rd_L :: Rd.\sigma_j\\ &\qquad \wedge Dd.\sigma_i = [\mathcal{V}]\\ &\qquad \bigwedge_{v \in \mathcal{V}} T.\sigma_j \in ValueValid(v, \hat{y}.\sigma_i)\end{aligned}$$

The action $Keep$ check that the values can be kept by using the predicate $ValueValid$. The action $Update$ is defined with the predicate $UpdateValid$ that checks that y can be updated with the new value.

8.2.4 Variable Transition Relation

We build the variable transition relation of a variable y by using the elementary transition relation of the observation which image is y. The variable transition relation is defined as a sequence of elementary transition relations.

Definition 19 *Given an observation $y \prec x$ and the elementary transition relations:*

$$\rightarrow_{sd}; \rightarrow_{rcv}; \rightarrow_{img}$$

Then the transition relation $\rightarrow_y$ that defines the behavior of y is:

$$\begin{aligned}&\forall \sigma_i : \sigma_i \rightarrow_y \sigma_{i+1} \triangleq\\ &\qquad \exists\, \sigma_s, \sigma_r : \sigma_i \rightarrow_{sd} \sigma_s \rightarrow_{rcv} \sigma_r \rightarrow_{img} \sigma_{i+1}\end{aligned}$$

The intermediary states between the elementary transitions are hidden to ensure a separated execution. A definition of the variable transition relation could be given as a conjunction of elementary relations but this definition eases the expression of the elementary relation transitions. The global transition relation is defined as the conjunction of the variable transition relations as we did in the feasibility analysis.

This model of the specification is equivalent to the state transition system defined for the feasibility analysis. The specification do not allow loss of values, but a loss is equivalent to not finally using this value to update the image. Therefore this model is bi-similar to the specification.

8.3 Model of the Implementation

The implementation is modelled by redefining the same actions as the specification. So we describe the evolution of the values through the same states of availability and use the same four sequences. We use some part of the example to illustrate how to define such a model. We first discuss the properties of the communication protocol. We suppose all communications are done through the same communication bus. To model a communication protocol, two characteristics need to be abstracted: when are the messages sent and what is the communication time. Moreover, are these characteristics determinate or is there a jitter? In a time triggered protocol, the evolution to the sent availability state is decided by the value of the time T. The evolution to the received state depends on the communication time. The value in the availability sequences are redefined as a new type that contains the instant when a value is sent.

$$Value = \langle \mathbb{T} \rangle$$

If the messages are sent with a period of P with a phase ϕ and if the communication time is d then we define the following actions for a communication such as '$speed \prec speed$. In the example no loss is allowed. Only one value at a time can be received, and all values that are received are directly assigned to the image.

$$\begin{aligned}&\forall \sigma_i, \sigma_j :\\ &Send.\sigma_i.\sigma_j \triangleq T.\sigma_j = \phi \pmod P\\ &\qquad \wedge It.\sigma_j = \{\langle T.\sigma_j \rangle\}\\ &Idle.\sigma_i.\sigma_j \triangleq T.\sigma_j \neq \phi \pmod P\\ &Rcv(St_L, St_R).\sigma_i.\sigma_j \triangleq St_R = \{\langle T_{sent} \rangle\}\\ &\qquad \wedge T.\sigma_j - T_{sent} = d\\ &\qquad \wedge St.\sigma_i = St_R :: St.\sigma_j\\ &\qquad \wedge Rd.\sigma_j = Rd.\sigma_i :: St_R\\ &\qquad \wedge St_L = \emptyset\\ &Lose(St_L).\sigma_i.\sigma_j \triangleq St_L = \emptyset\\ &Update(\mathcal{V}, Rd_L).\sigma_i.\sigma_j \triangleq Rd.\sigma_i = [\mathcal{V}]\\ &\qquad \wedge Rd.\sigma_j = \emptyset\\ &\qquad \wedge Dd.\sigma_i = [\mathcal{V}]\\ &\qquad \wedge Rd_L = \emptyset\\ &Keep(Rd_L).\sigma_i.\sigma_j \triangleq Rd_L = \emptyset\\ &\qquad \wedge Rd.\sigma_i = \emptyset\end{aligned}$$

Here the period P is used to define when the $Send$ action is allowed and so when values are passed to the sent availability state. The communication time and the instant a value is sent are used in the Rcv action. They define when the values are passed to the received availability state.

In an observation that models a computation, the availability state depends on the same kind of characteristics which are when the computation starts and the possible

execution time. The results of a scheduling analysis can be used to give these characteristics.

8.4 Correctness of an Implementation

An implementation is correct if the set of executions it defines is included in the executions that are defined by the specification. So the model of the specification must simulate the implementation. In order to check this property, we build a state transition system similar to the synchronized product of labelled transition systems. The actions are used as labels on the transition of the systems.

Definition 20 *Given the set of actions* $A_I = \{a_{I_k} | k \in [1..n]\}$ *that defines an elementary transition of the implementation and the set of actions* $A_S = \{a_{S_k} | k \in [1..n]\}$ *that defines the corresponding elementary transition of the specification, we define the set of couples of actions where the actions with the same label are joined.*

$$A = \{\langle a_{I_k}, a_{S_k} \rangle | k \in [1..n]\}$$

An elementary transition relation of the system that checks the correctness of the implementation is defined by:

$$\forall\, \sigma_i, \sigma_j : \sigma_i \rightarrow \sigma_j \triangleq (\bigvee_{k=1}^{n} a_{I_k}.\sigma_i.\sigma_j \wedge a_{S_k}.\sigma_i.\sigma_j) \\ \wedge \forall\, k, \forall\, \sigma_l : (a_{I_k}.\sigma_i.\sigma_l \Rightarrow a_{S_k}.\sigma_i.\sigma_l)$$

In this definition, the second part states that if a transition between two states is allowed by the implementation, it must be allowed by the specification. So if there is a conflict between the specification and the implementation then there is a deadlock. The global system is built by using these elementary transition relation to build the variables transition relations. Note that two actions are different if their parameters are different. For example $Rcv(\emptyset, \mathcal{V})$ is different from $Rcv(\emptyset, \emptyset)$. If the sizes of the sequences are bounded, then the number of different actions is also bounded. The sizes of the sequences are bounded if the system can be reduced to a finite system.

8.5 Reduction of the System

In order to proceed to the analysis of the system, we here also build a finite system equivalent to the system that is defined by the specification and the implementation. This is only possible if the variables introduced to define the implementation properties have properties that allow the reduction technique to be used. So the timed variables introduce by the model of the implementation must have a bounded shift to the time T. These properties are required to proceed to an automatic verification and must be stated by the user.

In this system, a deadlock exists if no behavior can be taken. A deadlock denotes either an incompatibility between the specification and the implementation or that the specification is not feasible. So the correctness of the implementation is checked with a model checking algorithm used to detect deadlocks.

9 Conclusion

We specify an abstract model postponing task and communication scheduling to specify real-time systems. Our framework proposes to specify real-time systems as a set of links between system variables. The timed properties of the system characterize the time-shift along the propagation of values in the system. They state that all the values that are used must be timely correct with respect to the user requirements. A dedicated state transition system that is bi-similar to the specification, is built to proceed to a feasibility analysis. We finally describe how to model an implementation of the system. A dedicated state transition system is also built to check the correctness of the implementation with respect to the specification. Perspectives are to enhance the implementation of the tool used to build the analysis state transition systems. This tool can then be used to proceed to the analysis of a larger scale example.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[2] S. Anderson and J. K. Filipe. Guaranteeing temporal validity with a real-time logic of knowledge. In *ICDCSW '03: Proc. of the 23rd Int'l Conf. on Distributed Computing Systems*, pages 178–183, 2003.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Data consistency in hard real-time systems. Technical report, 1993.

[4] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quinnec. The observation : an abstract communication mechanism. *Parallel Processing Letters*, 9(3):437–450, 1999.

[5] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[6] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[7] G. Roşu and S. Bensalem. Allen Linear (Interval) Temporal Logic – Translation to LTL and Monitor Synthesis. In *International Conference on Computer-Aided Verification (CAV'06)*, number 4144 in Lecture Notes in Computer Science, pages 263–277. Springer Verlag, 2006.

[8] X. C. Song and J. W. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.

[9] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.

[10] M. Xiong, R. Sivasankaran, J. A. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints: exploiting data semantics. In *RTSS '96: Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 240–253, 1996.

Multiprocessor Scheduling

LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets

Shelby Funk and Vijaykant Nadadur
University of Georgia
Athens, GA, USA
$\{$*shelby,nadadur*$\}$@*cs.uga.edu*

Abstract

This paper introduces LRE-TL, a scheduling algorithm based on LLREF, and demonstrates its flexibility and improved running time. Unlike LLREF, LRE-TL is optimal for sporadic task sets. While most LLREF events take $O(n)$ time to run, the corresponding LRE-TL events take $O(\log n)$ time. LRE-TL also reduces the number of task preemptions and migrations by a factor of n. Both identical and uniform multiprocessors are considered.

1. Introduction

In hard real-time systems, jobs have specific timing requirements, or deadlines. In these systems, inability to meet a deadline is considered a system failure. Therefore, it must be known that no deadlines will be missed before running the system. One way to do this is to use an optimal scheduling algorithm. We say a scheduling algorithm is optimal if it will schedule all jobs to meet their deadlines whenever it is possible to do so. For example, the earliest deadline first (EDF) scheduling algorithm [1], [2] is known to be optimal on uniprocessors when preemption is allowed.

As multiprocessors become more popular, they are used in a wider variety of applications, including real-time and embedded systems. This paper considers scheduling on identical multiprocessors, which contain m identical processors, and on uniform multiprocessors, which contain m processors whose speeds may vary from one another. While there are many advantages to using multiprocessor systems, scheduling with these systems can be complex. For example, on multiprocessors EDF is not optimal. In fact, EDF might miss deadlines on multiprocessors even if processors are idle approximately half the time [3], [4], [5].

To date, optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than other non-optimal algorithms. Some common restrictions are that (i) they have high overhead, (ii) they apply only to a restrictive model for jobs or processors, or (iii) the schedule must be quantum based (i.e., the scheduler is invoked every q time units for some constant q). There are two well known optimal multiprocessor scheduling algorithms, Pfair [6] and LLREF [7], each suffer from at least one of these shortcomings. While Pfair can schedule both periodic [1] and sporadic [8], [9] tasks, it applies only to quantum based systems on identical multiprocessors. On the other hand, LLREF can be scheduled on both identical and uniform multiprocessors, but it has high scheduling overhead and applies only to periodic task systems. This paper introduces a new scheduling algorithm, LRE-TL, which is based on the LLREF scheduling algorithm. We show LRE-TL is optimal for periodic and sporadic task sets and has much lower scheduling overhead than LLREF. Table 1 illustrates the running time of LRE-TL compared to LLREF (details provided in Section 4.4).

The remainder of this paper is organized as follows. Section 2 provides definitions of terms that will be used throughout the remainder of the paper. Section 3 provides an overview of the LLREF algorithm, which is used as a starting point for describing LRE-TL. Section 4 describes the LRE-TL scheduling algorithm, proves it is optimal for both periodic and sporadic tasks, and compares it to LLREF. Section 5 compares an LLREF schedule and an LRE-TL schedule. Section 6 discusses how to schedule LRE-TL on uniform multiprocessors. Finally, Section 7 provides some concluding remarks.

2. Model and Definitions

This paper considers a global multiprocessor scheduling algorithm for periodic [1] and sporadic [8], [9] task sets whose deadlines equal their periods.

We assume that tasks are independent and can be preempted at any time.

A task T_i is a program that repeatedly invokes jobs $T_{i,1}, T_{i,2}, \ldots$. Each task T_i is described using the 3-tuple (ϕ_i, p_i, e_i), where ϕ_i is T_i's offset, p_i is its period and e_i is its worst case execution time. Each job $T_{i,j}$ has a release time $a_{i,j}$, an execution requirement e_i and a relative deadline p_i — if $T_{i,j}$ arrives at time $a_{i,j}$ then it must be allowed to execute for e_i time units during the interval $[a_{i,j}, a_{i,j}+p_i[$. At any time $t \in [a_{i,j}, a_{i,j}+p_i[$, we say T_i's deadline at time t is $d_{i,j} = a_{i,j} + p_i$. If T_i is a periodic task set, then it invokes its first job at time $t = \phi_i$ and all the remaining jobs are invoked exactly p_i time units apart — i.e., $a_{i,j} = (j-1)p_i$ for all j. If T_i is a sporadic task, then it invokes its first job at any time $t \geq \phi_i$ and the remaining jobs are invoked no less than p_i time units apart — i.e., $a_{i,1} \geq 0$ and $a_{i,j} \geq a_{i,j-1} + p_i$ for all $j > 1$. A task set $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes a set of n periodic or sporadic tasks. Throughout this paper, we will clearly state whether τ is assumed to be a periodic or a sporadic task set.

One important parameter used to describe a task is its utilization $u_i = e_i/p_i$, which is the proportion of time T_i executes between its arrival time and deadline. For sporadic tasks, the utilization measures the "worst-case average" — i.e., the average proportion of required computing time assuming the task has a worst case sequence of arrivals ($a_{i,j} = a_{i,j-1} + p_i$) during the interval under consideration. The total utilization of task set τ, denoted $U(\tau)$, is the sum of the individual task utilizations, viz., $U(\tau) = \sum_{i=1}^{n} u_i$.

The LLREF scheduling algorithm partitions the time horizon into a sequence of intervals $[t_{j-1}, t_j[$, where $t_0 = 0$ and for each $j \geq 1$,

$$t_j = \min_{t > t_{j-1}} \{t = k \cdot p_i \mid T_i = (p_i, e_i) \in \tau \text{ and } k \in \mathbb{Z}\}.$$

For any time $t \geq 0$, we let $[t_{0_t}, t_{f_t}[$ denote the interval that contains t. At each time t, every task T_i has a *local execution requirement*, $\ell_{i,t}$. This is the amount of time that T_i must execute between time t and time t_{f_t}. The progress of each task during the interval $[t_{j-1}, t_j[$ may be viewed as a 2 dimensional plane in which the horizontal axis represents time (T) and the vertical axis represents local execution time (L). This plane is called the TL-plane. A task's *local utilization* is the proportion of time T_i must execute during the remainder of the current TL-plane, namely $r_{i,t} = \ell_{i,t}/(t_{f_t} - t)$. A task set's total utilization R_t at time t is defined to be the sum of each task's local utilization, viz., $R_t = \sum_{i=1}^{n} r_{i,t}$.

For our discussion, we need to be able to distinguish which tasks are *active* at any time t. We say a task T_i is active at time t if $\ell_{i,t} > 0$. We let $Active(t)$ and $NA(t)$ be the set of active and non-active tasks, respectively.

	LLREF	LRE-TL
Running time		
Initialize TL-plane	$O(n)$	$O(n)$
A events (per event)	NA	$O(\log n)$
B&C events (per TL-plane)	$O(n^2)$	$O(n \log n)$
Other overhead		
Max preemptions (per event)	$O(m)$	$O(1)$
Max migrations (per TL-plane)	$O(mn)$	$O(m)$

Table 1. Comparison of LLREF and LRE-TL.

We consider the problem of scheduling periodic and sporadic tasks on identical and uniform multiprocessors. Throughout this paper, we let m denote the number of processors. Without loss of generality, we assume the speed of the processors is 1 for identical multiprocessors — i.e. each processor performs one unit of work per unit of time. We denote a uniform multiprocessor $\pi = [s_1, s_2, \ldots, s_m]$, where $s_i \geq s_{i+1}$ for $1 \leq i < m$. If processor s_i executes for t units of time, then it performs $s_i \times t$ units of work[1]. Below we discuss the implementation of LLREF and LRE-TL on identical multiprocessors first and then show how these algorithms can be extended for implementation on uniform multiprocessors.

3. A Brief Overview of LLREF

In order to understand the LRE-TL scheduling algorithm presented in this paper, we must first describe the LLREF algorithm presented in [7], where the authors also proved that LLREF is optimal for scheduling periodic task sets on identical multiprocessors. A task set τ can be scheduled to meet all deadlines on m identical processors if the following two conditions hold [6]

$$U(\tau) \leq m, \text{ and } u_{max}(\tau) \leq 1, \quad (1)$$

where $u_{max}(\tau) = \max_{1 \leq i \leq n}\{u_i\}$ is the maximum utilization of all tasks in τ.

Given a task set τ satisfying the conditions above, LLREF schedules the tasks so that the *local* utilization continues to satisfy the stated conditions. As stated above, LLREF divides the time horizon into a sequence of consecutive and non-overlapping TL-planes. Each TL-plane ends at some task's deadline and there are no

1. In a mild abuse of notation, we let s_i denote both the ith processor and its speed.

deadlines within any TL-plane. Every task has a local execution requirement $\ell_{i,t}$, which denotes the amount of time the task must execute during the interval $[t, t_{f_t}[$. At the beginning of each TL-plane, $r_{i,t_{0_t}}$ is set to u_i for each task T_i (i.e, $\ell_{i,t_{0_t}} = u_i(t_{f_t} - t_{0_t})$).

LLREF makes scheduling decisions with the aim of achieving the following two goals.

1) No processor idles while a job is waiting, and
2) No task's local utilization ever exceeds 1.

Whenever LLREF makes a scheduling decision, it selects the m tasks with the largest local remaining execution and executes them until the next scheduling event. If fewer than m tasks have positive remaining execution, then LLREF executes all tasks T_i with $\ell_{i,t} > 0$ for $\ell_{i,t}$ time units. We use the following definitions to describe the tasks LLREF selects to execute at a given time t.

Definition 1. *At time t let $T_{(1)_t}, T_{(2)_t}, \ldots, T_{(n)_t}$ denote the tasks of τ sorted in weakly decreasing order according to local remaining execution. Thus,*

$$\ell_{(1)_t,t} \geq \ell_{(2)_t,t} \geq \ldots \geq \ell_{(n)_t,t}.$$

Let x_t denote the maximum number of tasks that can execute simultaneously at time t. Because, no more than m tasks can execute at one time, $x_t \leq m$. If fewer than m tasks have remaining, work, then x_t is the number of tasks with remaining work, viz.,

$$x_t = \min\{m, |Active(t)|\}. \quad (2)$$

When a scheduling event occurs at time t, LLREF executes tasks $T_{(i)_t}$ for $1 \leq i \leq x_t$. These tasks execute without interruption until one of two events occurs, namely a bottom (B) event or a critical (C) event. Below, we discuss these two events in more detail assuming tasks $T_{(i)_s}$ for $1 \leq i \leq x_s$ are scheduled at time s and the B or C event occurs at time t.

B (bottom) events: These events occur when a task hits the "bottom" of the TL-plane (i.e., when a task depletes its local remaining execution). Clearly, this can only happen to one of the x_s tasks that were scheduled to execute at time s. Using our sorting notation, task $T_{(x_s)_s}$ is the executing task with the least remaining execution requirement that triggers a B event at time t. If tasks are not rescheduled at this point, then the processor executing task $T_{(x_s)_s}$ will become idle, which could cause the first stated goal to be violated.

C (critical) events: These events occur when some task's local utilization becomes 1. Clearly, a task's local utilization decreases if it is executing. Hence, a C event is caused by the non-executing task with the largest remaining local execution, namely $T_{(m+1)_s}$. If this task is not allowed to execute immediately, its local utilization will exceed 1. This not only violates the second goal, it also means that $T_{(m+1)_s}$ will miss its deadline at time t_{f_t}.

Because each of these events cause one of LLREF's goals to be violated, they trigger a scheduling event. When a scheduling event occurs, LLREF determines which tasks are in $T_{(i)_t}$ for $1 \leq i \leq x_t$ and schedules these tasks to execute.

Two shortcomings of LLREF have been noted. First, it incurs fairly high overhead. Second, it is only optimal for periodic tasks with deadlines equal to periods. Below, we discuss how to address these two shortcomings.

4. LRE-TL: A Modification of LLREF

We now present our algorithm, which we call LRE-TL (local remaining execution-TL). The name LLREF does not accurately describe our scheduling algorithm's behavior because we no longer select to execute the task with the largest local remaining execution first. Below, we first present an analytical result that proves our proposed algorithm continues to be optimal. We then describe how the LRE-TL algorithm makes scheduling decisions and how to handle sporadic arrivals. Finally, we present the algorithm in detail and discuss its running time.

As noted above, LLREF has fairly high running time. When a scheduling event occurs at time t, LLREF must determine which tasks to execute (i.e., $T_{(i)_t}$ for $1 \leq i \leq x_t$). In addition, LLREF must determine when the next event will occur (i.e., it must find $\ell_{(x_t)_t,t}$ and $\ell_{(m+1)_t,t}$). This means the tasks must be at least partially sorted.

If the tasks are sorted at the beginning of the TL-plane, the process of re-sorting during a scheduling event can be done in $O(n)$ time by using the prior sort order. Say tasks are scheduled at time s and a B or C event occurs at time t. If we consider the tasks that executed during $[s, t[$ and the ones that did not execute during $[s, t[$ separately, it is easy to see that these tasks will maintain the same relative order at time t — i.e.,

$$\ell_{(i)_s,t} \geq \ell_{(j)_s,t} \quad \text{if } 1 \leq i \leq j \leq m \text{ and}$$
$$\ell_{(i)_s,t} \geq \ell_{(j)_s,t} \quad \text{if } m < i \leq j \leq n.$$

Hence, the proper sorting order for the tasks at time t can be found by merging $\{T_{(i)_t} \mid 1 \leq i \leq m\}$ and $\{T_{(i)_t} \mid m+1 \leq i \leq n\}$. While this observation does reduce the running time, maintaining a sorted list is the most expensive portion of each scheduling event. This leads us to question whether this step is actually

necessary. How important is it that LLREF select the m tasks with *largest* remaining execution? Will any m tasks do, provided they have *non-zero* remaining execution? Below, we show that the answer to this question is "*yes*".

Recall the following result presented by Hong and Leung [10], [11]

Theorem 1 ([10], [11]). *No optimal online scheduler can exist for a set of jobs with two or more distinct deadlines for any m-processor identical multiprocessor, where $m > 1$.*

The theorem does not claim that no optimal algorithm exists if all deadlines are equal. In fact, Hong and Leung [10], [11] present an optimal algorithm when the jobs all have the same deadline. LLREF deliberately divides each job into subjobs so that (i) the work done by each subjob is proportional to the duration of the TL-plane, and (ii) at all times *all of the subjobs have the same deadline.* The key to our ability to both reduce LLREF's running time and define LLREF for sporadic tasks is the recognition that optimal multiprocessor algorithms do exist when deadlines are all equal.

Below, we make the simple observation that if x_t tasks execute at all times t within a TL-plane, the total utilization R_t decreases as time progresses.

Theorem 2. *Let τ be any task set executing on m identical processors. Let the timeline be divided into TL-planes and tasks be assigned local execution proportional to their utilization as in LLREF. Let s be any time such that $R_s \leq m$ and $r_{i,s} \leq 1$ for all tasks T_i. Let X be any x_s tasks that have positive local remaining execution at time s. Assuming the tasks in X are scheduled to execute at time s, let t_e be the time at which the next B or C event will occur. Then for any Δ such that $0 \leq \Delta \leq t_e - s$,*

$$R_{t_e} \leq R_{s+\Delta} \leq R_s. \tag{3}$$

Moreover, if $R_s < m$ then $R_{t_e} < R_{s+\Delta} < R_s$.

In other words, if $U(\tau) \leq m$ and x_t tasks execute at all times t, then the total local utilization never increases value within any TL-plane, and it constantly decreases if $U(\tau) < m$.

Proof: Because the total work done during the interval $[s, s+\Delta]$ is equal to $x_s \times \Delta$, we know that $\sum_{i=1}^{n} \ell_{i,s+\Delta} = \sum_{i=1}^{n} \ell_{i,s} - x_s \times \Delta$. Therefore, we can determine $R_{s+\Delta}$ as follows.

$$\begin{aligned} R_{s+\Delta} &= \sum_{i=1}^{n} \frac{\ell_{i,s}}{t_f - s - \Delta} - \frac{x_s \times \Delta}{t_f - s - \Delta} \\ &= \sum_{i=1}^{n} \frac{\ell_{i,s}}{t_f - s} \times \frac{t_f - s}{t_f - s - \Delta} - \frac{x_s \times \Delta}{t_f - s - \Delta} \end{aligned}$$

Note that

$$\frac{t_f - s}{t_f - s - \Delta} = 1 + \frac{\Delta}{t_f - s - \Delta}$$

Therefore,

$$\begin{aligned} R_{s+\Delta} &= R_s + \frac{\Delta(R_s - x_s)}{t_f - s - \Delta} \\ &\leq R_s \end{aligned}$$

The last step follows both when $|Active(s)| \geq m$ and when $|Active(s)| < m$. In the first case, $R_s - x_s = R_s - m \leq 0$, because $R_s \leq m$. In the second case, $x_s = |Active(s)|$. Because $r_{i,s} \leq 1$ for all T_i, we can conclude that $R_s - x_s \leq 0$ in this case as well. □

With this theorem in mind, we make the following observations, which allow us to modify LLREF as described below..

Observation 1 The total local utilization decreases *regardless of which tasks execute* provided that x_t processors execute at all times t. **Observation 2** The total local utilization at the beginning of a TL-plane $[t_0, t_f[$ is equal to $\sum_{T_i \in Active(t_0)} u_i$. Hence, if a sporadic task T_i generates a job at time $t \in [t_0, t_f[$, the local utilization just prior to t will be at most $m - u_i$.

4.1. A Simplifying Observation

As noted above, the most expensive portion of processing a scheduling event is re-sorting the tasks. Theorem 2 above allows us to remove this step from the algorithm. We propose to maintain two heaps – one heap for each type of event. For both heaps, when a task is added to the heap, its key is set to the time at which the task will trigger a scheduling event. Heap H_B contains the set of executing tasks. Task $T_i \in H_B$ triggers a B event if it executes until time $(t + \ell_{i,t})$ without interruption, where t is the current time. H_B is a min heap whose key is $(t + \ell_{i,t})$. Heap H_C is the set of active tasks that are not executing. Task $T_i \in H_C$ triggers a C event if it does not execute before time $(t_{f_t} - \ell_{i,t})$, wheret_{f_t} denotes the end of the current TL-plane. H_C is a min heap whose key is $(t_{f_t} - \ell_{i,t})$.

Note that a task T_i's key does not change as long as T_i remains in the same heap. The value of t_{f_t} remains constant within any TL-plane. If $T_i \in H_C$, then $\ell_{i,t}$ is not changing over time, so $(t_{f_t} - \ell_{i,t})$ remains constant.

Also, if $T_i \in H_B$, then $\ell_{i,t}$ decreases as t increases, so the $(t + \ell_{i,t})$ remains constant.

If a task T_i switches from one heap to the other at time t, then its new key is set to ($t_{f_t} - T_i$.key+t), where T_i.key was its key prior to switching heaps. This observation allows us to maintain proper execution without having to update ℓ at each B or C event.

The modified algorithm will only preempt a task if it is absolutely necessary to do so. If a B event occurs, any task that was executing just prior to the B event will continue to execute (on the same processor) after the B event is handled. If a C event occurs, the task that triggered the C event must execute immediately. Therefore it will preempt one of the executing tasks and execute on that task's processor. Thus, a scheduling event at time t causes at most $\nu(t)$ preemptions, where $\nu(t)$ is the number of tasks whose total local utilization increase to 1 at time t.

The algorithm LRE-TL is described in more detail in Subsection 4.3. First, though, we discuss how to handle sporadic task arrivals in the middle of a TL-plane.

4.2. Scheduling Sporadic Tasks

In this subsection, we show how to handle the arrival of a job invoked by a sporadic task and we demonstrate that this method will not allow any deadline misses provided τ is a feasible task set. Assume a task T_s invokes a job at time t_s in the middle of a TL-plane. By Theorem 2 above, we know that the total local utilization is at most $(m - u_s)$ just prior to T_s's arrival. Therefore, we can set T_s's local execution to be proportional to its utilization, just as we would at the beginning of a TL-plane. Specifically,

$$\ell_{s,t_s} = u_s \cdot (t_{f_{t_s}} - t_s). \tag{4}$$

The theorem below shows that adding T_s to the set of active tasks at time t_s will not cause any other tasks to miss their deadlines. If, in addition, T_s does not have a deadline within the current TL-plane, it will also be guaranteed to meet its deadline.

Theorem 3. *Let τ be a sporadic task set such that $U(\tau) \leq m$ and $u_{max}(\tau) \leq 1$. Assume τ is scheduled using LRE-TL as described above. Let $[t_0, t_f[$ be a TL-plane for the LRE-TL schedule of τ. Assume task T_s is not active at time t_0 and becomes active at some time $t_s \in [t_0, t_f[$. If the algorithm sets ℓ_{s,t_s} according to Equation 4, then T_s will not cause any tasks to miss their deadlines. If, in addition, $t_s + p_s \geq t_f$, then τ can be scheduled to ensure T_s will meet its deadline at time $(t_s + p_s)$.*

Proof: We first argue that T_s will not cause other tasks to miss their deadlines and then demonstrate how to ensure that T_s will not miss its deadline.

Because T_s is not active at time t_0, we know $R_{t_0} \leq m - u_s$. By Theorem 2, just prior to t_s we know that $R_{t_s - \epsilon} \leq R_{t_0}$. Therefore, upon setting the value of ℓ_{s,t_s} to $u_s \cdot (t_f - t_s)$, making $r_{s,t_s} = u_s$, we know that $R_{t_s} \leq m$. Hence, once ℓ_{s,t_s} is added to the total local remaining execution it will still be possible to meet all local execution requirements by time t_f. As with all tasks, the amount of work T_s will have completed at time t_f will be the product of T_s's utilization and the total amount of time since it arrived. Hence, in subsequent TL-planes, T_s can have its local utilization set to $r_{s,t_0} = u_s$. Because $U(\tau) \leq m$, the total utilization of all TL-planes continues to be at most m. Hence, T_s will not cause any other tasks to miss their deadlines.

It remains to demonstrate that we can schedule τ to ensure that T_s will not miss its deadline. Because we assume $t_f \leq t_s + p_s$, we know that T_s will not have a deadline before time t_f. If we ensure that there is a TL-plane that ends at time $(t_s + p_s)$, then T_s will meet its deadline. Algorithm LRE-TL has control over the TL-planes and can ensure that this will incur. Therefore, T_s will meet its deadline at time $(t_s + p_s)$. □

The correctness of LLREF (and of LRE-TL) hinges on having (i) all tasks complete their local execution by the end of every TL-plane, and (ii) every task's deadline coincide with the end of some TL-plane. These algorithms do not guarantee any timing properties *within* a TL-plane, but they can make guarantees at the boundaries *between* TL-planes.

Because sporadic tasks do not have fixed arrival times, we cannot predict the pattern of the deadlines in advance. Hence, at the beginning of each TL-plane, we determine the duration of the TL-plane by finding the earliest upcoming deadline, d_{next}, and setting t_f equal to d_{next}. Doing this will ensure that all jobs of active tasks will have deadlines that correspond with the end of some TL-plane. In addition, we must ensure that non-active sporadic tasks will not have a deadline within the TL-plane. If we ensure that no TL-plane has a duration longer than p_{min}, the minimum task period of all tasks in τ, then for any job generated by a sporadic task there must be an TL-plane break between the job's arrival and deadline.

Hence, we take the following two steps in order to handle sporadic tasks.

TL-plane Boundaries Instead of setting TL-plane boundaries to be $k \cdot p_i$, where $k \geq 0$ and $1 \leq i \leq n$, we base TL-plane boundaries on the deadlines of active jobs. Let t_0 be the beginning of some TL-plane. We

Algorithm 1 LRE-TL

```
1: if t_cur = t_f then
2:     TL-Plane-Initialize
3: else
4:     if an A event occurred then
5:         LRE-TL-A-Event
6:     if a B or C even occurred then
7:         LRE-TL-BC-Event
8: if H_B.size() > 0 then
9:     t_next ← H_B.min-key()
10:    if H_C.size() > 0 then
11:        t_next ← min{t_next, H_C.min-key()}
12: else
13:    t_next ← t_f
14: let each processor execute its designated task
15: sleep until time t_next
```

determine t_f, the end of the TL-plane as follows. Let d_{next} be the earliest upcoming deadline and let p_{min} be the shortest task period ($p_{min} = \min\{p_i \mid 1 \leq i \leq n\}$). Then $t_f = \min\{d_{next}, t_0 + p_{min}\}$. This ensures that all jobs' deadlines coincide with the end of some TL-plane. We add one new heap H_D, which contains all current deadlines, to implement this modification efficiently.

The A Event We introduce a new event, namely the A (arrival) event. When a sporadic task T_s invokes a new job it triggers an A event. During an A event, ℓ_{s,t_s} is set to $u_s(t_{f_{t_s}} - t_s)$ and T_s is added to one of the heaps (H_B or H_C). Under most circumstances, T_s will be added to H_C, the heap containing the non-executing tasks, and execution will resume without preempting any tasks. However, if $u_s = 1$, then clearly T_s must preempt an executing task. Additionally, T_s will be scheduled to execute immediately without preempting any other tasks if $x_{t_s} < m$.

We have explored LRE-TL and justified that none of the modifications to LLREF sacrifice optimality. We now describe the algorithm in more detail.

4.3. Algorithm LRE-TL

The algorithm LRE-TL is comprised of four procedures. The main algorithm determines which type of events have occurred, calls the handlers for those events, and instructs the processors to execute their designated tasks. At each TL-plane boundary, LRE-TL calls the TL-plane initializer. Within a TL-plane, LRE-TL processes any A, B or C events. The TL-plane initializer sets all parameters for the new TL-plane. The A event handler determines the local remaining execution of a newly arrived sporadic task, and puts

Algorithm 2 TL-Plane-Initialize

Require: $Active$ contains the set of all tasks that are currently active. H_B and H_C are both empty. Task T_{min} has the shortest period of all tasks (whether active and non-active).

```
1: for all tasks T_i that arrived at time t_cur do
2:     if H_D.find-key(t_cur + p_i) = NULL then
3:         H_D.insert(t_cur + p_i)
4: t_f ← t_cur + p_min
5: if H_D.min-key() ≤ t_f then
6:     t_f ← H_D.extract-min()
7: z ← 1
8: for all T_i ∈ Active do
9:     ℓ ← u_i(t_f − t_cur)
10:    if z ≤ m then
11:        T_i.key ← t_cur + ℓ
12:        T_i.proc-id ← z
13:        z.task-id ← T_i
14:        H_B.insert(T_i)
15:        z ← z + 1
16:    else
17:        T_i.key ← t_f − ℓ
18:        H_C.insert(T_i)
19: for all processors z' s.t. m ≥ z' > z do
20:    z'.task-id ← NULL
```

the task in one of the heaps (H_B or H_C). The B and C event handler maintains the correctness of H_B and H_C.

Throughout this section, we discuss executing *tasks* rather than executing *jobs*. Recall deadlines are equal to periods and every deadline coincides with the end of a TL-plane. Therefore, given any TL-plane $[t_0, t_f[$ and any task T_i, at most one job of T_i overlaps with $[t_0, t_f[$. Because all scheduling decisions are made within a TL-plane, there is no confusion about which job of each task is being scheduled — we always schedule the job that overlaps with the current TL-plane.

At all times, each active task T_i will be in exactly one task heap (H_B or H_C). Throughout this section, each task has two fields. T_i.key is the time when T_i will cause an event (the event type depends on which heap T_i is in). T_i.proc-id is the processor T_i should execute on and is valid only if T_i is in H_B. In addition, each processor z has one field, z.task-id, which is the task currently assigned to processor z.

The heaps have five methods. H.min-key() returns the value of the H's minimum key. H.size() returns the number of objects in H. H.extract-min() removes the object with the smallest key from H. H.insert(I) inserts item I into the heap. H.find-key(k) returns

Algorithm 3 LRE-TL-A-Event

Require: The sporadic task T_s that invokes a job to trigger this algorithm is not in $Active$.

1: $\ell \leftarrow u_s(t_f - t_{cur})$
2: **if** H_B.size() $< m$ **then**
3: T_s.key $\leftarrow t_{cur} + \ell$
4: T_s.proc-id $\leftarrow z\{z$ is any idling processor$\}$
5: z.task-id $\leftarrow T_s$
6: H_B.insert(T_s)
7: **else**
8: **if** $u_s < 1$ **then**
9: T_s.key $\leftarrow t_f - \ell$
10: H_C.insert(T_s)
11: **else**
12: $T_b \leftarrow H_B$.extract-min()
13: T_s.key $\leftarrow t_{cur} + \ell$
14: T_b.key $\leftarrow t_f - T_b$.key$+t_{cur}$
15: $z \leftarrow T_b$.proc-id
16: T_s.proc-id $\leftarrow z$
17: z.task-id $\leftarrow T_s$
18: H_B.insert(T_s)
19: H_C.insert(T_b)
20: **if** H_D.find-key($t_{cur} + p_s$) = NULL **then**
21: H_D.insert($t_{cur} + p_s$)

a pointer to the object whose key equal k if one exists and returns NULL otherwise. The first two methods run in $O(1)$ time and the last three run in $O(\log H$.size()) time.

LRE-TL is illustrated in Algorithm 1. At the beginning of a TL-plane, LRE-TL will initialized the TL-plane. Within a TL-plane, it will process all A, B and C events. Once the initializer or events are completed, LRE-TL instructs the processors to execute their designated tasks and sleeps until the next event occurs.

The TL-plane initializer is illustrated in Algorithm 2. It first finds t_f (lines 1 through 6), which is set to the earliest upcoming deadline, but is never larger than $(t_{cur} + p_{min})$, where $p_{min} = \min\{p_i \mid 1 \leq i \leq n\}$. Once t_f is identified, the local execution values of active tasks are initialized accordingly (lines 8 through 18). The first m tasks are inserted into H_B and the remaining tasks are inserted into H_C. The keys are set to the time when the task will trigger a B event (if the task is in H_B) or a C event (if the task is in H_C). If there are fewer than m active tasks, the idle processors' task id's are set to NULL (lines 19 through 20).

The A event handler is shown in Algorithm 3. When a sporadic task T_s arrives, this algorithm determines T_s's local remaining execution, ℓ, and adds T_s to one of the heaps (H_B or H_C). If some processor is idle, then T_s is added to H_B (lines 3 through 6). If all m processors are busy, then T_s will only preempt a task if its has zero laxity. Hence, when H_B.size() $= m$, T_s is added to H_C if $u_s < 1$(lines 8 through 10) and T_s is added to H_B and some executing task T_b is moved from H_B to H_C if $u_s = 1$ (lines 12 through 19). Before returning, T_s's deadline is inserted into H_D (lines 20 through 21).

The B and C event handler is shown in Algorithm 4. It identifies which task(s) caused the events. After this algorithm executes the following conditions hold: (i) either all processors are busy or all tasks are executing, (ii) $\ell_b > 0$ for all $T_b \in H_B$ and (iii) $r_c < 1$ for all $T_c \in H_C$. The algorithm begins by handling any B events (lines 1 through 11). Any tasks $T_b \in H_B$ with remaining execution equal to 0 are removed from H_B and replaced by a waiting task (if one exists). The algorithm then handles the C events (lines 12 through 22). Any tasks $T_c \in H_C$ with utilization equal to 1 are removed from H_C and swapped with some executing task T_b (lines 14 through 9).

4.4. Run Time Analysis

The running time of LRE-TL depends on which routine it calls. Below we establish the running time for each event handler and compare LRE-TL's running time to LLREF's running time. This comparison is summarized in Table 1.

The TL-plane initializer has a loop that iterates $O(n)$ times. During these iterations, the two heaps are populated, which takes $O(\log(n-m) + \log m)$ time. This gives an overall running time of $O(n + \log(n - m) + \log m)$. Assuming $m \leq n$, the initializer takes $O(n)$ time. LLREF's initializer operates in a similar manner and also has a run time of $O(n)$.

The A event handler initializes the sporadic task parameters, inserts the sporadic task (and possibly one other task) into one of the task heaps and adds the deadline to the deadline heap. Because the deadline heap contains the deadline of every task in the B and C heaps, the running time of the A event handler $O(\log H_D$.size()) $= O(\log n)$. As LLREF does not handle sporadic tasks, this running time cannot be compared to that of LLREF.

The running time of the B and C event handler depends on the number of active tasks, which we will denote α. If $\alpha > m$, the handler moves tasks between the two heaps, which takes $O(\log m + \log(\alpha - m))$ time. Otherwise, H_C is empty and a task is simply removed from H_B, which takes $O(\log \alpha)$ time. Hence,

Algorithm 4 LRE-TL-BC-Event

Require: Each task $T_b \in H_B$ is executing on processor T_b.proc-id and will cause a B event at time T_b.key. Each task $T_c \in H_C$ has positive local execution, is not executing and will cause a C event at time T_c.key. The current time is t_{cur} and the current TL-plane ends at time t_f.

```
1: while H_B.min-key() = t_cur do
2:     T_b ← H_B.extract-min()
3:     z ← T_b.proc-id
4:     if H_C.size() > 0 then
5:         T_c ← H_C.extract-min()
6:         T_c.key ← t_f − T_c.key + t_cur
7:         T_c.proc-id ← z
8:         z.task-id ← T_c
9:         H_B.insert(T_c)
10:    else
11:        z.task-id ← NULL
12: if H_C.size() > 0 then
13:    while H_C.min-key() = t_cur do
14:        T_b ← H_B.extract-min()
15:        T_c ← H_C.extract-min()
16:        T_b.key ← t_f − T_b.key + t_cur
17:        T_c.key ← t_f − T_c.key + t_cur
18:        z ← T_b.proc-id
19:        T_c.proc-id ← z
20:        z.task-id ← T_c
21:        H_B.insert(T_c)
22:        H_C.insert(T_b)
```

the total running time within any TL-plane is

$$\begin{aligned}\sum_{\alpha=1}^{m} \log \alpha \quad + \quad & \sum_{\alpha=m+1}^{n} (\log m + \log(\alpha - m)) \\ = \quad & O(n \log m + (n-m)\log(n-m)) \\ = \quad & O(n \log n).\end{aligned}$$

By contrast, LLREF needs to update ℓ_i for the executing tasks T_i and establish the new sort order if any task are forced to wait. Updating ℓ takes $O(m)$ time if all processors are busy and $O(\alpha)$ time otherwise. As established earlier, the tasks can be re-sorted through a simple merge, which takes $O(\alpha)$ time. This gives a total running time of $\sum_{\alpha=1}^{m} \alpha + \sum_{\alpha=m+1}^{n} (\alpha + m) = O(n^2)$.

There is one benefit of LRE-TL that is not captured in the discussion of running time – namely, the reduction overhead due to fewer preemptions and migrations. Because LLREF sorts the tasks upon every B or C event, a single event could cause m tasks to be preempted. Upon resuming execution, each of these tasks might end up on a different processor. Thus, the maximum number of preemptions and migrations within each TL-plane is $O(mn)$. By contrast, LRE-TL preempts only when absolutely necessary – i.e., only when a C event occurs. Because utilization decreases over time, the number of C events within a TL-plane is at most $(m-1)$. Hence the number of preemptions and migrations is $O(m)$.

Using the above analysis, we can determine the maximum scheduling overhead per TL-plane. We can build this overhead the schedulability test given in Equation 1. One simple way of doing this would be to evenly distribute this overhead among the tasks. Let v denote the maximum total scheduling overhead per TL-plane and κ_i denote the maximum number of time slices required to schedule any job of task T_i. Then we could distribute this overhead among the tasks and modify the task utilization accordingly. For each T_i, the modified utilization would be $u'_i = (e_i + \kappa_i(v/n))/p_i$, or $u'_i = u_i + (\kappa_i/p_i)(v/n)$. Then τ is LRE-TL schedulable on m processors if $u'_{max} \le 1$ and $U'(\tau) \le m$, where u'_{max} and $U'(\tau)$ are the maximum and total values of u'_i over all tasks T_i. This overhead accounting could be improved by allocating more of the overhead to the task T_i with the minimum ratio of κ_i to p_i provided T_i's modified utilization does not exceed 1.

5. Example

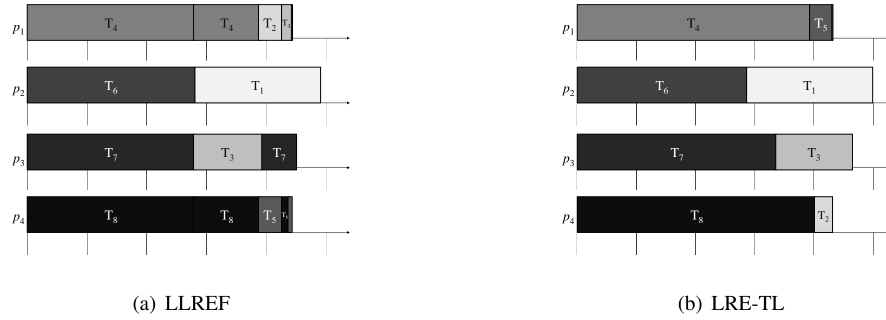
In this section we present the LLREF and LRE-TL schedule of the task set shown in Table 2, which was used in [7]. We illustrate the schedule on 4 processors for the first TL plane, $[0, 5[$.

	p_i	e_i
T_1	7	3
T_2	16	1
T_3	19	5
T_4	5	4
T_5	26	2
T_6	26	15
T_7	29	20
T_8	17	14

Table 2. Demonstration task set.

The two schedules are shown in Figure 1. Each timeline corresponds to one of the four processors. Figure 1(a) contains the LLREF schedule and Figure 1(b) contains the LRE-TL schedule.

As expected, LRE-TL has fewer preemptions and migrations. Even for this small example, the difference is quite stark. LLREF triggers 5 preemptions, 2 of



(a) LLREF

(b) LRE-TL

Figure 1. Schedule comparison.

which result in a task migration, whereas LRE-TL triggers only 1 preemption and migration[2]. (Some preemptions and migrations are not visible in the figures because the remaining execution is so small when the preemption occurs.)

There are several points in the LLREF schedule where a preemption can clearly be avoided. For example, on processor p_4, task T_5 preempts T_8, which later preempts T_5 and is once again preempted by T_8. This is an artifact of selecting the m tasks with the largest local remaining execution at every scheduling event. By contrast, LRE-TL permits both T_5 and T_8 to execute without being preempted. The only task that is preempted is T_6, which is preempted when T_1 becomes critical. Note that because $\ell_{(4)_0,0} + \ell_{(5)_0,0} > 5$, there must be at least one preemption regardless of what scheduling algorithm is used.

6. LRE-TL for Uniform Multiprocessors

Recently [12] LLREF was extended so that it can now be scheduled on uniform multiprocessors as well as identical multiprocessors. A periodic task set τ can be successfully scheduled on a uniform multiprocessor π provided the following conditions are satisfied [13]

$$\sum_{i=1}^{k} u_i \leq \sum_{i=1}^{k} s_i \text{ for } 1 \leq k < m, \text{ and} \tag{5}$$

$$U(\tau) \leq \sum_{i=1}^{m} s_i. \tag{6}$$

Chen and Hsueh [12] presented an extension of LLREF using these two conditions. If τ satisfies the above conditions for uniform multiprocessor π, then their extension to LLREF that ensures the above two conditions apply for the local utilization at all times. Their algorithm requires that at the beginning of the TL plane the m tasks with the largest local remaining execution $T_{(1)_0}, T_{(2)_0}, \ldots, T_{(m)_0}$ are scheduled to execute with task $T_{(i)_0}$ being assigned to processor s_i. As with identical multiprocessors, their algorithm triggers a B event at time t_b when some task T_b completes its execution time — i.e., $\ell_{b,t_b} = 0$. C events, however, are handled differently. A task T_c becomes critical at time t_c if T_c's utilization is equal to the speed of some processor s_i — i.e., $r_{c,t_c} = s_i$. When such an event occurs, T_c is assigned to processor s_i and s_i is removed from further consideration.

2. Because LRE-TL only preempts tasks when a C event occurs, every preemption will result in a migration. The same holds true for preemptions due to C events in LLREF.

6.1. Sporadic Tasks on Uniform Multiprocessors

We now show that LRE-TL can optimally schedule sporadic tasks in addition to periodic tasks on uniform multiprocessors. We schedule sporadic tasks in exactly the same manner as they are handled for identical multiprocessors with one important exception. If a set of k tasks has total utilization equal to the sum of the k fastest processors, then these k tasks must execute on the k fastest processors *for the remainder of the TL-plane* in order to guarantee they meet their deadlines. Hence, if a sporadic task T_s is among these k tasks, it must be scheduled on one of the k fastest processors as soon as it arrives. With this in mind, we define the following terms.

Definition 2. *Let τ be a task set that is feasible on some uniform multiprocessor π. For $1 \leq k < m$, let $Crit_k(\tau, \pi)$ be* TRUE *if the k highest-utilization tasks in τ have total utilization greater than the total speed of the $(k+1)$ fastest processors.*

$$Crit_k(\tau, \pi) = \left(\sum_{i=1}^{k} s_i \geq \sum_{i=1}^{k} u_i > \sum_{i=1}^{k+1} s_i \right).$$

If $Crit_k(\tau, \pi)$ *is* TRUE *for some* k *then tasks* T_1 *through* T_k *will need to dominate the* k *fastest processors.*

Assume that the tasks are indexed in decreasing order by utilization – i.e., task T_j *has the* j^{th} *largest utilization (ties can be broken arbitrarily). Let* $MinProc(T_j)$ *be the minimum processor speed that* T_j *can safely use – i.e., if* $j < m$ *is the smallest* $k \geq j$ *for which*$Crit_k(\tau, \pi)$ *is* TRUE. *Otherwise,*$MinProc(T_j) = m$.

We use $MinProc(T_s)$ to determine whether or not the sporadic task T_s must preempt some executing task when it invokes a new job in the middle of a TL-plane. Specifically, if $MinProc(T_s) = m$, then T_s can be handled in the same manner as described in Algorithm 3. If, however, $MinProc(T_s) < m$, then T_s must execute on one of the $MinProc(T_s)$ fastest processors when it invokes a job. By definition, exactly k tasks will have $MinProc$ values less than or equal to k. Therefore, there will be some task T_j such that $MinProc(T_j) > MinProc(T_s)$ and T_j is executing on one of the $MinProc(T_s)$ processors. The arriving sporadic task T_s can preempt any such task T_j. Provided this step is taken, LRE-TL will ensure no tasks miss their deadlines on uniform multiprocessors.

7. Conclusion

This paper presents a simple observation which has far reaching impact on the efficiency of the LLREF scheduling algorithm: Local task utilization constantly decreases within a TL-plane provided no processor idles while tasks are waiting to execute. Using this observation, we have significantly reduced the overhead of the LLREF algorithm – both by shortening the running time and by reducing the number of preemptions and migrations. We call our new algorithm LRE-TL and demonstrate that it can optimally schedule sporadic task sets on both identical and uniform multiprocessors.

We briefly discussed how scheduling overhead can be handled in determining LRE-TL schedulability. In the future, we plan to explore this approach more fully.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] S. Davari and S. K. Dhall, "On a real-time task allocation problem," in *Proceedings of the 19th Hawaii International Conference on System Science*, Honolulu, January 1985.

[3] T. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *24th Real-Time Systems Symposium*, 2003.

[4] ——, "An analysis of edf schedulability on a multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 8, pp. 760 – 768, 2005.

[5] C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997, pp. 140–149.

[6] S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, June 1996.

[7] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proceedings the* 27th *IEEE Real-Time System Symposium (RTSS)*, Los Alamitos, CA, 2006, pp. 101 – 110.

[8] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[9] M. Dertouzos, "Control robotics : the procedural control of physical processors," in *Proceedings of the IFIP Congress*, 1974, pp. 807–813.

[10] K. S. Hong and J. Y.-T. Leung, "On-line scheduling of real-time tasks," in *Proceedings of the Real-Time Systems Symposium*. Huntsville, Alabama: IEEE, December 1988, pp. 244–250.

[11] ——, "On-line scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 41, pp. 1326–1331, 1992.

[12] S.-Y. Chen and C.-W. Hsueh, "Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors," in *Proceedings of the 2008 Real-Time Systems Symposium*, pp. 147–156,.

[13] S. Funk, J. Goossens, and S. K. Baruah, "On-line scheduling on uniform multiprocessors," in *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 2001, pp. 183–192.

Multiprocessor Global Scheduling on Frame-Based DVFS Systems

Vandy BERTEN
Université Libre de Bruxelles
Fonds National de la Recherche Scientifique
vandy.berten@ulb.ac.be

Joël GOOSSENS
Université Libre de Bruxelles
joel.goossens@ulb.ac.be

Abstract

In this work, we are interested in multiprocessor energy efficient systems where task durations are not known in advance but are known stochastically. More precisely we consider global scheduling algorithms for frame-based multiprocessor stochastic DVFS (Dynamic Voltage and Frequency Scaling) systems. Moreover we consider processors with a discrete set of available frequencies.

We provide a global scheduling algorithm, and formally show that no deadline will ever be missed. Furthermore, we present simulations showing that we have an energy benefit in doing global scheduling instead of static partitioning.

1 Introduction

Nowadays, it is straightforward that energy efficiency is a crucial aspect of embedded systems where a huge number of small and very specialized autonomous devices are interacting together through many kinds of media (wired/wireless network, bluetooth, GSM/GPRS, infrared...). Moreover, we know that the uniprocessor paradigm will no longer hold in those devices. Even today, a lot of mobile phones are already equipped with several processors.

In this work, we are interested in multiprocessor energy efficient systems, where task durations are not known in advance, but are known stochastically, which means that we know the probabilistic distribution of their execution time. More precisely, we consider global scheduling algorithms for frame-based multiprocessor stochastic DVFS (Dynamic Voltage and Frequency Scaling) systems. Moreover, we consider processors with a *discrete* set of available frequencies.

In the past few years, a lot of work has been provided in multiprocessor energy efficient systems. Most work was done considering static partitioning strategies, meaning that a task was assigned to a specific processor, and each instance of this task runs on the same processor. First of those work were devoted to deterministic tasks (with a task duration known beforehand, or the worst-case is considered), such as [1, 10, 4, 5], and later probabilistic models were also considered [8, 7]. Only a little work has been provided about global scheduling, such as [3], but for deterministic systems, or [11], using some slack reclamation mechanism, but not really using stochastic information.

As far as we know, no work has been provided with global scheduling on stochastic tasks. We propose to work towards this direction. Notice that the frame-based model we consider in our work, where all tasks share the same (synchronous) period or deadline, is also used by many researchers, such as [10, 3, 7, 11]. This model attracts a lot of attention, both from industry and theoretical community. In the current state of the art of stochastic low-power multiprocessor systems, the knowledge we have about very general models is not accurate enough to allow practical implementations. This is why simple but realistic models are interesting, but can be seen as a step towards more general models which are going to be considered in a near future.

The contribution of this paper is to provide a first algorithm allowing to efficiently schedule a frame-based task set on a multiprocessor DVFS platform. We will proof that our algorithm never misses deadlines, and will show how we can save energy, compared to partitioned systems.

The paper is organized as follows: we first present the task and system model we consider. Secondly we present our algorithm, and prove its correctness. Then we provide some simulation results attesting the benefit of doing global scheduling, and finally we conclude and give some perspectives.

2 Model

We consider a set of n non parallel and non preemptive tasks $\tau = \{\tau_1, \dots, \tau_n\}$. Task τ_i requires x cycles with a probability $c_i(x)$, and its maximum number of cycles is w_i (Worst Case Execution Cycles, or WCEC). The number of cycles a task requires is not known before the end of its execution. We consider a *frame-based* model, where all tasks share the same (synchronous) period or deadline D. In the following, D denotes the frame length, and as we manage each frame independently, we denote by $t = 0$ the beginning of each frame.

Those tasks run on m identical CPUs $\Pi_1, \dots, \Pi_m$, and each of those CPUs can run at M frequencies $f_1, \dots, f_M$.

In this work, the execution time is assumed to be strictly proportional to the CPU frequency: if task τ_i takes α units of time at frequency f_k, the same task would have taken $\alpha \frac{f_k}{f_\ell}$ at frequency f_ℓ.

We consider that tasks cannot be preempted, but different instances of the same task can run on different processors, i.e., task migrations are allowed, but job migrations are not. We are interested in global scheduling techniques which schedule a queue of tasks; each time a CPU is available, it picks up the first task in the queue, choose a frequency, and run the job. We assume the system is work conserving[1], and the job order has been chosen beforehand, but in some cases, in order to ensure the schedulability, the scheduler can adapt that order. In other words, we assume that the initial task order is not crucial and can be considered to be a soft constraint. We will discuss later in this work the importance of the task order.

2.1 Examples

A simple example where this kind of system can be useful is a system where n web-cams are connected to a device with m processors. If the web-cams are synchronous, they could all send an image, let say, 24 times a second, and all those images should be processed before the next arrival. Task τ_i consists then in processing the images of the ith web-cam, which can be done on any of the m processors.

A symmetric example can also be considered: a $m-$CPUs device receives a stream containing, 24 times a second, n compressed images to decompress, and to send to n screens. In both cases, we know the distribution of the processing time, and would like to reduce as much as possible the energy consumed by the processors.

3 Global Scheduling Algorithm

In [2], we have provided techniques allowing to schedule such a task set on a *single* CPU. The main idea is to compute (offline) a function giving, for each task, the frequency to run the task based on the time elapsed in the current frame. This function, $S_i(t)$ gave the frequency at which τ_i should run if started at time t in the current frame. Here, for the sake of clarity, we are going to consider the symmetric function of S: $\hat{S}_i(d) \stackrel{def}{=} S_i(D-d)$ gives the frequency for τ_i if this task is started d units of time *before* the end of the frame.

In the uniprocessor case, we were able to give schedulability guarantees, as well as good energy consumption performance, using the worst case number of cycles, as well as the probability distribution of the number of cycles. We want to be able to provide both in this multiprocessor case, using a global scheduling algorithm. As far as we know, global scheduling algorithm on multiprocessor system using stochastic tasks, and a limited number of available frequencies, has not been considered so far.

The idea of our scheduling algorithm is to consider that a system with m CPUs, and a frame length D, is "close" to a system with a single CPU, but a frame length $m \times D$, or, with a frame length D, but m times faster. We then first compute a set of n $\hat{S}$-functions considering the same set of tasks, but a deadline $m \times D$. A very naive approach would consist in considering that when a task ends at time t, the total remaining available time before the deadline is the sum of remaining time available on each CPU, which means $D-t$ on the current CPU, and $D-t_p$ on the other ones, where at each instant, t_p represents the worst time at which the task currently running on Π_p will end, or the current time if no task is running. Then, we could use $\hat{S}_i(d)$ to choose the frequency.

Unfortunately, this simple approach does not work, because a single task cannot use time on several CPUs *simultaneously*, i.e., task parallelism is forbidden. However, if the number of tasks is reasonably greater than the number of CPUs, we think that in most cases, $\hat{S}_i(d)$ will not require to use more than the available time on the current CPU, and somehow, will let the available time on other CPUs for future tasks. And when $\hat{S}_i(d)$ requires more time than actually available, we just use a faster frequency.

Of course, we need to ensure the schedulability of the system, which cannot be guaranteed with the previous approach: for instance, at the end of a frame, we might have some slack time unusable because too short to run any of the remaining tasks. But as this time has been taken into account when we chose the frequency of previous tasks, we might miss the deadline if we do not take any precaution.

The algorithm we propose is composed of two phases, an off-line phase, and an on-line one. The off-line one consists in performing a (virtual) static partitioning, aiming at reserving enough time in the system for each task. This phase is close to what we did in [2] using the concept of *Danger Zones*. Briefly, in uniprocessor systems, the danger zone of a task τ_i starts at z_i, where z_i is such that if this task is not started immediately, we cannot ensure that this task and every subsequent tasks can all be finished by the deadline. In other words, if a task starts in its danger zone, and this task and all the subsequent ones use their WCEC, even at the highest frequency, some tasks will miss the (common) deadline.

The on-line phase uses both this pre-reservation to ensure the schedulability (but performing dynamic changes to this static partitioning), and the $\hat{S}$-functions, to improve the energy efficiency.

3.1 Virtual Static Partitioning

This first phase, which is performed offline, aims at "virtually" assigning each task to one processor — virtually meaning that a task assigned to a processor does not necessarily run on that processor — in such a way that if each task assigned to one processor takes its worst case

[1] A work conserving system is a system where tasks never wait intentionally. In other words, if a task is ready, no processor can be idle.

execution number of cycles, we can still manage to finish those tasks in a frame of length D. Figure 1, left part, shows an example of such a partitioning.

This basic idea is to put those tasks on the right side of the schedule (in light grey on Figure 1), just before the deadline, with the amount of time they would need to run in the worst case at the highest frequency. We call this grey zone the *reservation zone*. When we start a task, we remove it from this reservation zone, and start it, but in a way that a task that we run will never overlap with a task reservation even in the worst case.

The partitioning problem boils down to have n objects of size $\frac{w_i}{f_M}, \forall i \in \{1, \ldots, n\}$ that we need to put in m boxes of length D. This is actually a bin packing problem, and the optimal algorithm (giving a valid partitioning for every partitionable system) is known to be NP-hard [6].

If we denote by Γ_p the set of tasks assigned to CPU Π_p, we need to find an assignment such as:

$$\sum_{\tau_q \in \Gamma_p} \frac{w_q}{f_M} \leq D \, \forall p \in \{1, \ldots, m\},$$

meaning that no CPU has more than what it could run in the worst case, and

$$\forall p \neq q, \Gamma_p \cap \Gamma_q = \varnothing$$

meaning that a task cannot be assigned to several processors,

$$\bigcup_p \Gamma_p = \tau$$

which means all the tasks are assigned to some processor.

During the on-line phase, the partitioning will be updated by moving some tasks from a processor to another one. As long as those tasks have not started yet, they can be moved without any migration cost. But of course, a task can be moved to a processor Π_p only if there is enough space between t_p (the worst end of the task currently running on this CPU), and $D - A_p$ (the begin of the reservation zone, assuming the frame starts at time 0).

This static partitioning can be performed in many ways, but we propose in Algorithm 1 to do it as balanced as possible, by sorting tasks according to their WCEC.

Algorithm 1: Static partitioning

$A_p = 0 \, \forall p$; `// Reserved time on` Π_p
1 $\Gamma_p = \{\} \, \forall p$; `// Tasks assigned to` Π_p
2 **foreach** τ_i *descending sorted by* w_i **do**
3 $\quad q = \text{argmin}_p A_p$; `// CPU with the largest not yet assigned time`
4 $\quad$ **if** $D - A_q > \frac{w_i}{f_M}$ **then**
5 $\quad\quad A_q = A_q + \frac{w_i}{f_M}$; `//` τ_i `reservation`
6 $\quad\quad \Gamma_q = \Gamma_q \cup \tau_i$;
7 $\quad$ **else**
8 $\quad\quad$ Failed!

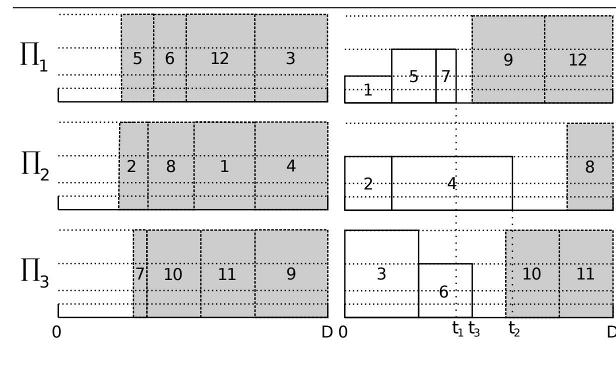
After this first step of virtual static partitioning, we can see the system as in Figure 1, left part. A_p is the length of the reservation zone on Π_p, then the length of the grey part.

Notice that it is not because we cannot manage to do this virtual partitioning that the system is not schedulable. But at least, if we manage to do so, then we can ensure that the system is schedulable, as we will show formally later in this paper. This virtual static partitioning can be computed offline, and used for the whole life of the system.

This partitioning can be done in $O(n \times \log m)$, if A_p's are stored in a heap. We also need in the off-line phase to compute the $\hat{S}$-functions, which can be done in $O(n^2 \times M \times W)$, where W is the number of samples in the distribution, using for instance the PITDVS$^{\text{closest}}$ algorithm described in [2].

Notice that the static partitioning aims at reserving the minimum amount of time required in the worst case, then this amount of reserved time corresponds to the time needed to run the worst case at the highest frequency f_M. Moreover, if the task number (written on tasks in Figure 1) gives the order in which tasks should be run, the partitioning is done sorting tasks on their worst case execution time. So the order shown on the partitioning could seem to be randomly chosen regarding to tasks number. However, tasks (virtually) assigned to a CPU can be seen as an unordered set: the only information we will need later about this task set is its total size. And except in some specific cases, tasks will be picked up by increasing task number.

Figure 1 Left: Static partitioning. Right: State of the system after having started tasks $\{\tau_1, \ldots, \tau_7\}$. Notice that reservations (light grey tasks, right aligned) correspond to worst cases, while effective tasks (white tasks, left aligned) are actual execution times, and change then from frame to frame. Vertical axis is frequency, horizontal is time. Then areas correspond to amount of computation.



3.2 On-line Algorithm

Based on the virtual static partitioning, the main idea of the on-line part is to start a task at a frequency which allows it to end before the beginning of the reserved zone on this processor. For instance, in Figure 1, τ_1 could start

on Π_1 using all the space between the beginning of the frame, and the reserved space for τ_5. But we will see situations where it would be more energy efficient to give more time for τ_1, in order to run it slower. In such cases, we can also move, for instance, τ_5 or τ_6 on Π_2, or τ_{12} to Π_3. By doing so, and because we never let a running task using the reserved time of another (not started) task, we can guarantee that, if we were able to build a partitioning in the off-line phase, no task will never miss its deadline. A formal proof of this will be given in Section 3.3. Of course, as soon as a task starts, we release the reserved time for this task.

The on-line part of the algorithm is given in Algorithm 4. We first give some explanation about two procedures we need in the main algorithm.

Remark that as we only move tasks which have not started yet, we do not need to move any context or perform any migration. The only thing we change is the information that, in the future, a job is going to start on a given processor.

3.2.1 MoveTasksOut

This procedure (Algorithm 2) aims at moving enough tasks from CPU Π_p, until enough space (the quantity s in the algorithm) is available, or no task can be moved anymore. For instance, in Figure 1, at time $t = 0$, we may want to run τ_1 on Π_1 at frequency f_2. But according to the worst case of τ_1, we do not have enough time to run this task between 0, and the beginning of the reserved area of τ_5. However, we can move τ_3 to Π_3, and τ_5 or τ_6 to Π_2.

While s units of time is not available, we take the largest task on Π_p, and put it on the CPU with the largest free space, where the free space is the space between t_q, the worst end of current job, and $D - A_q$, the begin of the reservation zone. This is of course an heuristic, since finding the optimal choice is probably an NP-hard or at least an intractable problem. We will show an improvement for this heuristic further on this paper.

This algorithm has a time complexity of $O(n \times \log m)$: the main loop can be at most run once for each task, and the argmax operation can have a complexity of $\log m$.

3.2.2 MoveTaskIn

This procedure (Algorithm 3) aims at trying to move a task τ_i assigned to a CPU Π_q to the CPU Π_p. The main idea is that we first move out as many tasks as needed from Π_p (line 1), until we have enough space to move τ_i in (lines 2 to 5). If we have not managed to get enough space, *false* is returned (line 7). Again, this algorithm is an heuristic, and is not always able to find a solution, even whether such a solution exists.

For instance (see Figure 1, right part), at the end of τ_7, we would like to start τ_8 on Π_1. But neither τ_9 nor τ_{12} can be moved on another CPU, so our algorithm fails in finding a solution. However, a smarter algorithm could find out that by swapping τ_8 and τ_9, τ_8 would be able to start on Π_1. Notice that giving a solution in any solvable case is probably also an NP-hard or at least an intractable problem.

The procedure we give here is quite naive, and not very efficient. We keep it simple for the sake of simplicity, but we present some improvements afterwards. The naivety of this algorithm does not affect the schedulability at all: it just makes the system to be forced more often to accept tasks order changes, which might degrade the energy efficiency (S-functions are computed according to the given order), and the user satisfaction, if its preferences are often not respected.

Algorithm 2: MoveTasksOut

```
Data: processor Π_p, current time t, space to free s
// Move out tasks from Π_p until s units of time are free from t.
1 while D − A_p − t < s do
2     τ_i = next task in Γ_p (sorted by decreasing w_i);
3     if No such τ_i then
4         return false;
5     q = argmax_{r≠p} D − A_r − t_r; // CPU with the maximal amount of available space
6     if D − A_q − t_q ≥ w_i/f_M then
          // Enough place to move τ_i on Π_q
7         Γ_p = Γ_p \ τ_i ; A_p −= w_i/f_M;
8         Γ_q = Γ_q ∪ τ_i ; A_q += w_i/f_M;
9 return true;
```

Algorithm 3: MoveTaskIn

```
Data: processor Π_p, current time t, task τ_i,
Result: true if τ_i can be moved on Π_p, false otherwise
// Move enough tasks from Π_p to let τ_i running
1 if MoveTasksOut(Π_p, t, w_i/f_M) then
      // We know that D − A_p − t ≥ w_i/f_M
2     let q be such as τ_i ∈ Γ_q;
      // Move τ_i from Π_q to Π_p
3     Γ_q = Γ_q \ τ_i ; A_q− = w_i/f_M;
4     Γ_p = Γ_p ∪ τ_i ; A_p+ = w_i/f_M;
5     return true;
6 else
7     return false;
```

The complexity of `MoveTaskIn` is dominate by the complexity of `MoveTasksOut`, and is then also $O(n \times \log m)$

3.2.3 Main algorithm

Here are the main steps of the procedure given in Algorithm 4, which is called each time a CPU (say Π_p) is available, at time t, with τ_i the next task to start. This procedure will always start a task at a speed guaranteeing deadlines, but not necessarily τ_i.

- line 1: We first evaluate d, the remaining time we have for $\tau_i, \dots, \tau_n$: if t_q is the worst time where Π_q is going to be available (the time of the last start, plus the worst case execution time of the current task at the chosen frequency), we have:

$$d = (D-t) + \sum_{q \neq p}(D - t_q) = mD - \left(t + \sum_{q \neq p} t_q\right).$$

- line 2: Let $f = \hat{S}_i(d)$, the frequency chosen for τ_i in the single CPU model with d units of time before the deadline. We are going to check if we can use this frequency (we assume this frequency to be a "good" one from the energy consumption point of view).

- line 3–6: If τ_i was not assigned to Π_p, we first try to move it to Π_p (Algorithm 3). If we have enough space on Π_p, the situation is easy. Otherwise, we need to move some tasks out from Π_p, in order to create enough space.

- line 5: If we cannot manage to make enough space, then we are not able to start τ_i right now. We try then the same procedure for τ_{i+1}, but we need to left-shift $\hat{S}-$functions of $\frac{w_i}{f_M}$. This is not required from the schedulability point of view (we ensure the schedulability by controlling the available time), but we guess it will improve the energy consumption. For the same reason, we will need to right-shift functions of the same amount when τ_i starts, because we have one task less to run after τ_i.

  This improvement is not presented here, but we have implemented it in the simulation we present in this paper. It requires to be done carefully, because we might have several swapped tasks.

- line 9: If we succeeded, we try to move as many tasks as possible from Π_p to other CPUs (Algorithm 2), until we have enough space to start τ_i at f, or no task can be moved anymore. We then start τ_i either at f, or at the smallest frequency allowing to run τ_i in the space we manage to free (line 10). As τ_i was assigned to Π_p (possibly after some changes), we are at least sure that we can start τ_i at f_M.

Notice that when `StartTask` is invoked, it is always possible to run a job, and therefore, we will never consider τ_{n+1} in Algorithm 4, line 5 (see next section for a proof).

The complexity of `StartTask` is a little bit complex, because this function is recursive. Let first compute the complexity when we do not need to invoke `StartTask` at line5. We have m (sum at line 1) + $\log(m)$ (line 2) + $n \log m$ (`MoveTaskIn` at line 4) + $n \log m$ (`MoveTasksOut` at line 9), then $O(m + n \log m)$.

If we do invoke `StartTask` recursively, then in the worst case, we have a depth of n calls. In this case, lines 1 to 5 are run n times ($O(n \times (m + n \log m))$, and lines 7 to 12 only once ($O(n \log m)$). Then, in total ($O(n \times (m + n \log m))$.

Algorithm 4: StartTask

Data: Processor Π_p, time t, task τ_i

1 $d = m \times D - \left(t + \sum_{q \neq p} t_q\right)$; `// Available time on the system`
2 $f = \hat{S}_i(d)$; `// Freq. we want to run` τ_i
3 **if** $\tau_i \notin \Gamma_p$ **then**
  `//` τ_i `is not on` Π_p`, we try to move it in`
4 **if not** `MoveTaskIn`(Π_p, t, τ_i) **then**
5   `StartTask`(Π_p, t, τ_{i+1});
6   **return**;
`// We have now` $\tau_i \in \Gamma_p$
7 $A_p -= \frac{w_i}{f_M}$; `// Release` τ_i `reservation`
8 $\Gamma_p = \Gamma_p \setminus \tau_i$;
`// Try to remove enough tasks (if needed) from` Π_p `to allow` τ_i `to run at the desired speed` f
9 **if not** `MoveTasksOut`(Π_p, t, $\frac{w_i}{f}$) **then**
  `// Not enough time to run` τ_i `at` f
  `// We know that` $D - A_p - t < \frac{w_i}{f}$
10 $f = \left\lceil \frac{w_i}{D - A_p - t} \right\rceil_{\mathcal{F}}$;
11 $t_p += \frac{w_i}{f}$; `// Worst end time for` τ_i
12 Start τ_i at f ;

3.3 Correctness

In this section, we will show the correctness of this algorithm, meaning that the on-line algorithm does not jeopardize the schedulability provided by the off-line phase. We will need two proofs for this: first, we will show that if we are able to obtain a virtual static partitioning, then we will always meet the deadline. Then, we will show that the algorithm "`StartTask`" runs all the tasks.

We remind the reader that t_q is the worst end time of tasks running on Π_q. If no task is running on this CPU, t_q is actual end time of the last task which ran on Π_q, 0 if no task has ever started on this CPU.

We first provide a definition:

Definition 1. *Let A_q the "reserved time" on Π_q, i.e. $\sum_{p:\tau_p \in \Gamma_q} \frac{w_p}{f_M}$.*

*A **correct state** is a state where, on each CPU Π_q, the worst end t_q is always lower than the begin of the reserved*

zone $[D - A_q, D]$, or, more formally, a state is said to be correct iff

$$t_q \leq D - A_q \; \forall q \in [1, \ldots, m].$$

We will show that, starting from a correct state, one step of the algorithm `StartTask` (or one call to Algorithm 4) will reach another correct state.

Lemma 1. *Algorithm 4 keeps states correct.*

Proof. In order to show the lemma, we will prove that, if before we call the algorithm, we have $t_q \leq D - A_q, \forall q$, this condition is still respected at the end. We will denote by A'_q the value of A_q before we call the algorithm, and by A''_q this value after the call. We then have to show that

$$t_q \leq D - A'_q \Rightarrow t_q \leq D - A''_q.$$

We first show that `MoveTasksOut` (Algorithm 2) and `MoveTaskIn` (Algorithm 3) respect this property.

MoveTasksOut (Algorithm 2, page 4). We can show that any iteration of the **while** loop keeps the property. The only lines that change A_q are the line 7 and 8. Here, we denote by A'_q (resp. A''_q) the value of A_q at the beginning (resp. the end) of the loop.

For A_q, we have from line 6 that $D - A'_q - t_q \geq \frac{w_i}{f_M}$. From line 8, we have $A''_q = A'_q + \frac{w_i}{f_M}$. Then,

$$D - A''_q + \frac{w_i}{f_M} - t_q \geq \frac{w_i}{f_M},$$

and therefore, $t_q \leq D - A''_q$.

For A_p, as we remove a task from Π_p, the condition remains obviously true. Remark that, if the function returns *true*, we can easily see that $D - A_p - t \geq s$.

MoveTaskIn (Algorithm 3, page 4). The proof is very similar to the previous one. We first call `MoveTasksOut`, which preserves the condition, as we have shown above. And with the same way as before, we can show that lines 3 and 4 also preserve the condition, because we run them if and only if $D - A_p - t \geq \frac{w_i}{f_M}$.

StartTask (Algorithm 4, page 5). The first part of the algorithm (lines 1 to 6) preserves the condition $t_q \leq D - A_q$ for sure: lines 1 and 2 do not change any value in the condition, `MoveTaskIn` preserves the condition, and if we invoke `StartTask`, we return right after the call.

When we are at line 7, we know that τ_i, the task we want to start, is on Π_p, the CPU which has just been released, or $\tau_i \in \Gamma_p$. As $A''_p = A'_p - \frac{w_i}{f_M}$.

Notice that as t corresponds to the time at which Π_p has just been released (or at the begin of the frame if $t = 0$), we have $t = t_p$.

Line 7 preserves the property, because we reduce A_p (if $t_p \leq D - A_p - \frac{w_i}{f_M}$, then $t_p \leq D - A_p$), as well as `MoveTasksOut` at line 9.

At line 9, we have two cases that we will consider separately. Notice that, as we stated before, `MoveTaskIn`$(\Pi_p, t, \frac{w_i}{f})$ returns true if and only if $D - A_p - t \geq \frac{w_i}{f}$. Then we can see the line 9 as the test 'if$(D - A_p - t < \frac{w_i}{f})$'.

Let t'_p be the value of t_p just before the test. By hypothesis, we know that $t'_p \leq D - A_p$ (A_p does not change in this part).

If $D - A_p - t'_p \geq \dfrac{w_i}{f}$, we have $t''_p = t'_p + \dfrac{w_i}{f}$, and then $t''_p \leq D - A_p$, which validates the first case.

If $D - A_p - t'_p < \dfrac{w_i}{f}$, then line 10 makes that $f \geq \dfrac{w_i}{D - A_p - t}$, and then $D - A_p - t'_p \geq \dfrac{w_i}{f}$, and we can then apply the same proof as above.

We now have finished the proof: if $t_p \leq D - A_p$ is true before we call `StartTask`, this condition is still true at the end of the procedure.

Remark that we do not need to make any hypothesis on $S_i(t)$, except that this function always return an allowed frequency. □

Lemma 2. *All tasks are started by the algorithm.*

Proof. The reason why we need to proof this is that we sometime skip a task, if we are not able to start it right now without violating any reservation.

We first have to do the hypothesis that `StartTask` is always called with the task with the smallest index that has not started yet. But of course, `StartTask` could possibly call recursively itself with a task with an higher index.

We can consider separately three cases:

- The task τ_i is already allocated to CPU Π_p. Then τ_i can for sure start on Π_p right away, possibly at the highest speed;
- The task τ_i is not on CPU Π_p, but we can move it there. So we are in the same situation as the first case;
- The task τ_i is not on CPU Π_p, and we cannot move it there. We will now consider this last case.

If it is impossible to move τ_i on Π_p, it is obviously because there is at least one task already reserved on Π_p. And as at the first level of `StartTask`, i is the smallest index of the not started tasks, the reserved tasks have all an index larger than i. Let j be the smallest index of the tasks reserved on Π_p.

As we cannot move τ_i on Π_p, we call `StartTask` with τ_{i+1}. If $i + 1 = j$ or τ_{i+1} can be moved on Π_p, then we can start a task. Otherwise, we try with $\tau_{i+2}, \tau_{i+3}, \ldots$, and we are then sure to reach τ_j at some point, or to start a task with an index between i and j. □

Theorem 1. *If a virtual static partitioning can be found, then algorithm `StartTask` runs all jobs, and meets all deadlines.*

Proof. The proof is a direct consequence of the two previous lemmas. The initial state, just after the virtual partitioning has been performed, is correct: we have $t_q = 0 \, \forall q \in [1, \ldots, m]$, and if the partitioning is correct, then $D \leq A_q \, \forall q \in [1, \ldots, m]$.

We can also see that if the final state (when all task have finished) is correct, then we have not missed any deadline: in the final state, $A_q = 0 \, \forall q \in [1, \ldots, m]$. Then, if the state is correct, we have $t_q \leq D \, \forall q \in [1, \ldots, m]$, where t_q is the end time of the last task running on Π_q. And obviously, if all last tasks have finished before the deadline, no deadline has been missed.

As the initial state is correct, the algorithm preserves the correctness, and all tasks are run by the algorithm, then the final state will be reach, and will be correct. Then all tasks meet their deadline. □

3.4 Algorithm Improvement

A drawback of the algorithms we present here is that in some cases, we are not able to start the task in the given order, and then accept to swap the order in which tasks are started. But our $\hat{S}-$functions are computed to be efficient in the case we respect the order. Unfortunately, we have some cases where we cannot avoid intrinsically this task swapping. But we can however improve the function `MoveTaskIn` and `MoveTasksOut` in order the reduce the cases where we need to change the task order. We show here how to do that for `MoveTaskIn` can be improved, but a similar modification can be done for `MoveTasksOut`.

The idea is that if we cannot manage to free enough space on the target CPU, then we can try to swap the task we want to move on this CPU with one of the task already there.

Algorithm 5: Improvement for Algorithm 3 (`MoveTaskIn`)

function `CanSwap` (τ_i, τ_j)
  p is such that $\tau_i \in \Gamma_p$;
  q is such that $\tau_j \in \Gamma_q$;
  return $\left(D - t_p - \left(A_p - \frac{w_i}{f_M}\right) \geq \frac{w_j}{f_M}\right.$
    and $\left. D - t_q - \left(A_q - \frac{w_j}{f_M}\right) \geq \frac{w_i}{f_M}\right)$;

function `SwapTasks` (τ_i, τ_j)
  p is such that $\tau_i \in \Gamma_p$;
  q is such that $\tau_j \in \Gamma_q$;
  $\Gamma_p = \Gamma_p \setminus \tau_i \cup \tau_j; A_p = A_p - \frac{w_i}{f_M} + \frac{w_j}{f_M}$;
  $\Gamma_q = \Gamma_q \setminus \tau_j \cup \tau_i; A_q = A_q - \frac{w_j}{f_M} + \frac{w_i}{f_M}$;

Those lines replace line 7 in Alg. 3 (`MoveTaskIn`):
foreach $j : \tau_j \in \Gamma_p$ **do**
  if `CanSwap` (τ_i, τ_j) **then**
    `SwapTasks` (τ_i, τ_j);
    return true;
return false;

4 Simulation Results

In this section, we will present several simulations we performed in order to evaluate the interest of doing global scheduling of such frame-based multiprocessor platforms, in terms of energy savings. The simulator has been written in c++, by the authors of this paper. Before we really evaluate our scheduling algorithm, we will first study how the task order can influence the gain in energy. We then compare the energy consumed by a platform with static partitioning with the same platform and job characteristics, when global scheduling is allowed.

In the plots we present here, the load of a system (horizontal axis) is computed in this way: We first define $D^{\min}$ as the minimal deadline that any system can reach:

$$D^{\min} = \frac{m}{f_M} \sum_i w_i.$$

We then define the load of a system as the ratio between the actual deadline D and the minimal deadline $D^{\min}$:

$$\frac{D}{D^{\min}} = \frac{D f_M}{m \sum_i w_i}.$$

Of course, on multiprocessor systems, a load of 1 is very rare to reach. A load of 10% does not mean that the system is busy at 10%, but that, if we neglect switching times, and the system only uses f_M, it would be busy at 10%.

We have consider two task sets. For the first one, we consider 32 tasks with normal distribution for the length (except that we truncate the tail in order to have a known WCEC, and reject the negative values).

On another side, we consider real traces which have been collected in the National Taiwan University, CSIE department, on devices decoding video streams. See [2] for more explanation about those traces. In the simulation we present here, we have 18 such tasks for Figures 3, 6 and 7, and 100 tasks for Figure 8.

In the following, we will mainly consider the Intel XScale CPU, but will also present some results on the Intel StrongArm SA-1100. The XScale provides 5 frequencies ranging from 150 to 1000 MHz, with a consumption going from 80 mW at 150 MHz to 1.6 W at 1000 MHz. More details can be found for instance in [9]. The StrongArm has 11 frequencies, between 60 and 206 MHz.

4.1 Impact of the Task Order

In this section, we compare several sorting criteria defining different task orders. We did not conduct a full study on this subject, and let this to further research, but we wanted to see how simple criteria could impact on the performance. We experimented many methods, but we only present here a few of them (others did not show significant differences).

We evaluate several task characteristics in order to sort tasks. Here are the names given in the figures legend:

- rand: tasks are randomly ordered;

Figure 2

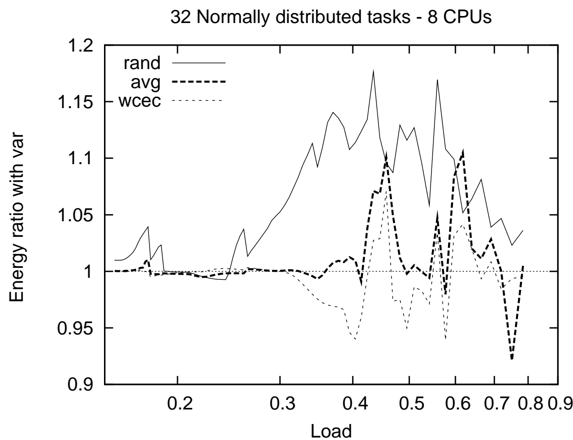
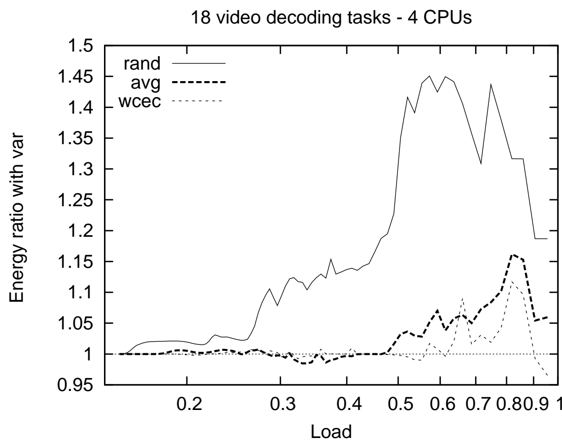


Figure 3



- wcec: tasks are sorted on decreasing worst case execution cycles;
- avg: tasks are sorted on decreasing average number of cycles;
- var: tasks are sorted on decreasing variance.

Intuitively, we may think that tasks with a smaller variance (or, in other words, with a better knowledge about their execution time) should be put at the end of a frame. This way, the scheduler will have a better chance to finish the last task very close to the deadline, and have then a slower average speed, which is known to be more efficient.

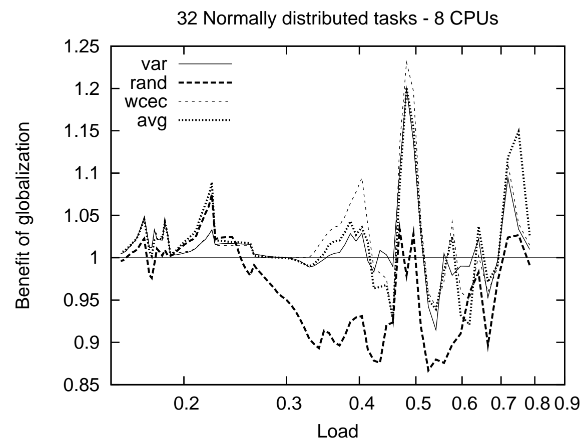
In Figures 2 (32 normally distributed (truncated) tasks on 8 XScale) and 3 (18 tasks with realistic distribution, on 4 XScale), we show two systems, where we compare for various loads the ratio between the energy consumption for var, and with the three other metrics. A value higher than one means then that, at that load, this metric consumes more energy, and then performs worse.

At a first look, we can see that rand performs worse that var on both platforms (using up to 15% more energy in the first plot, and up to 45% in the second case). The two other metrics (var and avg) do not show significant difference with the normal distribution (Fig. 2), but show a 10% loss in the realistic case, at high load.

The erratic aspect of the plots, especially the jumps we can observe in Fig. 2 around 0.2, can be explained quite simply, as we did for uniprocessor systems in [2]. The speed at which the first m tasks are started in a frame only depends upon the characteristics of the system, contrary to the speed of subsequent tasks, which will strongly depends on the time previous tasks actually took. Then, a slight change in the deadline, for instance, could cause one of the m first tasks to start at a higher speed, and has then a large impact on the system behavior.

4.2 Benefit of Globalization

Figure 4



In the next few plots, we will try to see in which configuration we have any benefit in global scheduling. As a first plot, we present in Figure 4 the same system as in Figure 2, but present another metric. Here, we show the ratio between the energy consumed with static partitioning and the energy with global scheduling. So the higher, the better is to use global scheduling. We show that for the four sorting metrics we presented above.

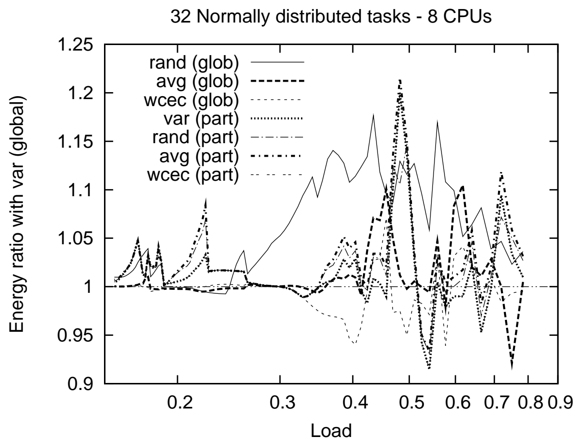
In the first plot (Fig. 4 ; 32 normally distributed tasks on 8 XScale CPUs), we can see that with rand, static partitioning performs better than global scheduling. But for other metrics, except at high load where it seems to be quite unpredictable which strategy is better, global scheduling saves always energy.

This plot does *not* show that static partitioning and random ordering is better that other strategies. It shows that if the order is given and random, then we should not use global scheduling for this task set.

In order to better show how global scheduling performs on this task set, we will show in Figure 5 the ratio between all the combinations (a task ordering) and (global

or partitioning), and the global strategy on var. From this figure, we can see that any couple task ordering/strategy behaving better than var/global (meaning having most of its point below 1) is also a global strategy.

Figure 5



We also present here simulations with other system and task parameters. Figure 6 presents the same system as in Figure 3 (but again with another metric). In Figure 7, we show the same task set, but on a StrongArm CPU. In Figure 8, we can see a much bigger system 100 (realistic) tasks, on a platform with 32 XScale CPUs.

All those simulations point out that when the order of tasks is arbitrary (e.g. rand), it sounds better to do static partitioning. But when the order is better, then most of the time, we gain several percents of energy by doing global scheduling. But not always for high loads: we observe very often that, for some high loads, static partitioning performs better than global.

It could seem counter-intuitive that by being more rigid (i.e. static partitioning), we can be more efficient. But several phenomena happen in our strategy, and can explain this behavior.

First of all, in static partitioning system, when a task ends, we know exactly how much time we can give to the next tasks allocated to the same CPU. This is the remaining time before the deadline. But in the global scheduling, we estimate the remaining time we will have for all tasks which have not started yet. And we cannot be very accurate in this estimation, because when a tasks finishes, $m - 1$ tasks could be still running, and we do not know when they will be over. We have then a less accurate knowledge about the system, which could lead to unlucky decisions.

Another phenomenon can be better understood by an example. Let us imagine a system with 3 identical tasks (τ_1, τ_2 and τ_3), and 2 CPUs (Π_1 and Π_2). If the variance is quite small, the scheduler on the equivalent uniprocessor system would choose to give each task approximately a third of the time space. On the dual processor system, the scheduler will then try to run τ_1 up to $2D/3$ on Π_1, and will make the same for τ_2 on Π_2. Then when the first of them ends, we have to start τ_3, but we cannot use $2D/3$ units of time, because only half of this is available on the released CPU, the other half being soon available on the other one. And then we need to speed up the CPU on which τ_3 runs, while the other CPU will be idle for a third of the frame length.

In the static partitioning case, we would have given all the frame to one job on the first CPU, and would have split the frame into two equal parts on the second CPU. And obviously, this scenario consumes less energy, because we use a more constant frequency on one processor, and a lower frequency on the other one.

Figure 6

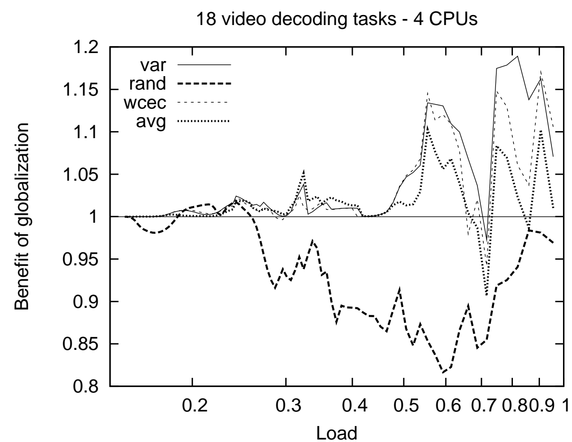
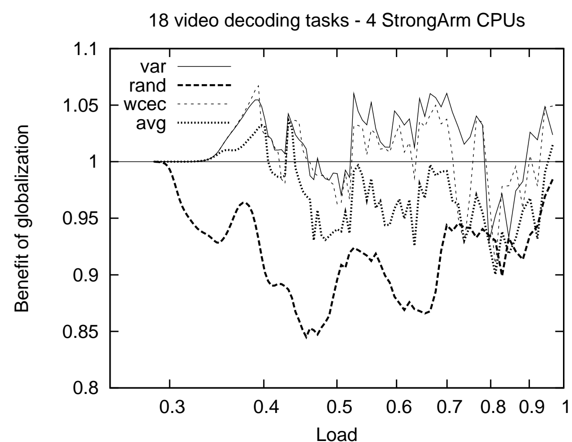


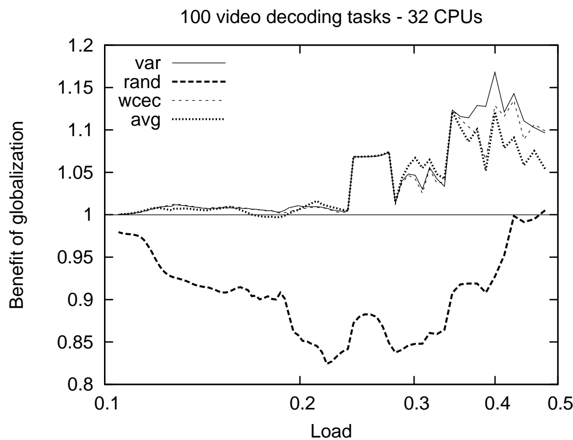
Figure 7



5 Conclusion and Future Work

In this paper, we have provided a first global scheduling algorithm for multiprocessor stochastic low-power frame-based systems. We extended uniprocessor results, and we formally proved that, if a static partitioning can be found

Figure 8



by any algorithm, then our scheduling algorithm will meet all deadlines.

Furthermore, we have shown through many simulations that this global algorithms can gain a lot of energy compared to static methods, especially if the tasks are smartly ordered. We have shown several simulations where we can gain up to 20% by doing global scheduling instead of static partitioning.

Lastly, our algorithm has a very reasonable on-line and off-line complexity, and we strongly believe that it would be easy to implement.

As a future work, here are a few points we want to look deeper, allowing to improve the energy consumption, or the number of systems we are able to schedule.

- If we accept to change the frequency during the execution of tasks, we can use the continuous model to obtain a frequency f, and use two frequencies $\lceil f \rceil_{\mathcal{F}}$ and $\lfloor f \rfloor_{\mathcal{F}}$ to "emulate" this f, where $\lceil f \rceil_{\mathcal{F}}$ (resp. $\lfloor f \rfloor_{\mathcal{F}}$) stands for the smallest frequency above (resp. largest below) f.
- Several steps require to solve NP-hard problems by using some heuristics: Static partitioning (Algorithm 1), `MoveTaskIn` (Algorithm 3), and `MoveTasksOut` (Algorithm 2). The efficiency of the first one improves the number of systems we can accept to schedule, the second one, the number of tasks we will need to swap (not run in the right order), and the third one, how close we can stay from the uniprocessor algorithm. We may try to further improve those three algorithms.
- In order to reduce leakage or static energy consumption, we could turn off CPUs if they are not needed anymore before the end of the frame.
- We believe that if jobs are parallelizable, we can still gain more energy by splitting them on several CPUs. But only a few research has been done on this subject so far, and we think it is worth to be deeper studied.

References

[1] AYDIN, H., AND YANG, Q. Energy-aware partitioning for multiprocessor real-time systems. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 113b.

[2] BERTEN, V., CHANG, C.-J., AND KUO, T.-W. Discrete frequency selection of frame-based stochastic real-time tasks. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (Kaohsiung, Taiwan, August 2008), RTCSA2008, IEEE Computer Society, pp. 269–278.

[3] CHEN, J.-J., HSU, H.-R., CHUANG, K.-H., YANG, C.-L., PANG, A.-C., AND KUO, T.-W. Multiprocessor energy-efficient scheduling with task migration considerations. In *ECRTS'04: Proceedings of the 16th Euromicro Conference on Real-Time Systems* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 101–108.

[4] CHEN, J.-J., AND KUO, T.-W. Energy-efficient scheduling of periodic real-time tasks over homogeneous multiprocessors. In *PARC* (September 2005), pp. 30–35.

[5] CHEN, J.-J., AND KUO, T.-W. Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In *ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 13–20.

[6] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[7] MISHRA, R., RASTOGI, N., ZHU, D., MOSSÉ, D., AND MELHEM, R. Energy aware scheduling for distributed real-time systems. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 21b.

[8] XIAN, C., LU, Y.-H., AND LI, Z. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *DAC '07: Proceedings of the 44th annual conference on Design automation* (New York, NY, USA, 2007), ACM, pp. 664–669.

[9] XU, R., MELHEM, R., AND MOSSÉ, D. A unified practical approach to stochastic DVS scheduling. In *EMSOFT'07: Proceedings of the 7th ACM & IEEE international conference on Embedded software* (New York, NY, USA, 2007), ACM, pp. 37–46.

[10] YANG, C.-Y., CHEN, J.-J., AND KUO, T.-W. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *DATE'05: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 468–473.

[11] ZHU, D., MELHEM, R., AND CHILDERS, B. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *RTSS'01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 686–700.

Author Index

Bünte, Sven 35
Balbastre, Patricia 115
Baruah, Sanjoy 23
Berten, Vandy 169
Burns, Alan 23, 75, 97, 115
Cassé, Hugues 55
Crespo, Alfons 115
Davis, Robert I. 23, 97
Dorin, François 13
Espes, David 67
Fabre, Jean-Charles 137
Fisher, Nathan 127
Funk, Shelby 159
Gonzalez Harbour, Michael 97
Goossens, Joël 13, 169
Hardy, Damien 45
Heydemann, Karine 55
Killijian, Marc-Olivier 137
Kirner, Raimund 35
Le Berre, Tanguy 147
Lu, Caroline 137
Mahfoudh, Saoucene 85
Mammeri, Zoubir 67
Masmano, Miguel 115
Mauran, Philippe 147
Minet, Pascale 85
Nadadur, Vijaykant 159
Ozaktas, Haluk 55
Padiou, Gérard 147
Puaut, Isabelle 45
Quéinnec, Philippe 147
Queudet, Audrey 107
Richard, Michaël 13
Richard, Pascal 13
Ripoll, Ismael 115
Rochange, Christine 55
Rothvoß, Thomas 23
Sarni, Toufik 107
Shi, Zheng 75
Valduriez, Patrick 107
Zabos, Attila 97
Zolda, Michael 35