CLASSY: a Clock Analysis System for Rapid Prototyping of Embedded Applications on MPSoCs

Xin An, Sarra Boumedien, Abdoulaye Gamatié, Eric Rutten

CLASSY: a Clock Analysis System for Rapid Prototyping of Embedded Applications on MPSoCs

Xin An, Sarra Boumedien, Abdoulaye Gamatié , Éric Rutten

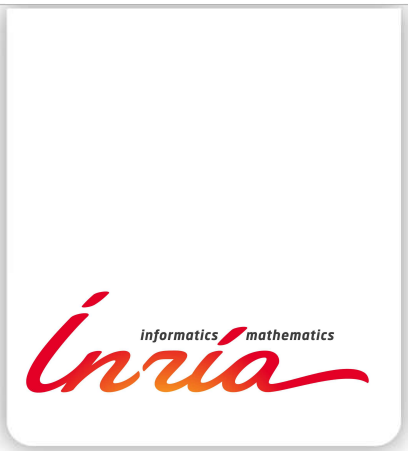
RESEARCH REPORT

N° 7918

March 2012

Project-Teams Sardes and DaRT

ISSN 0249-6399 ISRN INRIA/RR--7918--FR+ENG



CLASSY: a Clock Analysis System for Rapid Prototyping of Embedded Applications on MPSoCs

Xin An, Sarra Boumedien, Abdoulaye Gamatié , Éric Rutten

Project-Teams Sardes and DaRT

Research Report n° 7918 — March 2012 — 23 pages

Abstract: This paper presents an abstract clock-based reasoning for the rapid prototyping of embedded applications executed on multiprocessor systems-on-chip (MPSoCs). In this framework, a synchronous multi-clock modeling of application behaviors is considered. The scheduling of these applications on execution platforms composed of processors operating at various frequencies is described and analyzed with the proposed clock modeling. As in the static scheduling of synchronous dataflows (SDFs), requirements for admissible schedules are investigated, which come not only from expected application behavior, but also from execution platform properties. An algorithm is proposed to construct admissible schedules respecting the identified requirements. It is then adapted to support the synthesis of admissible schedules for adaptive system behaviors including, e.g., dynamic frequency changing or task migration. The modeling, analysis and algorithms presented in this paper have been implemented in a prototype tool named CLASSY (standing for *CLock Analysis SYstem*), providing also a way to visualize design results. The proposed approach provides a fast and cost-effective means to define correct-by-construction systems and simplify the design space exploration of complex embedded systems.

Key-words: MPSoC design, embedded applications, abstract clocks, synchronous approach, simulation and analysis

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

CLASSY: un système d'analyse d'horloges pour le prototypage rapide d'applications embarquées sur MPSoC

Résumé : Ce rapport présente un raisonnement à base d'horloges abstraites pour le prototypage rapide d'applications embarquées sur des systèmes-sur-puce multiprocesseurs (en anglais, *multi-processor system-on-chip* – MPSoCs). Dans ce cadre, une modélisation synchrone multi-horloge des comportements d'applications est considérée. L'ordonnancement de ces applications sur des plates-formes d'exécution composées de processeurs fonctionnant à des fréquences différentes est décrit et analysé à l'aide d'horloges. Comme dans l'ordonnancement statique de specifications flot de données synchrones, des critères d'admissibilité doivent être satisfaits non seulement par les comportements des applications, mais aussi par ceux des plates-formes d'exécution. Un algorithme est proposé pour construire des ordonnancements admissibles respectant les critères identifiés. Il est ensuite adapté pour supporter la synthèse d'ordonnancements admissibles pour comportements adaptatifs, incluant par exemple, des changements dynamiques de fréquences ou des migrations de tâches. La modélisation, l'analyse et les algorithmes présentés dans ce rapport ont été mis en œuvre dans un prototype d'outil nommé CLASSY (pour *CLock AnalySis SYstem*), fournissant également un moyen de visualiser les résultats. Notre approche fournit un moyen rapide et peu coûteux pour définir des systèmes corrects-par-construction et simplifier l'exploration de l'espace de conception pour systèmes embarqués complexes.

Mots-clés : Conception de système-sur-puce multiprocesseurs, applications embarquées, horloges abstraites, approche synchrone, simulation et analyse

Contents

1 Introduction

The size and sophistication of embedded systems have been rapidly growing with the explosion of digital technology in several areas such as transport, telecommunications, defense and multimedia. This technological progress has been achieved at the cost of an increased complexity of systems, and this will continue in the future for several reasons. First, embedded systems provide a wide range of services to their users thanks to a high integration of functionalities in digital integrated circuits. This trend is typically observed in multimedia phones, which allow to call, watch video, play music, browse Internet, etc. It is made possible by the ability to manufacture and increase the number of extremely small transistors in circuits. This number even exceeds one billion in a single chip today. Second, the increase of the number of processors in embedded systems enhances their performances. It also enables to distribute computations over multiple processors executing at different frequency and voltage levels in order to minimize the energy consumption. These computing resources offer the ability to replicate or migrate functions for reliability and quality of service.

Multiprocessor systems-on-chip (MPSoCs) are the most common solutions for the implementation of modern embedded applications, e.g. smart phones or tablets. They consist of several processing and memory units, interconnected by an on-chip dedicated structure [3]. If they offer a great opportunity to meet the aforementioned embedded system requirements, they require adequate design methodologies in order to reduce the complexity of design space exploration and to increase the productivity of developers.

As a solution to this demand, several existing approaches consider high-level models [14] [13] for expressing concurrency in order to efficiently exploit the parallelism and scheduling. These models are also used for rapid performance evaluation. This is very important for addressing large design spaces compared to low-level approaches such as RTL level designs or physical prototyping on FPGAs [9] [11], which are precise, but are often slow and tedious.

In this paper, we propose a high-level approach for a rapid assessment of MPSoC designs. The concurrency of system behaviors is represented by *abstract clocks* inspired by the synchronous reactive approach [4]. The way these behaviors are defined ensures correct-by-construction system scheduling, w.r.t. specified data dependencies or event precedences, on multiprocessor execution platforms. The abstract clock modeling is flexible enough to address adaptive system behaviors, including changes of processor frequencies and task migration. The proposed approach is supported by a tool, called CLASSY, which helps a designer to rapidly assess MPSoC design choices regarding temporal performances and energy consumptions. Thanks to its high abstraction level, it favors a rapid system prototyping. A preliminary work of parts of the results has been presented in [5]. A comparison of our approach with cycle-accurate simulations is shown for the motion JPEG decoder.

The rest of this paper is organized as follows: Section 2 discusses related works; Section 3 introduces our abstract clock design framework by focusing on the modeling of system structure; Section 4 deals with system execution via the synthesis of correct-by-construction scheduling; Section 5 extends the scheduling for adaptive MPSoC behaviors; Section 6 gives an overview of the dedicated CLASSY design and analysis tool; Section 7 reports a case study; finally, Section 8 gives conclusions and perspectives.

2 Related Work

Scheduling is central in system design when different operations have to be achieved on multiple execution units. In [16], authors distinguished four basic scheduling problems: *i)* the unconstrained scheduling (UCS) consisting in finding a feasible (or optimal) schedule w.r.t. a set of operations O, a set of unit types U, a mapping function $m : O \rightarrow U$ and a partial order on O denoting precedence constraints; *ii)* the time-constrained scheduling (TCS) and *iii)* resource-constrained scheduling (RCS) problems, which respectively add time and resource constraints on UCS problem; and *iv)* the time- and resource-constrained scheduling (TRCS) problem, combining both TCS and RCS. To solve these problems, four scheduling techniques are discussed, namely as soon as possible/as late as possible ASAP/ALAP scheduling, list scheduling, force-directed scheduling and integer linear programming (ILP). In this paper, we address an RCS problem with list scheduling technique, which is a common choice for solving RCS problems [16]. We do not use ILP, which can also solve RCS problems, because of its inevitable cost to guarantee optimism [16] and its unsuitability to deal with adaptive system behaviors.

Various models of computation have been proposed to model the application behaviors and to facilitate their scheduling and analysis on execution platforms for embedded systems. Among them, synchronous dataflows (SDFs) [8] capture the concurrent execution of applications and facilitate their analysis. Their authors developed a whole theory to statically schedule SDF graphs on homogeneous architectures. They proposed techniques for constructing periodic admissible sequential and parallel schedules, respectively referred to as PASS and PAPS. A period in PASS is constructed by computing the balance equations on data rates, while PAPS is achieved by constructing acyclic precedence graphs based on a number of periods of PASS. The proposed theory assumes homogeneity and uniform execution time on each processor, and not necessary synchronous processors. However, when scheduling an application on processors with different

frequencies, it does not take into account the possible delays between processor clock cycles.

Another relevant scheduling algorithm is the self-time scheduling [1] in which a task is executed as soon as it is enabled, i.e., input data are ready. Therefore, its implementation requires specific execution platforms such as synchronous architectures. In [6], the authors propose an operational semantics for SDF graphs to analyze the throughput by describing self-timed executions in terms of labeled transition systems. In [17], a self-timed scheduling on MPSoCs is studied. None of these studies investigates the impact of potential delay between processor cycles on scheduling.

In [15], the authors study the scheduling of real-time tasks on a heterogeneous platform with dynamic voltage and frequency scaling features. They propose a heuristic scheduling algorithm to explore the mapping choices from tasks to processor types and then to frequencies with the goal of energy minimization. The algorithm considers independent tasks as the input, and thus does not require to investigate the precedence relations or the potential delay between processor cycles.

In the Sesame framework [12], Kahn process networks (KPNs) are used to support system-level simulation and exploration of heterogeneous SoCs and reconfigurable architectures. They do not investigate the design of scheduling algorithms but rather consider them as a plug-in module which can be implemented as needed. As a result, simple scheduling schemes, like first come first served algorithms, are considered in their experiments. In our framework, we study admissible scheduling requirements, and propose a correct by construction scheduling algorithm.

The synchronous reactive MoC [4] is another popular support for the modeling and analysis of embedded systems. It considers a discrete time representation and assumes instantaneous computations and communications, referred to as "synchrony hypothesis". The synchronous composition allows to describe concurrency. In our approach, we consider a synchronous multi-clock modeling to model MPSoC systems. We explicitly represent processor clock cycles with abstract clocks, which enables us to address the potential delays between them during scheduling.

3 Clock-based Modeling of System Structure

We introduce a few basic definitions that are considered in our clock analysis system. We define models for application behavior, execution hardware platform and the mapping of both.

3.1 Application Behavior

We consider periodic embedded applications defined as a directed graph of tasks. These tasks exchange data according to the connections specified in an application graph. Each task has its own local activation clock according to which an associated sequence of events is observed. We construct our models by using the tagged signal system [7]. In the next, the following sets are assumed: a discrete set $\mathbb{T}$ of logical instants, having a smallest element τ_{min} and associated with a partial order $\leq$; and a value domain $\mathbb{V}$.

Definition 1 (event) *An event e is a pair (τ, v), where $\tau \in \mathbb{T}$ is a logical instant, and $v \in \mathbb{V}$ is a value.*

The set $\mathcal{E}$ of all possible events is associated with *a partial order relation* $\prec$ such that:

$$\forall e_1 = (\tau_1, b_1), e_2 = (\tau_2, b_2), \tau_1 \leq \tau_2, \tau_1 \neq \tau_2 \Rightarrow e_1 \prec e_2.$$

An event can have various meanings, e.g. communication and computation operations. For a task, at most one event occurs at a logical instant. Such an event denotes the task is active at

this instant. All events associated with the same task are totally ordered over the time. Given two events from different tasks, observed at the same logical instant, their respective precedence constraints w.r.t. all other events must be satisfied by each other. For instance, if events e_1 and e_2 occur at the same logical instant, then e_1 must satisfy all precedence constraints between e_2 and any other events, and vice versa.

Definition 2 (task and application behavior) *Given a task t, the behavior of t, denoted by b_t, is a totally ordered set of events. The behavior b_T of an application composed of a set T of tasks is a tuple $(E, C, \prec)$ where E is the set of events observed in all task behaviors, i.e. $E = \bigcup b_t, t \in T$, C is a precedence set composed of pairs of events (e_i, e_j) such that $e_i \prec e_j$ and $\prec$ is a precedence relation over E.*

Figure 1 illustrates an application behavior b_T with $T = \{t_0, t_1, t_2\}$, where $b_{t_0} = \{e_0^0, e_0^1\}, b_{t_1} = \{e_1^0, e_1^1\}$, $b_{t_2} = \{e_2^0, e_2^1\}$. Each event occurrence represents a task activation. The *arrows* in the figure are used to represent the precedence relations between events. For example, the arrow from event e_0^0 to event e_2^0 represents $e_0^0 \prec e_2^0$. The precedence set of this application behavior is $C = \{(e_0^0, e_2^0), (e_0^1, e_2^1), (e_0^1, e_1^1)\}$. The absence of arrow connection between two events means no precedence constraint between them, e.g., event e_0^1 and event e_1^0.

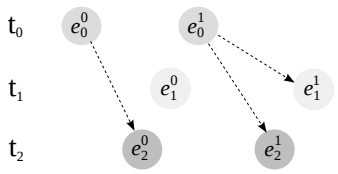


Figure 1: An application behavior b_T

We denote by $\mathcal{B}$ the set of all possible application behaviors. Many embedded applications have periodic behaviors. It is the case of streaming applications and time triggered applications. In our model, they are represented by a repetition of some application behavior patterns over the time, as defined as follows.

Definition 3 (periodic application behavior) *Given a periodic application composed of a set of tasks T, its behavior b_T is defined as a pair (π, ω), where $\pi \in \mathcal{B}$ is a behavior over T, repeated ω times over the time with $\omega \in \mathbb{N}^*$.*

If we consider the application behavior b_T in Figure 1 as the period pattern of a periodic application behavior $(b_T, 20)$ repeated 20 times, we obtain a horizontal repetition of the shown pattern.

3.2 Execution Platform Behavior

We consider an execution platform consisting of a set of processors P operating synchronously according to a reference clock and communicating via a shared memory. The platform can be heterogeneous, meaning that different kinds of processing elements can be supported, e.g. processors, hardware accelerators, etc. However, the characteristics of all these processing elements are assumed to be known at design time, and particularly during mapping of applications on a hardware platform. They include the usual information provided in the data sheets of processing elements, e.g. range of possible values for frequencies w.r.t. voltage levels.

Let P denote the set of processing elements in a platform. In our approach, we model platform behaviors through their clock activations according to given frequency values f_i of used processing elements $p_i \in P, 1 \leq i \leq |P|$. We define the reference clock $\mathcal{K}$ of the platform with the frequency value calculated as $LCM(f_1, ..., f_{|P|})$, where LCM denotes the Least Common Multiple. More concretely, the clock activations instants of the processing elements are modeled within a trace by considering the inverse of frequency values $1/f_i$, i.e. their period values. They are also referred to as processing element *clock cycles* in our approach.

We notice that a cycle $1/f_i$ of a processing element p_i is equal to a (integer) number of cycles $1/LCM(f_1, ..., f_{|P|})$ of the reference clock. We use $n_r(p_i)$ to denote this number. Figure 2 illustrates the behavior of a platform composed of three processors p_0, p_1 and p_2 with frequencies $f_0 = 100MHz, f_1 = 50MHz$ and $f_2 = 40MHz$.

	0	1	2	3	4	5	6	7	8	...
$\mathcal{K}$	•	•	•	•	•	•	•	•	•	...
p_0	•		•		•		•		•	...
p_1	•			•			•			...
p_2	•				•				•	...

Figure 2: Clock trace of processors

3.3 Mapping of Applications on Execution Platforms

As in usual software/hardware co-design, we define the mapping of applications on execution platforms. Such a decision usually precedes the scheduling of application tasks on processing elements.

Definition 4 (software-hardware mapping) *Given an application composed of a task set T and an execution platform composed of a processing element set P, a mapping of the application onto the platform is defined as a total function $M : T \rightarrow P$.*

In the above definition, we notice that the inverse function $M^{-1} : P \rightarrow 2^{|T|}$ of mapping function M is not necessarily a total function since the processing elements of a platform may not be all used for execution. After an application/execution platform mapping, we associate each event e occurring in an application behavior b_T with two parameters $\alpha(e/p_i)$ and $\beta(e/p_i)$, respectively representing the number of processor cycles corresponding to its computation and communication costs w.r.t. considered processor element p_i. These values are specified in terms of number of clock cycles and can be obtained statically by profiling their executions on the target processing elements.

4 Clock-based Modeling of System Execution

A notion of admissible schedule is proposed based on these models. An algorithm is provided for automatic generation of admissible schedules given a system specification. Afterward, we show how performance analysis is achieved on a scheduled system.

4.1 Scheduling of Applications on Execution Platforms

The scheduling decides when to execute the tasks specified in an application graph on selected processing elements of a platform from a mapping choice. Here, we assume that the events

belonging to the same task behavior are always executed on the same processor, and the execution of a single event is non-preemptive. The scheduling algorithm defined in the following aims to define at which logical instants w.r.t. the reference clock, task events are executed on processing elements.

For convenience, we represent the schedules of tasks on processors by means of a *ternary abstract clock* encoding. Such an abstract clock is a ternary-valued string over $\{-1, 0, 1\}$. The values 1 and 0 respectively represent the *active* and *idle* instants of a processing element executing some tasks w.r.t. the reference clock. The meaning of the value -1 is contextual: a sequence of -1 means active at these instants if it is preceded by 1, otherwise it denotes idle.

Figure 3 shows three ternary clocks, representing the scheduling of the tasks given previously in Figure 1 on the processors considered in Figure 2. For instance, from the ternary clock denoted by $clk(t_0/p_0)$ (i.e., scheduling of task t_0 on processing element p_0), the execution of event e_0^0 starts from the very first instant of the reference clock, and takes one clock cycle of p_0. The execution of event e_0^1 starts at the fourth instant of the reference clock and takes one cycle. In between their executions, the clock has two idle instants. In these ternary clocks, the value 1 indicates the logical instant at which the execution of the actions related to an event starts on the associated processor. The sequence of -1's following this value represents the duration of the whole event execution. The value 0 indicates the instant at which an event has to wait for execution. Typically, a wait of event may happen upon *i*) synchronization w.r.t. precedence constraints, i.e., its preceding events have not finished, and *ii*) resource unavailability, i.e., its mapped processor is running another event. Only a flavor of ternary clocks is given here, more aspects including the red "1" will be explained later.

	0	1	2	3	4	5	6	7	8	9
p_0	•		•		•		•		•	
$clk(t_0/p_0)$	1	-1	0	-1	1	-1				
$clk(t_1/p_0)$	0	1	-1	0	-1	-1	1	-1		
p_1	•			•			•			•
$clk(t_2/p_1)$	1	-1	-1	0	-1	-1	1	-1	-1	

Figure 3: An example of task schedules in terms of ternary clocks

A nice feature of ternary clocks is that they can be represented in a compact way. For instance, in Figure 3, the ternary clock $clk(t_2/p_1)$ is written as $1(-1)^2 0(-1)^2 1(-1)^2$, where the exponent denotes the number of repetitions. Such a notation is quite adequate when manipulating periodic ternary clocks that capture the execution of periodic embedded applications on MPSoCs.

4.2 Admissible Scheduling of Applications on Platforms

We target *admissible* schedules as in [8]. However, our notion of *admissibility* for schedules is slightly different as follows: *i*) the precedence constraints between events defined in application behaviors are preserved, *ii*) the executions of two or more events at the same time are not allowed on the same processor, and *iii*) when the activation instants of task events and the clock instants or ticks of their executing processors do not coincide, the non null delay between these instants is fully taken into account, i.e., processors are not yet ready to execute the events. For such an event, its effective execution is postponed to the next processor clock tick from its current position in time. Authors in [8] consider the point *i*) as the data dependency between task firings, and implicitly ensure point *ii*). Regarding point *iii*), they assume an architecture in which processors are always available to execute ready tasks, i.e., the possible non null delays between activation instants of task events and clock ticks are not taken into account. To the best of our knowledge,

no prior work addressed this aspect. An example of execution scenario illustrating the issue raised by point *iii*) is shown in the scheduling of t_1 on p_0, denoted by $clk(t_1/p_0)$ in Figure 3. The task t_1 starts its execution at the second instant of the reference clock, denoted by red colored "1", which lies between two clock ticks of processor p_0. This is not a valid execution.

The scheduling of an application behavior $(E, C, \prec)$ consists of the scheduling of its elementary events E w.r.t. C. We first define the scheduling of an event. Based on it, we introduce the three requirements regarding admissibility.

Definition 5 (schedule of an event) *A schedule of an event e on a processor p with parameters $\alpha(e/p)$, $\beta(e/p)$ and $n_r(p)$ is a ternary clock: $clk(e/p)_{pos} = (1(-1)^{(\alpha(e/p)+\beta(e/p))*n_r(p)-1})_{pos}$ where the subscript pos denotes a reference position on the reference clock, indicating the start instant of e on p.*

The schedule of an event encodes the beginning and the duration of its execution, respectively denoted by *pos* and the length of its corresponding scheduling clock. In Figure 3, the schedule of event e_0^1 of task t_0 on p_0, where $\alpha + \beta = 1$ and $n_r(p_i) = 1$, is $(1(-1))_4$. This means from instant number 4, w.r.t. reference clock, processor p_0 executes e_0^1 during the length of the clock $(1(-1))$.

The previous requirement *iii*) mainly concerns the *pos* information determined in the schedule of events and the considered processor clock. To deal with it, we define the *cycle integrity* of executions on a processor, which states that an execution on a processor cannot start within two successive processor clock ticks and its duration should be an integer number of processor clock cycles.

Definition 6 (cycle integrity) *A schedule on a processor is said to satisfy the cycle integrity of the processor if and only if the starting instants of all executions of the schedule always coincide with processor clock ticks, and the executions always occupy an integer number of clock cycles.*

The previous requirement *i*) concerns the precedence constraints defined on observed events. Admissible schedules of events must preserve these constraints.

Definition 7 (precedence constraint preservation) *Given two events with a precedence constraint $e_i \prec e_j, (i \neq j)$, their respective schedules $clk(e_i/p_t)_{pos_i}$ and $clk(e_j/p_k)_{pos_j}$ preserve the precedence relation if and only if: $pos_i + |clk(e_i/p_t)| \leq pos_j$ where $|clk|$ means number of clock instants w.r.t reference clock.*

In Figure 3, the precedence relation $e_0^0 \prec e_2^0$ is not preserved, as the schedule clock of e_0^0 ends at the third instant while the schedule of e_2^0 starts at the first instant. In the same example, the scheduling clocks preserve the precedence relation $e_0^1 \prec e_2^1$.

The aforementioned requirement *ii*) concerns the arbitration of processor access by ensuring that a processor cannot be used to run two different events at the same time.

Definition 8 (conflicting schedules) *Given two event schedules $clk(e_i/p), clk(e_j/p), i \neq j$ on the same processor p, they are conflicting denoted by $clk(e_i/p) \nparallel clk(e_j/p)$ if and only if there exists an instant at which both schedules are active.*

In Figure 3, the schedule of event e_0^0 is in conflict with that of e_1^0 on p_0, where they all expect to execute at the second instant.

Given the above definitions associated with our three requirements, we compute an admissible schedule of an application as a set of ternary clocks representing the schedules of all its events satisfying Definitions 6, 7, without any conflicting schedules. However, having a set of event schedules as the result is not pleasant, neither for verification of admissibility nor performance

analysis. Instead, we consider a ternary clock representation of the schedule of each task behavior, which follows the time progress of its mapped processor from the beginning, and describes its corresponding execution state on this processor, as defined below.

Definition 9 (schedule of a task behavior) *A schedule of a task behavior b_t on its mapped processor p is a ternary clock $clk(b_t/p)$, which represents its execution status on p, from the very first instant of the reference clock to the end of its execution. Accordingly, we say it is admissible if all schedules of its events are admissible.*

Typically, the schedule of a task behavior is constructed by synthesizing the schedules of all its events, i.e. by assembling their scheduling clocks properly according to their position *pos*, together with waiting 0's in between if needed. Figure 4 gives three possible admissible schedules w.r.t. the example of Figure 3.

	0	1	2	3	4	5	6	7	8	9
p_0	•		•		•		•		•	
$clk(t_0/p_0)$	1	-1	0	-1	1	-1				
$clk(t_1/p_0)$	0	-1	1	-1	0	-1	-1	-1	1	-1
p_1	•			•			•			•
$clk(t_2/p_1)$	0	-1	-1	1	-1	-1	1	-1	-1	

Figure 4: Admissible task schedules in terms of ternary clocks

Now, we define admissible schedules of an application behavior on a MPSoC platform as follows.

Definition 10 (admissible schedule of app. behavior) *An admissible schedule of an application behavior b_T on execution platform P w.r.t. mapping $M : T \to P$ is a set of admissible schedules of all tasks, i.e. $\{clk(t/M(t)), t \in T\}$.*

We represent the schedule of an application behavior b_T on a MPSoC as a set of ternary clocks corresponding to the schedules of all its task behaviors. The admissibility depends on whether the schedules of all events embedded in the schedules of task behaviors satisfy our basic three requirements.

We are interested in the performance analysis of executions, which concerns the running states of processors. Indeed, the scheduling clocks of task behaviors represent their execution states on processors. The running state of a processor with a single task running on it is represented exactly by the scheduling clock of the task. However, when two or more tasks are mapped on the same processor, we have to *compose the schedules* of all tasks running on it to provide a global behavior. For this purpose, we define a binary composition operator $\sqcup$, synthesizing the schedules of events from the schedules of different task behaviors on the same processor.

Definition 11 (composition operator $\sqcup$) *Given two admissible task schedules $clk(t_i/p)$, $clk(t_j/p)$, $i \neq j$ on the same processor p, their composition $clk(t_i/p) \sqcup clk(t_j/p)$ is a new ternary clock tracking the running state of p over the reference clock, with the length equal to the maximal length of the operands, i.e.:*

$$|clk(t_i/p) \sqcup clk(t_j/p)| = max\{|clk(t_i/p)|, |clk(c_j/p)|\},$$

which integrates all schedules of events from $clk(t_i/p)$ and $clk(t_j/p)$, and complements idle instants by using 0 followed by -1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
p_0	•		•		•		•		•		•		•		•		•		•	
$clk(t_0/p_0)$	1	-1	0	-1	1	-1	**0**	**-1**	**-1**	**-1**	1	-1	0	-1	1	-1				
$clk(t_1/p_0)$	0	-1	1	-1	0	-1	-1	-1	1	-1	0	-1	1	-1	0	-1	-1	-1	1	-1
p_1	•			•			•			•			•			•			•	
$clk(t_2/p_1)$	0	-1	-1	1	-1	-1	1	-1	-1	**0**	0	-1	-1	**1**	-1	-1	**1**	-1	-1	

Figure 5: An example of constructing periodic scheduling

Let us consider the two scheduling clocks $clk(t_0/p_0)$ and $clk(t_1/p_0)$ in Figure 4, we obtain:

$$clk(t_0/p_0) \sqcup clk(t_1/p_0) = 1(-1)1(-1)1(-1)0(-1)1(-1).$$

We refer to this composed clock as *schedule of processor* p_0.

Definition 12 (admissible schedule of a processor) *An admissible schedule of a processor p w.r.t. the schedule of an application behavior $\{clk(t/M(t)), t \in T\}$ is a ternary clock as follows: $clk(p) = \sqcup clk(t_i/p), t_i \in M^{-1}(p)$.*

We have described our requirements for admissible schedules, and given a number of definitions on the basis of our reasoning on ternary clocks. One should note that "admissibility" is more a global property. When we say the schedule of a task is admissible, it only means that all the requirements related to the schedules of its associated events are satisfied. To facilitate the performance analysis, we define the scheduling clocks of processors, which represents the execution behavior of all tasks on this processor.

4.2.1 Principle of our Scheduling Algorithm

As input parameters for our algorithm, we consider:

- a periodic application behavior $b_T = (\pi, \omega)$, defined on a set T of tasks and a set E of events;
- a platform composed of a finite set of processors $P = \{p_{i,i\in 1..n}\}$ that operate at frequency values defined in the set $\{f_{j,j\in 1..m}\}$;
- a mapping function $M : T \to P$ with parameters α, β defined for each event $e \in E$.

In particular, our algorithm is flexible enough to support non-periodic application behaviors, for instance, an application behavior having an initialization part and a periodic part, or application behavior without periods. For the former case, one can divide the application into two parts: the phase which is a periodic behavior of one period and the rest periodic behavior, and compose these two scheduling results. As for the latter case, it can be simply seen as an application behavior of one period.

Our goal is to define an algorithm that generates admissible schedules in the form of the scheduling clocks of all task behaviors according to the above input parameters. To reach this goal, we need to compute the schedules for all events in E, and embed them in the corresponding schedules of task behaviors. A *scheduling order* is defined such that the scheduling of an event is enabled only when all its precedent events according to $\prec$ have been scheduled. Moreover, it also indicates a total order between events when they are candidates at the same time for execution on the same processor. As we consider periodic applications, our algorithm defines a scheduling order for the events within one period and operates on it in a periodic fashion. By this way, we construct a periodic scheduling order.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	•		•		•		•		•		•		•		•		•		•
$clk(t_0/p_0)$	1	-1	0	-1	1	-1	0	-1	1	-1	0	-1	1	-1					
$clk(t_1/p_0)$	0	-1	1	-1	0	-1	1	-1	0	-1	1	-1	0	-1	1	-1			
p_1	•			•			•			•			•			•			•
$clk(t_2/p_1)$	0	-1	-1	1	-1	-1	1	-1	-1	0	-1	-1	1	-1	-1	1	-1	-1	

Figure 6: An example to construct our admissible schedule

Definition 13 (scheduling order) *A scheduling order ϕ of a periodic application behavior $b_T = (\pi, \omega)$ is a totally ordered list of task events within π respecting all precedence constraints in b_T.*

Let us consider the application behavior b_T shown in Figure 1 as the period of a periodic application behavior, a possible scheduling order is: $\phi =< e_0^0, e_1^0, e_2^0, e_0^1, e_1^1, e_2^1 >$. Note that, the scheduling order can be also defined for a finite number k of periods by considering an application behavior (π, ω) as (π^k, ω').

Our algorithm does not make the definition of the scheduling order mandatory for the users, and would construct one arbitrarily if undefined. Moreover, as the scheduling results may vary according to different scheduling orders, one can play with it so as to refine the results. The algorithm does not ensure optimal scheduling due to the arbitrary or user defined scheduling order, but aims to generate an admissible one given the inputs.

Compared to the construction of periodic admissible sequential and parallel schedules in [8], we have one single algorithm to deal with both cases. In [8], the parallel schedule is constructed periodically based on the sequential schedule. The period is defined first by scheduling a number of periods of the sequential schedule on a set of processors. Then, it is repeated. In our admissible schedule, this schedule construction is not directly feasible because we consider architectures supporting different processor frequencies and cycle integrity as requirement.

An example is given in Figure 5, which considers the schedule of Figure 4 as the schedule of one period, and repeats it. However, the schedule of the second period violates of the cycle integrity at instants 13 and 16 denoted by red "1"s of $clk(t_2/p_1)$. Thus, to construct a periodic schedule, we must take into account the execution platform and in particular the processor clocks. If we postpone the start of second period to the instant number 12, at which the processors have the same state as the very first instant, we can construct a periodic schedule. However, such a schedule would have many unnecessary delays. In fact, our principle is to schedule the observed events as soon as possible according to the scheduling order. We first compute the LCM of all the frequencies of the platform components so as to evaluate $n_r(p_i), p_i \in P$ of the platform. Provided the scheduling order ϕ, our algorithm proceeds periodically, and within each period, it computes the schedules of events following the given order. During the scheduling of an event $e \in b_t$ on its mapped processor p, the task and processor scheduling clocks are constructed incrementally by computing and concatenating the event schedule $(clk(e/p)_{pos})$ to them. The computation of *pos* ensures admissible schedules. A task/processor schedule is complemented by idle instants between its last instant and the start instant *pos* of the event schedule when *pos* is not its next scheduling instant.

Let us consider the example in Figure 5, where the application behavior has two periods and $\phi =< e_0^0, e_1^0, e_2^0, e_0^1, e_1^1, e_2^1 >$. The resulting schedule of our algorithm is given in Figure 6. According to ϕ, event e_0^0 is firstly scheduled on p_0 from the first instant number 0 and occupies one processor clock cycle. The event e_1^0, which has no precedent events, is then scheduled on the same processor. It can only start its execution after e_0^0, i.e., at instant number 2. The event e_2^0

is scheduled on processor p_1. However, since it has precedent event e_0^0, its schedule cannot start until instant number 2. Moreover, as this instant does not coincide with processor clock of p_1, the earliest instant to start the schedule of e_2^0 is at instant number 3. This process continues for the rest of the period and all following periods. The blue colored 1 values indicate the start instants of the three task behaviors at the second period.

4.2.2 Definition of Scheduling Algorithm

Algorithm 1 Schedule of an event on a processor w.r.t. a precedence relations preservation position

1: **Inputs:** an event e, its source task t, its mapped processor p, current scheduling clocks of t and p: $sclk(t), sclk(p)$, the numbers of communication and computation cycles required for e to execute on p: $\alpha(e/p), \beta(e/p)$, the number of reference clock ticks corresponding to one cycle of p: $n_r(p)$, a precedence relations preservation position rp representing the position where all its precedent events have finished.
Local: sp: the very first instant of p after rp.
Outputs: incremented clocks $sclk(t), sclk(p)$.
2: **if** $rp > |sclk(p)|$ **then**
3: $sp \leftarrow \min\{index | index/n_r(p) = 0, index > rp\}$;
4: $sclk(p) = sclk(p) \oplus 0(-1)^{[sp-|sclk(p)|-2]}$
5: **end if**
6: **if** $|sclk(t)| < |sclk(p)|$ **then**
7: $sclk(t) = sclk(t) \oplus 0(-1)^{[|sclk(p))|-|sclk(t)|-1]}$
8: **end if**
9: $sclk(t) = sclk(t) \oplus 1(-1)^{[(\alpha(e/p)\oplus\beta(e/p))*n_r(p)-1]}$;
10: $sclk(p) = sclk(p) \oplus 1(-1)^{[(\alpha(e/p)\oplus\beta(e/p))*n_r(p)-1]}$

Algorithm 2 defines our scheduling algorithm. It invokes Algorithm 1 for the schedule of an event on a processor. It constructs the admissible scheduling clocks of tasks and processors in an incremental way. In the algorithms, we use a binary operator $\oplus$ to support the concatenation of ternary clocks, defined as follows.

Definition 14 (concatenation operator $\oplus$) *Given two ternary clocks clk_1 and clk_2, their concatenation $clk_1 \oplus clk_2$, appends clk_2 to the end of clk_1.*

For instance, $1(-1)^5 \oplus 1(-1)0(-1) = 1(-1)^5 1(-1)0(-1)$.

From Line 2 to Line 8 in Algorithm 2, the scheduling clocks of tasks and processors are initialized to empty. The number of reference clock cycles per processor cycle denoted by $n_r(p)$, is computed for each processor p. The periodic scheduling of this algorithm can be seen from Line 9. Within each period, every event in the list ϕ is scheduled *sequentially* on its processor (see Line 10).

From Line 11 to Line 16, together with Algorithm 1, deal with the scheduling of an event e_j^k on its mapped processor, which updates the corresponding task and processor scheduling clocks $sclk(t_j)$ and $sclk(M(t_j))$. In order to generate admissible scheduling clocks, the algorithm takes care of our three admissibility requirements as follows: two local variables, rp in Algorithm 2 and sp in Algorithm 1, are used to deal with the precedence relation and cycle integrity, whereas the non-simultaneous execution constraint is ensured by the incremental construction of scheduling clocks. rp computes the last instant w.r.t. the reference clock where all its precedent events finish executions (see Line 14 of Algorithm 2). sp takes into account rp and computes the very next instant, respecting cycle integrity of the corresponding processor, thus allowing the processor to

Algorithm 2 Scheduler

1: **Inputs:** a periodic application behavior $b_T = (\pi, \omega)$, a scheduling order ϕ, frequency f of each processor $p \in P$, and a mapping $M : T \rightarrow P$.
Local: rp: the last instant at which all corresponding precedent events finish executions, and $n_r(p), \forall p \in P$.
Output: the scheduling clocks $sclk(t), sclk(p), \forall t \in T, \forall p \in P$.
2: **for all** $t \in T$ **do**
3: $sclk(t) = \oslash$
4: **end for**
5: **for all** $p \in P$ **do**
6: $sclk(p) = \oslash$;
7: $n_r(p) = LCM(f_1, ..., f_{|P|})/f_p$
8: **end for**
9: **for** $i = 1 \rightarrow \omega$ **do**
10: **for all** $e_j^k \in \phi$ **do**
11: **if** $pre(e_j^k) \bigcap \phi = \oslash$ **then**
12: *Algo.1:* schedule e_j^k on $M(t_j)$ w.r.t. 0
13: **else**
14: $rp \leftarrow \max\{|sclk(t_p)|, \forall e_p^q \in pre(e_j^k) \bigcap \phi\}$;
15: *Algo.1:* schedule e_j^k on $M(t_j)$ w.r.t. rp
16: **end if**
17: **end for**
18: **end for**

run an event. When the processor is idle, then an event has to wait (denoted by 0 followed by a sequence of -1's) until the instant sp (Line 4 of Algorithm 1).

Moreover, the scheduling clock of the task must coincide with the schedule of its mapped processor. In other words, the schedule of task event e should be the same on both scheduling clocks. This is ensured by the statements from Line 6 to Line 8 of Algorithm 1. The algorithm looks at the instant where the processor is about to run the event, and appends 0 followed by a sequence of -1s to the task scheduling clock if it needs to wait. Finally, the schedule of the event is appended to the corresponding task and processor scheduling clocks in Line 9 and Line 10 of Algorithm 1 respectively.

Property 1 (Correctness) *Algorithm 2 always generates an admissible schedule in the sense of Definition 10.*

Proof 1 *To prove this property, we need to prove the following: the precedence relation preservation (Definition 7), cycle integrity (Definition 6) for all scheduling clocks, and non-simultaneous execution schedules exist for the same processor (Definition 8).*

Precedence relation preservation. We distinguish the precedence relations defined a) on events within the same task behavior, and b) on events from different task behaviors. For case a), $\forall e_j^p, e_j^q, e_j^p \prec e_j^q$, the precedence relation is preserved due to the fact that the algorithm schedules e_j^p before e_j^q and the scheduling clocks are computed incrementally, i.e. the schedule of e_j^q is appended to the corresponding scheduling clocks after the one of e_j^p. For case b), $\forall e_i^p, e_j^q, i \neq j, e_i^p \prec e_j^q$ is kept due to, on the one hand, the algorithm projects e_j^p before e_j^q because of the order of ϕ, and on the other hand, the schedule of e_j^q starts after the finish instant the schedule of e_j^p which is under the charge of the variable rp.

Cycle integrity. This requires that all generated scheduling clocks have 1s and 0s only coincide with corresponding processor tick instants. Since the clocks are only updated in Lines 4, 7, 9, 10 of Algorithm 1, we need to prove these updates do not introduce violation. Initially, all clocks

are empty, thus if the codes of Lines 4 and 7 are executed, the 0 *would be appended to the first instant which apparently coincides the clock tick. Moreover, after a number of* -1*s are appended, each clock has the next instant coinciding with the clock tick due to sp for Line 4 and* $sclk(p)$ *for Line 7 (whose next instant is a clock tick). Then for Lines 9 and 10, each* 1 *is appended to a clock instant coinciding with clock tick, and each clock finishes just before a tick instant as the schedule of an event spans a number of processor cycles. As a result, the property follows for all scheduling clocks.*

Non-simultaneous execution schedules. This property follows due to the following two facts of our algorithm. Firstly, before scheduling an event e*, the algorithm always matches its corresponding task and processor scheduling clocks, which ensures that the schedule* $clk(e)_{pos}$ *of* e *is the same on both clocks w.r.t. the reference clock. Secondly, the processor scheduling clock is computed incrementally, which ensures that at the same instant at most one event could be scheduled.*

Given an application behavior $b_T = (\pi, \omega)$ and a scheduling order ϕ, the algorithm computes the schedules of $\omega * |\phi|$ events, where $|\phi|$ denotes the number of events in ϕ. During the schedule of each event, the scheduling clocks of all its preceding events within ϕ are checked, where the number of these events is up to $|\phi|$. Thus, the complexity of the algorithm is $\omega * |\phi|^2$. As the size of a scheduling order is linear to the number of tasks denoted by $|T|$, it can also be represented as $\omega * |T|^2$.

4.3 Performance Analysis

With the generated scheduling clocks of tasks and processors, various performance parameters can be analyzed in our framework.

The scheduling clocks of processors characterize the execution states of processors over the time. Given the scheduling clock $sclk(p_i)$ of a processor $p_i \in P$, it is quite direct to compute its execution time $ET(p_i)$ and usage ratio $UR(p_i)$ as follows:

- $ET(p_i) = |sclk(p_i)| * (1/LCM(f_i, ...f_{|P|}))$,
- $UR(p_i) = nbc(sclk(p_i))/(nbc(sclk(p_i)) + nic(sclk(p_i)))$,

where functions $nbc(sclk(p_i))$ and $nic(sclk(p_i))$ count the number of busy processor cycles and of idle processor cycles of a scheduling clock respectively. The usage ratio of a processor captures its usage efficiency, and is computed by the number of busy cycles divided by overall number of running cycles.

The global execution time of an application behavior $(E, C, \prec)$, is the maximal execution time among all its mapped processors:

$$ET(E, C, \prec) = max\{ET(p_i), p_i \in P\}.$$

The other interesting performance parameter is the energy consumptions. Our framework is able to compute it, if provided with corresponding profiling results. For instance, given the energy consumption values of a busy and an idle cycle of processor $p_i \in P$, denoted by $bec(p_i)$ and $iec(p_i)$ respectively, as well as its resulting scheduling clock $sclk(p_i)$, we compute the energy consumption $EC(p_i)$ of processor p_i as follows, and then the overall energy consumption $EC(E, C, \prec)$ as well.

- $EC(p_i) = nbc(sclk(p_i)) * bec(p_i) + nic(sclk(p_i)) * iec(p_i)$;
- $EC(E, C, \prec) = \sum EC(p_i), p_i \in P$.

Furthermore, the scheduling clocks of tasks can be used to analyze the distance between the executions of two communicating events from different task behaviors within the same application period. Let us consider two communicating tasks A and B, where A produces a data block to feed B in each period. By computing the distance, and the number of produced events by A within this distance, we get indications about the required buffer size.

5 Capturing MPSoC Execution Adaptivity

In this section, we extend our clock-based scheduling and analysis framework to dynamic features of embedded systems, e.g. frequency changing and task migration. We consider periodic applications (π, ω) with an adaptivity at period level, meaning that when an adaptation request occurs, the execution of a system continues for the current period, and will change only in the next period.

To facilitate the generation of an admissible schedule in the sense of Definition 10, in reaction to adaptation requests, we consider an off-line modeling and generation algorithm. The adaptation requests are assumed to be provided before the scheduling generation. They consist of the timing information in terms of number of periods about the instants at which an adaptation is required, and the kind of adaptation, e.g., frequency variation or task migration. In the scheduling and performance analysis, we take into account the *time and energy consumption penalty* caused by context changing within adaptive behaviors.

Scheduling of Dynamic Executions. Thanks to the incremental scheduling style of Algorithm 2, its extension for execution adaptivity is easily constructed. Consider an adaptation request (rt, rb) with $rt \in \mathbb{N}^+$ representing the scheduling period within which the request occurs, and rb representing the expected behavior. The basic idea consists of first using the scheduling algorithm 2 to generate the scheduling clocks for the first rt periods, and then applying the adaptation behavior rb and the corresponding penalty denoted by $\lambda(rt, rb)$ in terms of number of processor cycles, up to a new request or to the end of the schedule. Algorithm 3 provides the pseudo-code for the scheduling of adaptive behaviors.

Algorithm 3 Scheduler Regarding Adaptation Requests

Inputs: a periodic application behavior (π, ω), a scheduling order ϕ, frequency f of each processor $p \in P$, mapping $M : T \to P$, and an ordered adaptation request list $\mathcal{L} =< (rt_1, rb_1), ..., (rt_m, rb_m) >$, $m \in \mathbb{N}^+$ as well as the adaptation penalty $\lambda(rt_i, rb_i)$.
Local: rp: the last instant where all its precedent events finish executions, and $n_r(p), \forall p \in P$.
Output: the scheduling clocks $sclk(t), sclk(p)$ for all tasks and processors.
Line 2 to Line 8 of Algo. 2.
for $i = 0 \to m$ **do**
  if i=0 **then**
    Line 9 to Line 18 of Algo. 2 with $1 \to rt_{(i+1)}$;
    apply $rb_{(i+1)}, \lambda(ri_{(i+1)}, rb_{(i+1)})$;
  else if i=m **then**
    Line 9 to Line 18 of Algo. 2 with $(rt_i + 1) \to \omega$;
  else
    Line 9 to Line 18 of Algo. 2 with $(rt_i + 1) \to rt_{(i+1)}$;
    apply $rb_{(i+1)}, \lambda(ri_{(i+1)}, rb_{(i+1)})$.
  end if
end for

Given an ordered adaptation request list $\mathcal{L} = \{(rt_1, rb_1), ...,$

$(rt_m, rb_m)\}$, the defined algorithm decomposes the scheduling of a periodic application behavior (π, ω) into $m+1$ pieces (From Line 2 to Line 11 of Algorithm 3). Each piece deals with the scheduling under one specific system state which includes the current processor frequencies and mapping M. The algorithm starts the scheduling w.r.t. the initial system state until the first adaptation event happens, and then adapts the system state (From Line 3 to Line 5 of Algorithm 3 where $i = 0$). Regarding the following adaptation requests, the algorithm firstly performs the scheduling according to the current system state until the adaptation request happens, and then applies the corresponding adaptation behavior (From Line 9 to Line 10 of Algorithm 3 where $i = 1, ..., m-1$). At last, after all adaptation events are taken into account, and all the events that happen earlier than the last adaptation event have been scheduled, the algorithm schedules the remaining events accordingly (From Line 6 to Line 7 of Algorithm 3 where $i = m$). In our algorithm, we have implemented two kinds of adaptation requests: *frequency changing* and *task migration*. In particular, when frequency changing is considered, the frequency to be used must be either compatible with the initial LCM, or provided at the beginning in order to compute the compatible LCM. Moreover, we can also consider the adaptation in the application level, where the application changes its behavior due to external environment. In this case, we just need to take the new system specification, and then continue the scheduling process. The way our algorithm takes into account the adaptation penalty $\lambda(rt, rb)$ is to concatenate a number of idle λ cycles to the corresponding processor scheduling clocks.

Property 2 (correctness of algorithm 3) *Algorithm 3 always generates admissible schedules.*

Proof 2 *We still need to prove the three constrains as in the proof of Property 1. As this algorithm does not change the scheduling behavior of events, the proof for precedence relation preservation and non-simultaneous execution still holds. The difference lies in the proof of cycle integrity, as we take into account adaptation penalty as shown in Lines 12, 17, 21 of Algorithm 3. However, since we always add a number of cycles at the end of the current scheduling clocks, such operation still keeps the cycle integrity accordingly. Thus, the property follows.*

In fact, Algorithm 3 complies with the policy according to which an adaptation is performed once the tasks executing on processors have finished the current period. We have also considered another policy by applying the adaptation only when all tasks have finished all the computations initiated in the current period. Regarding this policy, another scheduling algorithm has also been constructed on the basis of Algorithm 2. It decomposes the scheduling of the periodic application as in Algorithm 3. The main difference is that each time before continuing to generate the scheduling clocks w.r.t. an adaptation request, it synchronizes all the scheduling clocks according to the one that terminate the last. Due to space limitation, we do not give this algorithm in the paper.

With the generated scheduling clocks, the performance analysis is as in Section 4.3. An *online* scheduling algorithm can be constructed by simply checking whether to adapt at the end of the scheduling of one period (between Line 17 and Line 18 of Algorithm 2), and performing the adaptation behavior w.r.t. penalty cost correspondingly.

6 Implementation: the CLASSY tool

Our prototype tool, named CLASSY (CLock AnalySis SYstem), implements the modeling, scheduling and analysis approach described in previous sections. It has around one thousand Java code lines and consists of five modules (see Figure 7) as follows:

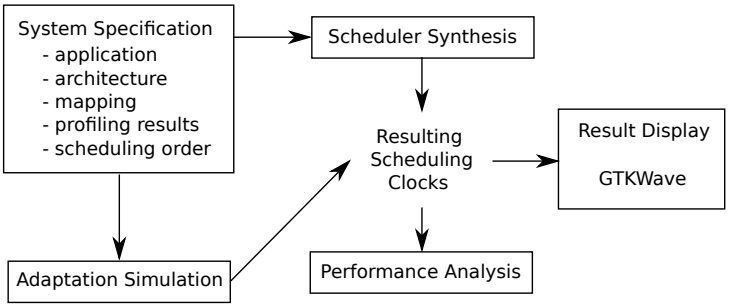


Figure 7: Overview of the CLASSY tool.

- *system specification* provides interface for the user to define application behavior (including task behaviors, precedence relation), execution platform behavior (including processors and frequencies), application-architecture mapping, as well as performance analysis parameters like number of cycles evaluated for the computation of events, energy consumptions regarding the idle and active cycles of processors.
- *scheduler synthesis* generates an admissible schedule w.r.t. the system specification, which is a set of scheduling clocks including the scheduling clocks of all tasks as well as the composed scheduling clocks of processors.
- *adaptation simulation* provides 1) an interface to describe the adaptation requests, i.e. the adaptation behaviors (either frequency changing or task migration) and corresponding timings, and 2) an off-line simulation process to generate the scheduling clocks of all tasks and processors.
- *performance analysis* computes, on the basis of generated scheduling clocks, the execution time, energy consumption as well as the distances between two events from different tasks.
- *result display* generates a *vcd* file to feed GTKWave so as to visualize the schedules of tasks on their mapped processors, as well as the running states of processors.

7 Case study: M-JPEG decoder

We consider a *motion JPEG (M-JPEG) decoding* algorithm as case study application. We implement it with the SoCLib environment [2], dedicated to SoC prototyping and cycle-accurate simulation. The obtained results are compared with those observed with CLASSY. The simulation results observed with our tool are correct-by-construction and obtained rapidly at a low cost.

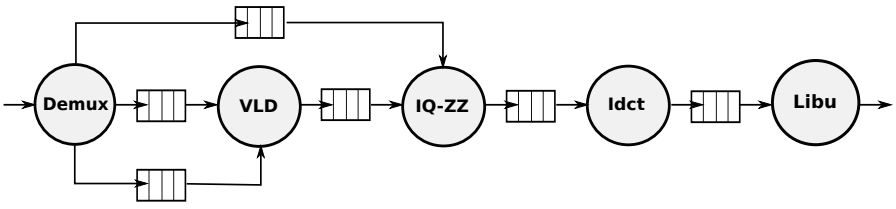


Figure 8: Application graph specifying the motion JPEG decoder

7.1 Informal presentation

The M-JPEG decoder applies some filters to pixel streams for image decoding. It is represented in Figure 8. The core algorithm manipulates 8×8-pixels blocks and is composed of five tasks as follows: Demux, a demultiplexer of M-JPEG algorithm input pixels streams; VLD, a variable length decoding consisting in combined Huffman and run-length decoding on compressed images; IQ-ZZ (also referred to as Iqzz in the sequel), an inverse quantization followed by inverse zig-zag reordering that rearrange pixels of a block in diagonal in order to enhance the image rendering; Idct: an inverse discrete cosine transformation; and Libu: a construction of image lines from a given number of 8×8-pixels blocks resulting from image decompression.

For the execution of the M-JPEG decoder, we consider a multiprocessor platform with a shared multi-bank memory, which can be configured to support up to five processors interconnected with a bus or a network-on-chip (NoC).

Concerning the application/platform mapping, a buffer is mapped on the memory bank associated with its consumer task. When two communicating tasks are mapped on different processors, a task consumes its data from its memory bank and writes produced data on the memory bank of the processor executing the consumer task. The studied mapping configurations are summarized in Table 1. We consider up to five processors $\{p_1, p_2, p_3, p_4, p_5\}$. For instance, in configuration number 1, all M-JPEG tasks are executed on processor p_1 while in configuration number 2, the successive tasks Demux, Vld, Iqzz are executed on p_1 and the successive tasks Idct and Libu are executed on p_2. In configuration number 3 also the same processors are considered, but the defined mapping does not select successive tasks to execute on the same processor. We refer to multiprocessor configurations like the number 2 as *successive task mappings* and multiprocessor configurations like the number 3 as *non successive task mappings*.

7.2 Abstract clock-based design

To achieve our experiments with CLASSY, we identified from the considered SoCLib implementation an application behavior for the M-JPEG according to the proposed clock-based framework. The obtained periodic behavior is illustrated in Figure 9. It is composed of two parts: 1) an initialization part, indicated by a blue curve, where some initial communications are achieved between the Demux task and the Vld and Iqzz tasks; and 2) a periodic part, indicated by a red curve, which is repeated 36 times and consists of pixel block-wise decoding of an image.

We schedule the above two parts of the application behavior separately, then we put them together. The scheduling orders for the initialization and the periodic parts are respectively as follows:

- $\phi_1 = \{e^1_{demux}, e^1_{iqzz}, e^2_{demux}, e^1_{vld}\}$,
- $\phi_2 = \{e^3_{demux}, e^2_{vld}, e^2_{iqzz}, e^1_{idct}, e^1_{libu}\}$.

Regarding the input profiling data for each task of the M-JPEG, Table 2 gives the values $\alpha(e) + \beta(e)$ corresponding to each event shown in Figure 9. The given values are average values obtained from a profiling of the application implementation in SoCLib.

7.3 Comparison of simulation results

A part of the simulation results obtained from our clock-based approach on the M-JPEG are reported in Figure 10, together with those observed with SoCLib. They represent the temporal performances associated with the mapping configurations summarized in Table 1. In Figures 10(a) and 10(b) all processors always operate at the same frequency, while it is not the case in

Configurations identifiers	M-JPEG tasks	Mapped processors
1	Demux, Vld, Iqzz, Idct, Libu	p_1
2	Demux, Vld, Iqzz	p_1
	Idct, Libu	p_2
3	Demux, Iqzz, Libu	p_1
	Vld, Idct	p_2
4	Demux, Vld	p_1
	Iqzz	p_2
	Idct, Libu	p_3
5	Demux, Iqzz	p_1
	Vld, Libu	p_2
	Idct	p_3
6	Demux	p_1
	Vld	p_2
	Iqzz	p_3
	Idct, Libu	p_4
7	Demux, Libu	p_1
	Vld	p_2
	Iqzz	p_3
	Idct	p_4
8	Demux	p_1
	Vld	p_2
	Iqzz	p_3
	Idct	p_4
	Libu	p_5

Table 1: Analyzed mapping configurations for M-JPEG.

Figures 10(c) and 10(d). Two system implementations are considered in SoCLib according to the communication infrastructure: bus *versus* NoC.

The experiments show that our clock-based approach yields results with similar tendency as those obtained with SoCLib. The precision of the results provided by CLASSY appears good when compared to the NoC-based results. However, it is not the case when considering the bus-based results. This observation is explained by the fact that NoCs offer higher communication performances than buses. The execution time obtained with NoCs is therefore shorter thanks to reduced communication time. In addition, possible bus access conflicts, which increase the communication overhead, lead to lower performances compared to NoC-based implementations. This issue is usually observed when the number of processors sharing the same bus gets higher. This may explain the increase of the execution time in Figure 10(b), from configuration number 5 (three processors) to configuration number 8 (five processors). Since in our clock-based model of the M-JPEG application, the approximation of input profiling data given in Table 2 does not cover such communication overheads, the obtained results are less precise w.r.t. bus-based implementations.

For the results obtained with processors operating at different frequencies in CLASSY, i.e. Figures 10(c) and 10(d), the initialization part of the M-JPEG application behavior in Figure 9, has been scheduled first on the processor with the highest frequency. In addition, we considered a scheduling order defined over two periods. For tasks consisting of a huge number of events, specifying the system behaviors can be very tedious.

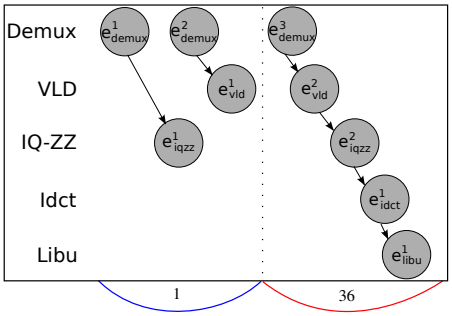


Figure 9: Application behavior for M-JPEG.

Tasks	Observed events	Number of repetitions	Number of processor cycles
Demux	e^1_{demux}	1	12651
	e^2_{demux}	1	21032
	e^3_{demux}	36	2464
Vld	e^1_{vld}	1	28042
	e^2_{vld}	36	3007
Iqzz	e^1_{iqzz}	1	1668
	e^2_{iqzz}	36	4946
Idct	e^1_{idct}	36	8978
Libu	e^1_{libu}	36	1496

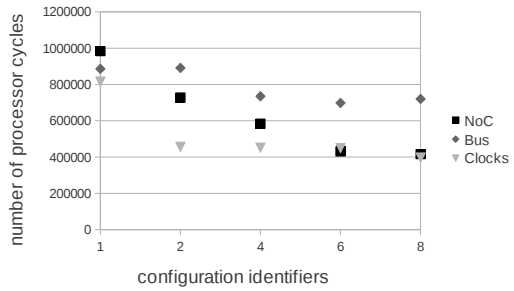
Table 2: Profiling data about M-JPEG tasks as inputs for CLASSY.

8 Conclusions and perspectives

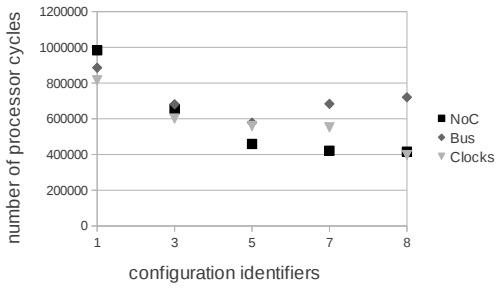
We presented an abstract clock-based framework to model and analyze embedded system executions on MPSoCs. We adopted a synchronous multi-clock model for system description. We proposed a ternary clock encoding for their schedules on MPSoCs. We defined admissible schedules for a correct-by-construction system execution w.r.t. three basic requirements: *precedence relation preservation*, *non-simultaneous executions of tasks on the same processor* and *cycle integrity* (the delay between task activation instants and executing processor clock cycles is fully taken into account).

The proposed scheduling algorithm has been extended to support adaptive system behaviors including dynamic frequency changing and task migration. Our clock-based framework has been implemented in a prototype tool, named CLASSY, to help for rapid system prototyping. A few experiments have been reported about simulations of the M-JPEG algorithm on a multiprocessor platform. They showed that CLASSY provides relevant information in comparison with cycle-accurate simulation tools such as SoCLib. The accuracy of CLASSY quite depends on its input profiling data about the execution of elementary tasks. However, finding a confident approximation of communication delays as part of these information is a crucial issue. Indeed, communication temporal behaviors are usually less deterministic than the pure computational ones.

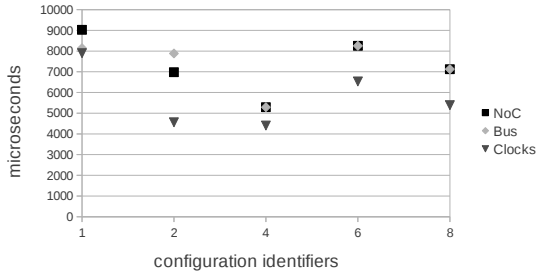
Our future work aims to improve CLASSY for more precision regarding communication aspects in systems. A connection with a specific environment such as Cafes [10], which automatically synthesizes mappings on NoC-based MPSoCs, is one solution to provide CLASSY with input mapping choices associated with computed communication costs. We also plan to inte-



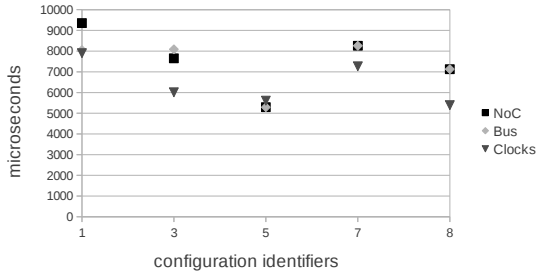
(a) Successive task mappings (proc. with same frequency)



(b) Non successive task mappings (proc. with same frequency)



(c) Successive task mappings (proc. without same frequency)



(d) Non successive task mappings (proc. without same frequency)

Figure 10: Execution times for M-JPEG decoder on an image: CLASSY *vs* SoCLib cycle-accurate simulations (comm. via bus and NoC).

grate our algorithm into some heuristics so as to allow optimal scheduling. An example is to minimize energy consumption while still meeting a deadline. We can play with exploring the mappings or scheduling orders choices to achieve this. Another interesting direction is to combine our framework with controllers so that it can take decisions about system adaptation in reaction to environment.

References

[1] *Embedded multiprocessors: Scheduling and synchronization.* CRC Press, 2000.

[2] The SoClib Project, 2011. `http://www.soclib.fr`.

[3] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35:70 –78, jan 2002.

[4] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64 – 83, jan 2003.

[5] Abdoulaye Gamatié. Design of Streaming Applications on MPSoCs using Abstract Clocks. In *Design, Automation and Test in Europe Conference (DATE'2012)*, Dresden, Germany, 2012.

[6] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *In Proc. of ACSD'06, IEEE*, pages 25–34, 2006.

[7] Edward A. Lee and Alberto Sangiovanni-vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998.

[8] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987.

[9] Hyung Gyu Lee, U.Y. Ogras, R. Marculescu, and N. Chang. Design space exploration and prototyping for on-chip multimedia applications. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 137 –142, 0-0 2006.

[10] César Marcon, Ney Calazans, Edson Moreno, Fernando Moraes, Fabiano Hessel, and Altamiro Susin. Cafes: A framework for intrachip application modeling and communication architecture design. *J. Parallel Distrib. Comput.*, 71:714–728, May 2011.

[11] Matthias May, Norbert Wehn, Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, Andreas Herkersdorf, Daniel Ziener, and Jürgen Teich. A rapid prototyping system for error-resilient multi-processor systems-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 375–380. European Design and Automation Association, 2010.

[12] Kamana Sigdel, Mark Thompson, Andy D. Pimentel, Carlo Galuzzi, and Koen Bertels. System-level runtime mapping exploration of reconfigurable architectures. In *Reconfigurable Architecture Workshop(RAW'09)*.

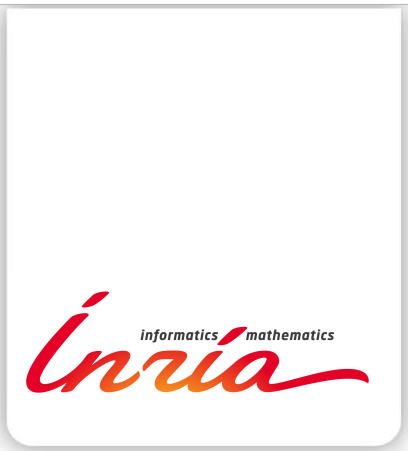
[13] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 899 –904, 0-0 2006.

[14] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *CODES+ISSS'07*, pages 9–14, New York, NY, USA, 2007. ACM.

[15] Bruno Virlet, Xing Zhou, Jean Pierre Giacalone, Bob Kuhn, Maria J. Garzaran, and David Padua. Scheduling of stream-based real-time applications for heterogeneous systems. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pages 1–10, New York, NY, USA, 2011. ACM.

[16] R.A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *Design Test of Computers, IEEE*, 12(2):60 –69, summer 1995.

[17] Jun Zhu, Ingo Sander, and Axel Jantsch. Energy efficient streaming applications with guaranteed throughput on mpsocs. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT'08, pages 119–128, 2008.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399