Proceedings of the

11^{th} Junior Researcher Workshop on Real-Time Computing

JRWRTC 2017

http://www.rtns17.org/jrwrtc2017/

Grenoble, France

October 3-6, 2017

Message from the Workshop Chairs

Welcome to the 11^{th} Junior Researcher Workshop on Real-Time Computing, held in conjunction with the 25^{th} International Conference on Real-Time and Network Systems (RTNS) in Grenoble, October 2017. The workshop provides an informal environment for junior researchers, where they can present their ongoing work in a relaxed forum and engage in enriching discussions with other members of the real-time systems community.

We would like to take this opportunity to express our gratitude to the members of the Program Committee listed below for thoroughly reviewing all the submitted papers. We would also like to thank all the authors who submitted their work to the workshop and hence contributed to its success. We wish you success in your scientific careers and we hope that the workshop will help you develop your ideas further. This year, JRWRTC accepted six peer-reviewed papers, which cover various topics of the real-time field such as scheduling, WCET analysis, time predictability, mixed criticality and sensor networks.

Yet, organizing this workshop would not have been possible without the help of many people. First, we would like to thank Claire Maiza, Catherine Parent-Vigouroux, and Pascal Raymond, General Chairs of RTNS 2017 for their guidance. We would also like to thank the local organizing committee, Amaury Graillat, Sophie Quinton, Hamza Rihani, and Valentin Touzeau for having put their time in ensuring that all the details were smooth. We would also like to thank Enrico Bini and Claire Pagetti, Program Chairs of RTNS 2017, for their scientific work. We finally acknowledge Geoffrey Nelissen (CISTER, Portugal) for his precious advice in the organization of the event.

On behalf of the Program Committee, we wish you a pleasant workshop. May the environment be stimulating, with fruitful discussions and the presentation be enjoyable and entertaining.

Mitra Nasri, Max Planck for Software Systems
Guillaume Phavorin, Universit de Poitiers
JRWRTC 2017 Workshop Chairs

Program Committee

Matthias Becker	Mlardalen University, Sweden
Alessandro Biondi	Scuola Superiore Sant'Anna, Italy
Georg von der Bruggen	TU Dortmund, Germany
Simin Cai	Mlardalen University, Sweden
Dakshina Dasari	Robert Bosch GmbH, Germany
Frank Drr	IPVS, University of Stuttgart, Germany
Pontus Ekberg	Uppsala University, Sweden
Tomasz Kloda	INRIA, France
Jing Li	Washington University in St. Louis, USA
Morteza Mohaqeqi	Uppsala University, Sweden
Borislav Nikolic	CISTER/ISEP, Portugal
Abhilash Thekkilakattil	AtlasCopco Industrial Technique R&D, Sweden
Kecheng Yang	University of North Carolina at Chapel Hill, USA

Table of Contents

Multimode application on a reconfigurable platform

Introducing a new model and a first protocol

Joël Goossens
Faculté des Sciences of the Université libre de Bruxelles Brussels, Belgium
Joel.Goossens@ulb.ac.be

Xavier Poczekajlo*
Faculté des Sciences of the Université libre de Bruxelles Brussels, Belgium
Xavier.Poczekajlo@ulb.ac.be

ABSTRACT

We consider the new problem of multimode applications for reconfigurable platforms in the context of hard real-time scheduling. In this problem, the taskset **and the hardware** may change over the time whenever the current mode of the system changes. Ensuring schedulability here requires to prove *(i)* The system schedulability of every mode, *(ii)* That any allowed mode change can take place with respect to the given timing constraints. Solving *(i)* is a schedulability problem upon heterogeneous systems. We propose a model to formalise the whole problem, and a first protocol to run any allowed mode change in the system with respect to the timing constraints. Finally, we propose a validity test to ensure that the property *(ii)* is respected.

Keywords

Multimode application; Reconfigurable system; Hard real-time; Heterogeneous system

1. INTRODUCTION

Hard real-time systems become more and more complex. Their correctness must be proven to ensure the safety of the system. When using a classic mono-mode application, proving the system correctness often leads to over-approximation of the workload. On certain systems, where some functions are executed only in certain situations, it is useful to use a more realistic and advanced model. The multimode application model fulfils this role. For an example, the application of an airplaine system will have very different *modes* depending on whether the airplane is on the ground before the take-off, or in cruise. Splitting the whole taskset into several sub-tasksets allows more precise bounding on the overall load at any instant. This is crucial when defining the system requirements, and may lead to huge gains in term of system capacity.

On a multimode application, the running taskset changes over the system lifespan. It may be interesting to adapt the system as well to fit each taskset the best. Today's FPGAs allow run-time hardware reconfiguration at high rate. Even more interesting, FPGAs allow dynamic partial reconfiguration (DPR): an FPGA may be divided into several partitions each having different configurations, and a single partition may be reconfigured without jeopardising the whole system. [1] describes, in a low-level approach, how DPR may be used and also gives more details about the current level of performance of the existing FPGAs. It is also possible to use processors which can change their speed at run-time.

Related work. The survey [5] proposes various solutions for a multimode application on a *uni*processor system. More important, it unifies a vocabulary for mode change applications. For an example, the notions of *periodicity* and *synchronous* or *asynchronous* protocols are as useful in uniprocessor systems as in multiprocessor systems. Based on those notions, [3] proposes the first multiprocessors protocols: a synchronous protocol *SM-MSO* and an asynchronous protocol *AM-MSO*. It also computes an upper-bound for the *makespan* of a taskset on a given system, which represents the required time to scheduled the pending jobs during the mode change (at most one job per task). This upper bound is then refined in [2].

Multimode protocols, as they currently exist, only handle the transition phase. During each mode execution, a scheduler must be used to handle the taskset. Multiprocessor systems may be scheduled by partitioned, global, semi-partitioned, or clustered algorithms. The latest has been well studied for heterogeneous system, and [4] proposes LPG_{IM}. It is, to the best of our knowledge, one of the most efficient approach for this problem. LPG_{IM} considers the system as clusters of identical processors and assigns a sub-taskset to each cluster. Inside the cluster, a global scheduler may be used. The problem of heterogeneous system is hence reduced to identical multiprocessor systems. This approach may be used in the context of multimode application on reconfigurable systems.

To the best of our knowledge, no such model nor mode change protocol exist, where both the hardware and the software can change during the lifespan of the system.

Contributions. In this paper, we introduce the first model where the taskset *and* the hardware may change over the time. We also propose a first synchronous protocol for such application and its feasibility test.

2. MODEL

A reconfigurable multiprocessor system is a system composed of *partitions*. A partition can be any computing unit, as an FPGA partition or a general purpose processor. Each partition can implement a *configured processor* through reconfiguration. The *configured processor* behaviour is defined by its *configuration*. In the case of a processor with a fixed behaviour, it is modelised as a partition with only two configurations: *on* and *off*. From now on, *configured processors*

*Corresponding author.

will be denoted as *processors*.

A multimode application for a reconfigurable system is defined by a set of x different modes $M \stackrel{\text{def}}{=} \{M^1, M^2, ..., M^x\}$. Each mode $M^h \stackrel{\text{def}}{=} \langle \tau^h, S_h, \widetilde{\Theta}_h, \Delta_h \rangle$ has to execute a taskset τ^h with the given scheduler S_h on a system composed of a set of processors (through partitions reconfiguration) with respect to the configuration multiset $\widetilde{\Theta}_h \in \Theta^m$ (see section 2.2.2). When the system requires a mode change to M^h, it must be done within a delay of Δ_h unit of time.

The system activates and deactivates the mode with the following constraint: at any time, one and only one mode can be active in the system. When a mode is activated, its taskset is *enabled*.

2.1 Taskset model

The taskset $\tau \stackrel{\text{def}}{=} \cup_{h=1}^{x} \tau^h$ is composed of several sub-tasksets τ^h, each one being the specific taskset of the h^{th} mode.

Each taskset is composed of n_h tasks: $\tau^h \stackrel{\text{def}}{=} \{\tau_1^h, \ldots, \tau_{n_h}^h\}$.

Each task τ_i^h is a sporadic task, defined with three parameters $\langle C_i^h, D_i^h, T_i^h \rangle$ — a worst-case execution time, a relative deadline, and a minimum inter-arrival time.

When a mode's taskset is *enabled*, all its tasks are enabled. At most one taskset can be enabled at any instant in the system. When a task τ_i^h is enabled, it may release a job with respect to the minimum inter-arrival time T_i^h. When a taskset is disabled, all its tasks are disabled and may not release new jobs. However, if a job of a disabled taskset is not complete, it must be completed with respect to its absolute deadline. Those jobs are called *rem-jobs* in the following.

2.2 System model

2.2.1 Processors and partitions

The system is composed of m reconfigurable partitions denoted $P \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_m\}$. The m partitions can dynamically be configured as m processors with a specific configuration. This operation is denoted as *reconfiguration*.

A partition has a type, depending on whether the partition represents a specific FPGA partition, or an other type of computing unit. A partition of type ρ must always be configured in a specific configuration. If the partition is not used, it must be configured in the configuration $\theta_{\rho,0}$ later defined in Section 2.2.2.

The configured partitions form a system of m *configured processors* $\Pi \stackrel{\text{def}}{=} \{\pi_1, \pi_2, \ldots, \pi_m\}$, denoted as processors. Hence, they form an unrelated system of several *clusters*. A *cluster* is a set of identical processors, i.e., partitions from the same type sharing the same configuration.

P_h represents all the used partitions of a mode M^h and $P_{h,\text{cl}}$ represents the used partitions of a given cluster cl for the mode M^h.

2.2.2 Partitions configuration

Reconfiguration can occur during a transition phase. $\Theta \stackrel{\text{def}}{=} \cup_\rho \Theta_\rho$ represents the set of all available configurations set in the system. A configuration set Θ_ρ represents all the configurations available for partitions of type ρ (e.g. a partition of FPGA with a given number of logic blocks).

$\Theta_\rho \stackrel{\text{def}}{=} \{\theta_{\rho,0}, \ldots, \theta_{\rho,\ell_\rho}\}$ where:

- $\theta_{\rho,0}$ is the configuration where the partition is off, on which no task can be scheduled. A partition configured with $\theta_{\rho,0}$ is *free*.
- $\theta_{\rho,1}, \ldots, \theta_{\rho,\ell_\rho}$ represent ℓ_ρ different configurations. Each configuration executes the different tasks at an *unrelated* speed, i.e., the speed of the processor configured as such depends on the job being executed. Its partition is *used*.

δ_z represents the delay to change to a specific configuration $z \in \Theta$. During this reconfiguration time, the processor cannot execute any task. It is important to note that the reconfiguration time does not depend on the previous configuration of the partition. The reconfiguration time varies because the partitions may be from different hardwares like different FPGA models.

For any partition p, $\theta(p)$ represents the current configuration of p. $\widetilde{\Theta}_{h,\rho}$ represents the configurations of mode h for the partitions of type ρ.

2.2.3 Tasks progression rate

Because the system is heterogeneous, the job speed depends on the type of the processor which executes the job. Those data are an input of the schedulers which are black boxes. Hence, there are omitted because of space constraints.

2.3 Mode transition

The system may receive a *Mode Change Request* $\text{MCR}(h)$ to the destination mode h, whenever the system is not handling a transition phase. This instant is denoted $t_{\text{MCR}(h)}$.

The transition graph represents all the possible configuration transitions. A transition between M^{src} and M^{dst} is possible if and only if there is an edge from M^{src} to M^{dst} in the transition graph, denoted by $(M^{\text{src}}, M^{\text{dst}})$.

The system enters a reconfiguration phase at $t_{\text{MCR(dst)}}$. It must then reconfigure itself according to the new mode within the given delay Δ_{dst}. After this delay, the destination mode M^{dst} is activated and the transition phase ends.

For a given mode M^h, $\widetilde{\Theta}_h$ contains the required configurations for the execution of τ^h.

After a *Mode Change Request* to mode M^h which occurred at $t_{\text{MCR}(h)}$, the system must be reconfigured before the instant equal to $t_{\text{MCR}(h)} + \Delta_h$ to be feasible.

3. PROTOCOL

3.1 Protocol

The mode-change protocol must ensure that the rem-jobs are correctly scheduled and that the system is able to activate the new mode within the given delay. For that purpose, it is composed of an offline computation phase, and of two run-time phases. The offline phase computes for each couple $(M^{\text{src}}, M^{\text{dst}})$ of the source mode M^{src} the necessary reconfigurations.

The run-time phase 1 begins at $t_{\text{MCR(dst)}}$ and is completed for each cluster, when all of their processors are idle.

The run-time phase 2 is the reconfiguration of the required partitions.

3.1.1 Hypothesis on the schedulers

We consider in this protocol only clustered schedulers. Unlike global schedulers, clustered schedulers allow tasks to migrate *only* between the processors of the cluster where the task is statically assigned. In addition, we consider only

schedulers that are preemptive, fixed-job priority and work-conservative. We use the notions defined in [3].

The taskset of a mode h is divided into several sub-tasksets, one per cluster: $\tau^h = \cup_{\text{cl}}\ \tau^{h,\text{cl}}$. The scheduler S_h may use a specific scheduling policy for each cluster cl which respects the three assumptions aforementioned. Each scheduling policy must schedule the sub-taskset $\tau^{h,\text{cl}}$ of each cluster cl feasibly. Every $\tau^{h,\text{cl}}$ is determined at design time.

3.1.2 Offline computation

The offline computation creates two tables per couple $(M^{\text{src}}, M^{\text{dst}})$ from the transition graph: the Empty Partitions Reconfigurations table (EPRT) and the Used Partitions Reconfigurations table (UPRT). Those tables are necessary to reconfigure the system after a MCR(dst) when the mode M^{src} is active.

Each table depends on both the source mode M^{src} and the destination mode M^{dst}. Their computations use the makespan upper bound of each cluster of M^{src} (see [3]).

The Empty Partitions Reconfigurations table contains a list of configurations required by M^{dst} not used by M^{src}. Those reconfigurations will be launched at $t_{\text{MCR(dst)}}$. Its computation is described by the Algorithm 1.

The Used Partitions Reconfigurations table contains a list of couple of configurations. A couple $(\theta_{z_1}, \theta_{z_2})$ in the UPRT indicates that a partition configured in θ_{z_1} will be reconfigured in θ_{z_2}. Its computation is described by the Algorithm 2.

Algorithm 1: Creation of the EPRT for $(M^{\text{src}}, M^{\text{dst}})$

```
1  Input: M^src: the source Mode
2         M^dst: the destination Mode
3  Output: EPRT: the EPRT for (M^src, M^dst)
4
5  begin
6    let EPRT be an empty multiset
7
8    For each type of partition ρ
9      let Θ⃗_{dst,ρ} ≝ Θ̃_{dst,ρ} \ Θ̃_{src,ρ} ∩ Θ̃_{dst,ρ}: a vector
10     Order Θ⃗_{dst,ρ} by reconfiguration time,
11          in decreasing order
12     let free = the number of free partitions
13          of type ρ for the mode M^src
14     Add the free^th first elements of Θ⃗_{dst,ρ}
15          in EPRT
16 end
```

Algorithm 2: Creation of the UPRT for $(M^{\text{src}}, M^{\text{dst}})$

```
1  Input: M^src: the source Mode
2         M^dst: the destination Mode
3  Output: UPRT: the UPRT for (M^src, M^dst)
4
5  begin
6    let UPRT be an empty multiset
7
8    For each cluster cl of M^src
9      For each p ∈ P_{src,cl}
10       makespan(p) ≝ makespan(cl)
11
12   For each type of partition ρ
13     let Θ⃗_{src,ρ} the vector of config.
14       for used partitions of type ρ in M^src
15     Order Θ⃗_{src,ρ} by makespan
16     let Θ⃗_{dst,ρ} ≝ Θ̃_{dst,ρ} \ Θ̃_{src,ρ} ∩ Θ̃_{dst,ρ}: a vector
17     Order Θ⃗_{dst,ρ} by reconfiguration time,
18          in decreasing order
19     let free = the number of free partitions
20        of type ρ for the mode M^src
21     Remove from Θ⃗_{dst,ρ} the free^th first el.
22     if |Θ⃗_{dst,ρ}| > 0
23       For each n^th el. of Θ⃗_{dst,ρ}
24         let θ_{dst,ρ} = Θ⃗_{dst,ρ}(n)
25         add to UPRT: (Θ⃗_{src,ρ}(n), θ_{dst,ρ})
26 end
```

3.1.3 Run-time phase 1: Schedule rem-jobs

This phase relies heavily on the SM-MSO protocol from [3].

At the MCR(M^{dst}), the protocol deactivates the mode M^{src} and disables all the current enabled tasks. Then, it keeps the scheduler S^{src} to schedule the rem-jobs until idle-time is reached for every processor.

Because the scheduler S^{src} can schedule τ^{src} with no deadline miss: the rem-jobs will be feasibly scheduled by using the same scheduler.

3.1.4 Run-time phase 2: Reconfiguration

At the MCR(M^{dst}), the unused partitions of M^{src} are reconfigured. For each element θ_{z_1} in UPRT, the protocol reconfigures a free partition to the configuration θ_{z_1}.

When all the processors of a cluster are idle, the protocol launches its required reconfigurations. Each partition p of the cluster picks, if possible, a couple $(\theta_{\text{src}}, \theta_{\text{dst}})$ in the EPRT where the source configuration θ_{src} is the current configuration of p, i.e., $\theta_{\text{src}} = \theta(p)$. An entry of the EPRT can be picked only once per transition phase. Then, the partition p is reconfigured to the destination configuration θ_{dst}.

When all the cluster are idle, the system can safely activate the mode M^{dst}, and enables all the tasks of τ^{dst}.

3.1.5 Example

To illustrate our protocol, we show an example of a multimode application with two modes M^1, M^2 (see Figure 1). The system is allowed to change from M^1 to M^2, and we show how works the transition phase. For that purpose, here are the partial specifications of the system:

- For mode M^1: $\widetilde{\Theta}_1 = \{\theta_0, \theta_1, \theta_1, \theta_2\}$, $\tau^1 = \{\tau^{1,1}, \tau^{1,2}\}$, where $\tau^{1,1} = \{\tau_1, \tau_2, \tau_3\}, \tau^{1,2} = \{\tau_4, \tau_5\}$;
- For mode M^2: $\Delta_2 = 5.5$, $\widetilde{\Theta}_2 = \{\theta_2, \theta_4, \theta_4, \theta_5\}$, $\tau^2 = \{\tau^{2,1}, \tau^{2,2}, \tau^{2,3}\}$, where $\tau^{2,1} = \{\tau_6, \tau_7, \tau_8\}$, $\tau^{2,2} = \{\tau_9\}$, $\tau^{2,3} = \{\tau_{10}\}$;
- Reconfiguration time for θ_4, θ_5: $\delta_{\theta_4} = 2$, $\delta_{\theta_5} = 1$;
- Makespan upper-bound for mode M^1 clusters: $makespan(\tau^{1,1}) = 3.5$, $makespan(\tau^{1,2}) = 4.5$;
- EPRT$(M^1, M^2) = \{\theta_4\}$,
  UPRT$(M^1, M^2) = \{(\theta_1, \theta_4), (\theta_1, \theta_5)\}$;
- Tasks specification are omitted because of space constraint and relevance;
- All the partitions have the same type and hence the same available configurations.

At $t = 0$, each active task releases a job.

At $t = 3$, τ_3 releases a new job.

At $t = 4$, τ_1, τ_2 and τ_4 release a new job.

At $t = 4.5$ a MCR(2) occurs: the mode M^2 must be activated by $t_{\text{MCR}(2)} + \Delta_2 = 4.5 + 5.5$. However, $\tau_1, \tau_2, \tau_3, \tau_4$ have active rem-jobs. For an example, the second job of τ_1 was released at 4 and must be completed. The scheduler used by M^1 is kept to schedule the rem-jobs. p_1 is free in the mode M^1: it is immediately reconfigured to a configuration in the EPRT: θ_4.

At $t = 5.5$, the cluster $\tau^{1,2}$ is idle, but no couple (τ_2, X) exists in the UPRT: p_4 is not reconfigured. At $t = 7$, the

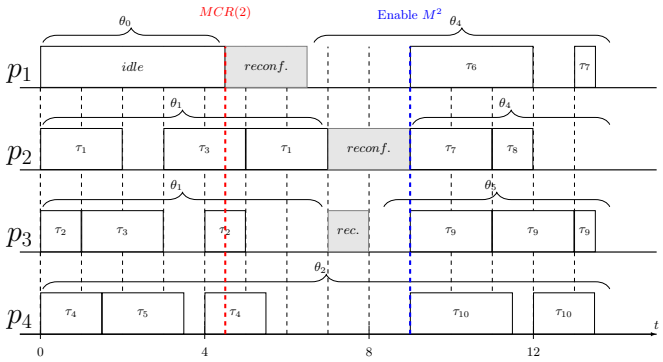

Figure 1: Example of a Multimode application with reconfigurations

cluster $\tau^{1,1}$ is idle, so each configuration picks a couple in the EPRT: (θ_1, θ_4) and (θ_1, θ_5).

At $t = 9$, all the partitions are idle. The mode M^2 is activated, and its taskset τ^2 enabled. Because $t_{\mathrm{MCR}(2)} + \Delta_2 \geqslant 9$, the transition delay time constraint was respected.

3.2 Validity test

We provide a validity test for our protocol. The validity test is a sufficient condition which indicates whether a given multimode application is feasible or not, i.e., if all the deadlines of the rem-jobs will be met and if the transition phase delay will be respected, for any mode change allowed by the transition graph.

By construction, all the deadlines will be met during a transition phase. However, we must provide an upper bound of the transition phase for every mode M^{dst}. This upper bound depends on the upper-bound of the makespan of a cluster $\mathrm{ms}(\mathrm{src}, \mathrm{cl})$, defined by [2] (see Corollary 2.2). Without loss of generality, we assume the tasks to be ordered by processing time.

$$\mathrm{ms}(\mathrm{src}, \mathrm{cl}) \stackrel{\mathrm{def}}{=} \begin{cases} c_{|\tau^{\mathrm{src,cl}}|}, \text{ if } |\tau^{\mathrm{src,cl}}| = |P_{\mathrm{cl}}| \\ \frac{\sum_{i=1}^{|\tau^{\mathrm{src,cl}}|-1} c_i}{|P_{\mathrm{cl}}|} + c_{|\tau^{\mathrm{src,cl}}|}, \text{ otherwise} \end{cases}$$

For any couple $(M^{\mathrm{src}}, M^{\mathrm{dst}})$, the upper bound for an empty partition reconfiguration (EPR-UB) is:

$$\text{EPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}) \stackrel{\mathrm{def}}{=} \max(\{\forall z \in \widetilde{\Theta}_{\mathrm{src}}\ \delta_z\})$$

For any couple $(M^{\mathrm{src}}, M^{\mathrm{dst}})$, the upper bound for an used partition reconfiguration (UPR-UB) is:

$$\text{UPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}) \stackrel{\mathrm{def}}{=} \max(\{\forall \rho\ \text{UPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}, \rho)\})$$

with

$$\text{UPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}, \rho) \stackrel{\mathrm{def}}{=} \max(\{\forall i = 0..|\widetilde{\Theta}_{\mathrm{dst},\rho}|\ \delta_{z_{i,\rho}} + \mathrm{UPM}(\rho, i)\})$$

where

- z_{i_t} is the i^{th} element of the table of configurations of M^{dst} for partitions of type ρ, ordered by reconfiguration time in decreasing order and
- $\mathrm{UPM}(\rho, i)$ is the i^{th} element of the vector of the upper bound makespan of each partition of type ρ of P_{src}, ordered by duration increasing.

Therefore, for any couple $(M^{\mathrm{src}}, M^{\mathrm{dst}})$, the upper bound of a transition phase is:

$$\mathrm{UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}) \stackrel{\mathrm{def}}{=} \max(\text{EPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}), \text{UPR-UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}))$$

Hence, the upper bound for any transition allowed to M^{dst} is:

$$\max(\{\mathrm{UB}(M^{\mathrm{src}}, M^{\mathrm{dst}}) | (M^{\mathrm{src}}, M^{\mathrm{dst}}) \in \text{TransitionGraph}\})$$

4. CONCLUSION AND FUTURE WORK

The technical advance for FPGAs leads us to consider a new paradigm for multimode application, with a system that is reconfigured in an efficient way for each mode. To the best of our knowledge, no such model currently exists. In this paper, we introduce the first model which fills that gap. This model can be seen as an extension of the multimode application model for multiprocessor systems. We also propose a synchronous protocol, associated with a validity test for any application and system on the new model. This protocol is more than a simple generalisation of existing protocols, using the makespan of each cluster to tighten the transition phase delay, and thus allowing more applications to pass the validity test provided by this article.

Future work. A first contribution would be to tighten the validity test by computing a makespan for each processor rather than each cluster. Furthermore, we intend to propose a more generic model. Our first model can be extended via several parameters. We want to be able to handle schedulers which allow inter-cluster migrations. We also want to introduce mode independent tasks to describe tasks that need to release jobs at any instant in the system, even during a transition phase (see periodicity in [5]). To fully take advantage of the mode independent tasks, we intend to propose an asynchronous protocol with periodicity.

In the current model, the different partition types do not share any configuration. A configuration for an FPGA partition can be used on an other FPGA partition with more logic blocks. It may be interesting to have configuration that are usable by different partition types to gain more flexibility.

5. REFERENCES

[1] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. C. Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *Real-Time Systems Symposium*, pages 1–12, 2016.

[2] V. Nélis. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems*. PhD thesis, Université libre de Bruxelles, 2010.

[3] V. Nélis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Euromicro Conference on Real-Time Systems*, pages 151–160, 2009.

[4] G. Raravi, B. Andersson, V. Nélis, and K. Bletsas. Task assignment algorithms for two-type heterogeneous multiprocessors. *Real-Time Systems*, 50(1):87–141, 2014.

[5] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

Towards statistical estimation of worst case inter-core communications

Anselme Revuz*
Inria and University Paris-Est
2 rue Simone Iff
Paris, France
anselme.revuz@gmail.com

Liliana Cucu-Grosjean†
Inria
2 rue Simone Iff
Paris, France
liliana.cucu@inria.fr

ABSTRACT

Statistical approaches have recently received increased effort from the real-time community. While taking into account that worst case scenarios are rare events, the statistical approaches may decrease considerable the pessimism of a real-time analysis. In this paper we present first results on the statistical estimation of inter-core communications on a multicore platform. Our preliminary results are based on the utilization of the Extreme Value Theory and they are promising as they allow to detect insufficient observations of the communications.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** •**Networks** → Network reliability;

Keywords

probabilistic worst case execution time, multicores, manycores

1. INTRODUCTION AND RELATED WORK

The arrival of new complex architectures, as an answer to more and more functionalities request by end users, imposes the appearance of new time analyses to real-time designers. Most architectures include today cache memories, instructions pipelines to cite some common features that have a direct impact on the execution time of a program on a processor. Mainly conceived to ensure excellent average time behaviour, these features may increase in an important way the worst case execution times of programs. Following the same evolution, multicores and manycores architectures are increasing the calculation capacity by adding inter-core communication costs to the time behaviour of the programs.

Due to intellectual property concerns but also not sufficiently analyzed time behaviour, the real-time designers search for new analysis techniques to solve the problem of time estimation. Complete models of the architectures could be proposed but their models may bring higher complexity to the analyses and also increased pessimism. Indeed the worst-case scenarios are difficult to model and also may have a low probability of appearance. As a result, probabilistic and statistical methods seem promising as they take into account this low probability of appearance of a worst case situation.

*MSc student
†Researcher

In this context, approaches taking into account the probability of appearance of a worst case execution time have been proposed and different existing schedulability results are applied for job-level probabilistic execution times [5], task-level probabilistic worst case execution times [9] or taking into account dependences [10]. While probabilistic worst case execution time estimation has received an important effort either by measurement-based reasoning [6] [7] [12] [4] [3] [11] or static reasoning [1] [2], different statistical tests are used within current measurement-based solutions.

From our best knowledge the problem of inter-core communications is yet not received any solution based on the utilization of statistical approaches considering worst case estimations.

Our contribution: In this paper we present first arguments in favor of statistical estimation of inter-core communications when complete models are difficult to propose or their utilization is too costly because of the pessimistic hypotheses.

Organization of the paper: The paper is organized as follows. In Section 2 we present the main definitions and notations required by our contribution. We remind the statistical theory of Extreme Value Theory in Section 3. Numerical experiments are presented in Section 4 which allows us to conclude in the last section of the paper.

2. PROBABILISTIC WORST CASE EXECUTION TIME AND PROBABILISTIC WORST CASE EXECUTION COMMUNICATION TIME

In this section we first provide the classical definition of execution time and worst case execution time of a program and based on these formulations, we propose a similar definition for inter-core communications.

Definition *The probabilistic execution time (pET) of the instance of a program describes the probability that the execution time of that instance is equal to a given value.*

For instance the j^{th} instance of a program τ_i may have a pET

$$\mathcal{C}_i^j = \begin{pmatrix} 2 & 3 & 5 & 6 & 105 \\ 0.7 & 0.2 & 0.05 & 0.04 & 0.01 \end{pmatrix} \quad (1)$$

If $f_{\mathcal{C}_i^j}(2) = 0.7$, then the execution time of the j^{th} instance of τ_i has a probability of 0.7 to be equal to 2.

The definition of the probabilistic worst-case execution time (pWCET) of a program is based on the relation $\succeq$

between two probability distributions, provided in the definition below.

Definition [8] Let $\mathcal{X}$ and $\mathcal{Y}$ be two random variables. We say that $\mathcal{X}$ is worse than $\mathcal{Y}$ if $F_{\mathcal{X}}(x) \leq F_{\mathcal{Y}}(x)$, $\forall x$, and denote it by $\mathcal{X} \succeq \mathcal{Y}$. Here $F_{\mathcal{X}}$ is the cumulative distribution function of $\mathcal{X}$.

In Figure 1 $F_{\mathcal{X}_1}(x)$ never goes below $F_{\mathcal{X}_2}(x)$, meaning that $\mathcal{X}_2 \succeq \mathcal{X}_1$. Note that $\mathcal{X}_2$ and $\mathcal{X}_3$ are not comparable. By CDF we understand the cumulative distribution function of a random variable.

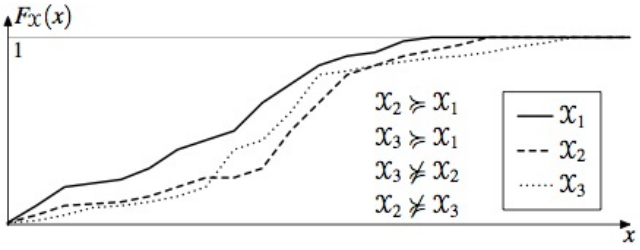

Figure 1: Possible relations between the CDFs of various random variables

Definition *The probabilistic worst case execution time (pWCET) $\mathcal{C}_i$ of a program τ_i is an upper bound on the pETs $\mathcal{C}_i^j$ of all instances of τ_i $\forall j$ and it may be described by the relation $\succeq$ as $\mathcal{C}_i \succeq \mathcal{C}_i^j$, $\forall j$.*

Graphically this means that the CDF of $\mathcal{C}_i$ stays under the CDF of $\mathcal{C}_i^j$, $\forall j$. Thus the **probabilistic worst case execution time is upper bounding all probabilistic execution times** of a program.

Definition *The probabilistic inter-core communication time (pCT) of a program describes the probability that the communication time required for the program to finish its execution is equal to a given value.*

Definition *The probabilistic worst case communication time (pWCCT) of a program is an upper bound on all possible pCTs of that program.*

3. MEASUREMENT-BASED APPROACH AND EXTREME VALUE THEORY

A measurement-based approach for estimating the pWCCT of a program has two main parts: (i) collecting the communication time traces of the program; and (ii) estimation the pWCCT based on the set of communication time traces obtained during the first step.

The second step is done in our paper by using the Extreme Value Theory (EVT) [6] [7] [12] [4]. According to EVT if the maximum of communication times of a program converges, then this maximum of the communication times $\mathcal{C}_i, \forall i \geq 1$ will converge to one of the three possible curves described in Figure 2: Fréchet, Weibull and Gumbel corresponding to a shape parameter $\xi < 0$, $\xi > 0$, and $\xi = 0$, respectively.

EVT has two different formulations: Generalized Extreme Value (GEV aka BM) and Generalized Pareto Distribution (GPD). BM is based on the block maxima reasoning that groups the communication times by smaller groups and only the largest value of each group is considered for the pWCCT estimation (see Figure 3). GPD is a (graphical) method based on the threshold approach that considers only the values larger than the chosen threshold for the pWCCT estimation (see Figure 4).

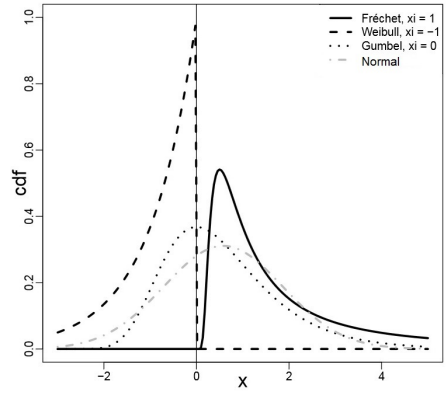

Figure 2: The three possible upper-bounds of a set of $\mathcal{C}_i, \forall i \geq 1$ of the same program

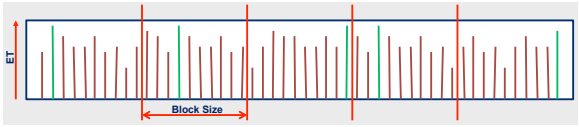

Figure 3: BM keeps the largest value for each block

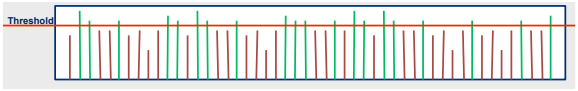

Figure 4: GPD keeps all values above the threshold

Existing work considers the execution time traces obtained mainly in isolation. **Our contribution is obtaining the communication time traces while different instances of different programs are executed on several cores**. The context of our numerical experiments is detailed in Section 4.

4. EXPERIMENTS

Our experiments on inter-core communications have been done on Kalray MPPA-256 processor. The MPPA-256 processor has 16 clusters, each of which contain set of 16 working cores and 1 resource manager core. It contains also 4 sub-sets I/O which handle external communications. Thus MPPA-256 has 256 working cores for an overall of 288 cores (management and working cores together).

In order to facilitate the communication between clusters, two Networks on Chip are used. The first NoC transfers control information while the second NoC is used to transfer data. A global view of the processor is provided in Figure 5[1].

Our measurements concern the time of transfer between cores within the same cluster of the MPPA processor. We use several cores and each executes its own thread while only two cores are specifically used to measure. The first core lunches a counter and it sends a signal to the second core (which executes a part of the program started on the first core). This second core sends back a signal to the first core that stops its counter once the signal is received. In our case, we measure the time since the start of the signal from the first core to the arrival back to the first core of the answer from the second core. In order to make our experiments close

[1]Kalray Courtesy

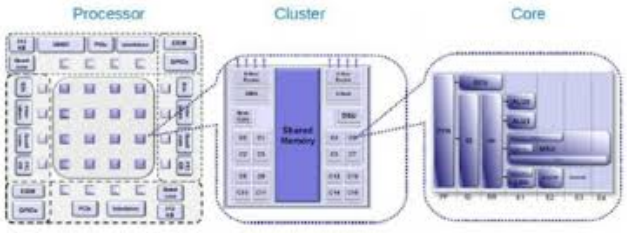

Figure 5: Global view of the MPPA-256

to the reality, we have also introduced interferences on other cores: other cores send also signals more or less regularly to each other during the measurements, which increases the time transfer of the original signal (the measured one).

In reality the measures should be representative for all input values and for all execution environments. In our case the tests are simple with no input value and the different execution environments are obtained by varying the interference level.

Once our measurements obtained, we use the EVT tool to propose estimations of the probabilistic worst case distributions.

For a first scenario with low interferences we have obtained time values between 901[2] and 1392 for the same program executed between the same two cores, the average value being 1070. For the scenario with high interferences we have obtained time values from 1064 and 1792 for the same program executed between the same two cores, the average value being 1395. These values are resumed in the table below for an easier comparison.

Scenario	Minimum	Maximum	Average
With low interferences	901	1392	1070
With high interferences	1064	1792	1395

The statistical estimation of the worst-case inter-core communication time is done by using the Extreme Value Theory (see Section 3) and more precisely we use the implementation available at `http://inria-rscript.serveftp.com`. The web page of the tool requires a secured connection using the login *aoste* and the password *aoste*.

Both sets of measured time values (with low and with strong interferences) provide set of values that are identically distributed but not independent. The independence hypothesis is not mandatory to apply the Extreme Value Theory and the tool used is taking into account this specific case. The estimated probability distributions are presented in Figure 6, respectively, Figure 7.

In Figure 6, the blue line indicates the probability distribution of the measured values. The two versions of the Extreme Value Theory (BM in black and GPD in red) provide extreme estimations that are not converging. This comes from the statistical dependences which has an important impact in this case on the BM method. Thus for the scenario with low interferences, new measurements are required to understand the dependences between the communication times. With our current understanding we are not able to indicate if they are coming from the measurement protocol or scheduling choices of the processor. This is, thus, left as future work.

[2]The time values are given as number of core cycles.

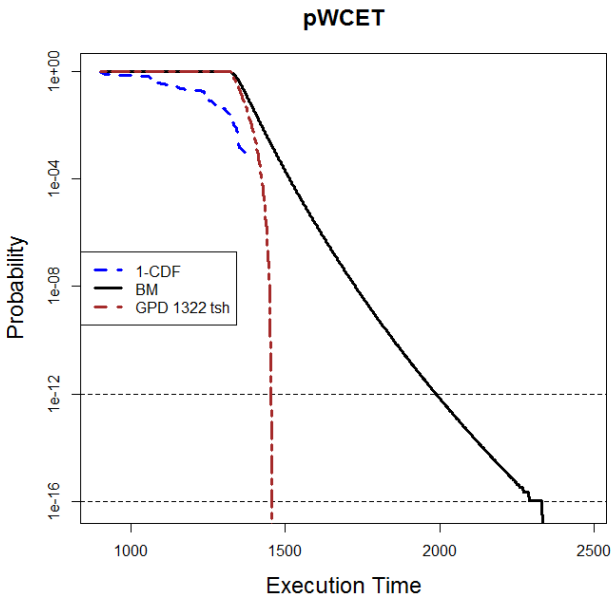

Figure 6: Statistical estimation of inter-core communications with low interferences

In Figure 7, the blue line indicates the probability distribution of the measured values. The two versions of the Extreme Value Theory (BM in black and GPD in red) provide extreme estimations that are converging. This indicates that estimated probability distribution is close to an achievable worst case as the measured communication times are not very different from the estimated extremes. This also indicates that our interferences are heavy and less heavier interferences should be considered. This is also identified as future work.

5. CONCLUSIONS

In this paper we have presented first results on the utilization of the Extreme Value Theory for the statistical estimation of worst-case inter-core communications on multicore processors. After detailing the context of our work, we have introduced the Extreme Value Theory. Our experiments concern a Kalray processor and we have provided its description. Our statistical estimations indicate that (1) the scenario with low interferences requires a better understanding of the dependences between the communication times and (2) the scenario with high interferences is pessimistic, less heavier interferences should be considered in the future.

6. ACKNOWLEDGMENTS

The authors would like to thank Amaury Graillat and Mihail Asavoae for their help on the utilization of the Kalray processor. Moreover the authors would like to thank Adriana Gogonel and Cristian Maxim for their support on the utilization of the EVT tool.

This research and its related results have been supported by the French LEOC collaborative project Capacites.

7. REFERENCES

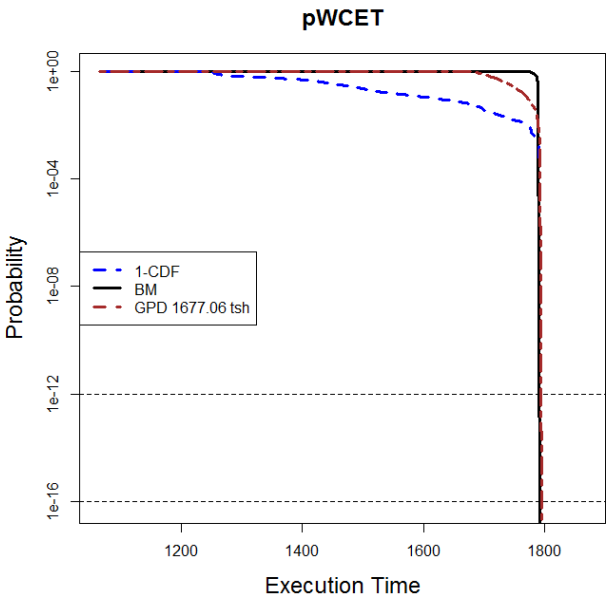

Figure 7: Statistical estimation of inter-core communications with high interferences

[1] S. Altmeyer, L. Cucu-Grosjean, and R. Davis. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Systems*, 51(1):77–123, 2015.

[2] S. Altmeyer and R. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, 2014.

[3] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *22nd International Conference on Real-Time Networks and Systems*, page 279, 2014.

[4] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *the 24th Euromicro Conference on Real-time Systems*, 2012.

[5] J. Díaz, D. Garcia, K. Kim, C. Lee, L. Bello, L. J.M., and O. Mirabella. Stochastic analysis of periodic real-time systems. In *the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, 2002.

[6] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium*, 2001.

[7] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[8] J. López, J. Díaz, J. Entrialgo, and D. García. Stochastic analysis of real-time systems under preemptive priority-driven scheduling. *Real-Time Systems*, pages 180–207, 2008.

[9] D. Maxim and L. Cucu-Grosjean. Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *the IEEE Real-Time Systems Symposium*, 2013.

[10] A. Melani, E. Noulard, and L. Santinelli. Learning from probabilities: Dependences within real-time systems. In *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, ETFA*, pages 1–8, 2013.

[11] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the sustainability of the extreme value theory for WCET estimation. In *14th International Workshop on Worst-Case Execution Time Analysis*, pages 21–30, 2014.

[12] L. Yue, I. Bate, T. Nolte, and L. Cucu-Grosjean. A new way about using statistical analysis of worst-case execution times. *ACM SIGBED Review*, September 2011.

Reducing the gap between theory and practice : towards A Proven Implementation of Global EDF in Trampoline

Khaoula BOUKIR
University of Nantes
LS2N UMR 6004
1 rue de la Nöe - BP 9210
Nantes 44321, France
khaoula.boukir@ls2n.fr

Jean-Luc BÉCHENNEC
CNRS
LS2N UMR 6004
1 rue de la Nöe - BP 9210
Nantes 44321, France
jean-luc.bechennec@ls2n.fr

Anne-Marie DÉPLANCHE
University of Nantes
LS2N UMR 6004
1 rue de la Nöe - BP 9210
Nantes 44321, France
anne-marie.deplanche@ls2n.fr

ABSTRACT

With technological progress, multiprocessor and multicore computing platforms are nowadays largely used in the majority of domains especially in real-time systems and embedded systems. This has introduced an increasing number of scientific researches in multiprocessor real-time scheduling. However, most of results are mainly theoretical and very few implementations on real systems have been studied.

In this paper we present our implementation of a multiprocessor scheduler, based on global EDF policy, in an OSEK/VDX real-time operating system called Trampoline. It's a preliminary step since we intend to support this implementation with a formal verification of its correctness, after that we consider to extend this study to other scheduling policies which are more "sophisticated".

CCS Concepts

•**Computer systems organization** → *Real-time operating systems;*

Keywords

Real-time scheduling, EDF, RTOS, Trampoline

1. INTRODUCTION

In embedded computing field, many standards for real-time systems have emerged to provide solutions to the increasing complexity of embedded electronic architecture. The daily challenge is to satisfy high performance requirements and to be able to execute in dynamic environments where the characteristics of the computational load cannot be precisely predicted.

This has pushed researchers and developers to consider multiprocessor and multicore architectures for executing real-time applications. In real-time scheduling community, many multiprocessor policies have been proposed, offering the optimality (allowing a total use of processing units and guaranteeing the satisfaction of all temporal constraints). Also a lot of theoretical studies have proven that a lot of global scheduling policies offer this property better, compared to partitioned one [1], since it has the advantage of automatically performing load balancing between processors. However, small attention has been given to the implementation of such policies within a real platform. This is due to the complete abstraction of implementation constraints made in the literature (overheads, data structure choices, scheduling events handling, etc.). This gap between the theoretical model and the reality does not encourage the adoption of these policies within real systems. Therefore, it is essential to confront them with real targets in order to objectively evaluate their interest.

In this context, our goal is to study in depth the implementation of global multiprocessor scheduling policies within a real platform. Moreover we wish to pursue this research with a formal study of the scheduler's behaviour using verification tools, in order to prove the correctness of our implementation.

The rest of the paper is organized as follows. Section 2 provides a short overview of real-time systems in general and Trampoline RTOS. Section 3 presents our goals. Section 4 discusses the choices of implementation to integrate a global scheduler. Finally section 5 states our perspectives and the future prospects of our research.

2. BACKGROUND

This section introduces the principle of real-time scheduling and presents Trampoline real-time operating system.

2.1 Survey of real-time multiprocessor scheduling

2.1.1 Multiprocessor real-time scheduling

Real-time scheduling essentially refers to determining the order according to which tasks are to be executed on the processor(s) such that the temporal constraints are satisfied. The decision of this order is the result of scheduling algorithms that calculate the priority of every job.

Note that scheduling algorithms can be classified based on the target platform on which tasks are to be run. Accordingly the classes of scheduling algorithms are : **uniprocessor scheduling** in which the problem is to determine a temporal allocation of the tasks on a single processor ; **multiprocessor scheduling** for which the execution is to be run on several processors, thus the problem is to define a spatial allocation of tasks on processors and then a temporal allocation on each processor.

In multiprocessor scheduling, there are three approaches for task scheduling: **partitioned scheduling** in which each task is statically assigned to one processor and no migration

of tasks is permitted ; **global scheduling** in which tasks compete for the use of all processors, hence migration is allowed ; **hybrid scheduling** which is a cross between the former two approaches.

In global scheduling, the first algorithms proposed were a generalization of uniprocessor algorithms such as EDF or RM in multiprocessor. However, it has been shown that this extension does not allow a better use of processing units than partitioned algorithms [2]. In the early 1990s, new approaches for global scheduling have emerged introducing the "fairness" of execution. The fair policies, such as PFair [3] enable reaching optimality by imposing a certain execution rhythm, they come close to the fluid execution but at the expense of significant number of migrations. Recently, other policies like U-EDF [4] have shown that it is possible to preserve the optimality by releasing the fairness. Nonetheless, it is more complicated to calculate the scheduling decision in these policies.

2.1.2 GLobal EDF scheduling

Global EDF (for Global Earliest Deadline First) is a global scheduler that generalizes the well-known EDF policy [5] in multiprocessor, for which the highest priority job is the one with the earliest absolute deadline. Thus, for Global EDF, m jobs with the closest deadline are executed on m free processors.

As a first step in our study, we are interested in the implementation of this policy.

2.2 Trampoline RTOS

Trampoline [6] is an open source real-time operating system that implements the OSEK/VDX standard and its successor AUTOSAR. It was developed in Real-Time Systems team of LS2N for academic purposes.

2.2.1 Trampoline architecture

The architecture of Trampoline RTOS contains three major layers as shown in Fig. 1: *API (for Application Programming Interface)* which gathers all the services exposed to the application ; *Kernel* that contains all the low-level functions required to: start the OS, start/stop, schedule or synchronise the processes (tasks) ; *BSP (for Board Support Package)* which depends on the target machine has 4 components: the external interrupt handler, the system call handler, the context switch manager and memory protection manager.

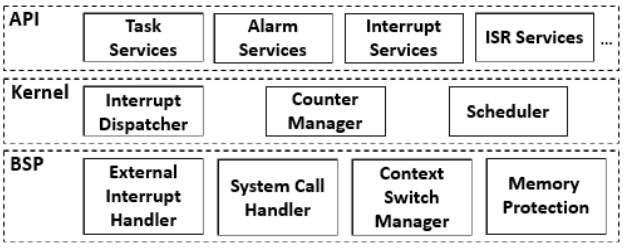

Figure 1: Trampoline architecture

Note that Trampoline RTOS is currently portable to different embedded targets: ARM7, AVR (Atmel), ARM - Cortex-M, ARM Cortex-A, PowerPC, POSIX etc.

2.2.2 Scheduling

As mentioned, Trampoline is an OSEK/VDX based operating system, thus it implements a basic preemptive scheduling policy based on fixed priority and partitioned scheduling in multiprocessor. The priority levels of tasks are statically assigned.

Trampoline's *Scheduler* handles a list to store the descriptors of ready tasks in uniprocessor and one list per core in multiprocessor. This list is implemented as a table of FIFOs indexed by the priority of tasks. The descriptors of the tasks which are newly activated or released from an event waiting are placed in their priority level FIFO in the order of their activation date. The preempted tasks are placed in the head of the FIFO.

Trampoline's *Scheduler* uses another structure *tpl_kern* which stores all the informations about the running tasks.

3. OUR GOAL

The main goal of our work is to try to bring the theory into practice in real-time scheduling, mainly by verifying with model-checking whether the scheduling properties announced by theoretical studies can still be preserved when implemented. For that the model needs to be as close as possible to the implementation. We will mainly investigate the implementation of global multiprocessor scheduling policies.

Thus we first intend to modify Trampoline so that it can support global scheduling policies. After that we consider to verify, based on a formal model, several functional and temporal properties such as : the scheduler produces the correct execution sequence, the scheduler decisions don't lead to a deadlock, a task is not executed at the same time by more than one processor, etc.

Trampoline has already been modeled by Tigori [10]. In this model all the source code of the OS is abstracted by a combination of UPPAAL's functions [11] and a network of extended finite automata; The variables used in the model are the control variables of the operating system; Actions and conditions associated with each transition are the same actions and conditions of the operating system program and the imperative expression code associated with each transition of the automata is very close to the operating system code. Thus the model is very close to the actual source code and is a good basis for global scheduler implementation modeling.

4. IMPLEMENTATION DETAILS

In this section, the choices made for the implementation of Global EDF in Trampoline are presented. First, we will discuss the time representation and the computation of relative deadlines of the tasks. Then, a presentation of the data structure used to implement the list of ready tasks is provided, followed by a presentation of the scheduler's functions.

4.1 Time representation

Since Global EDF is based on job's absolute deadline dates, it requires a time management mechanism able to : 1) represent and compare such dates using the minimum memory possible; 2) handle the timer overflow without a large runtime overhead.

There are two different ways for time representation. In both of them, time is a variable represented using n bits and

a resolution :

- *linear time model*: where the time runs from 0 to 2^n-1. For example in POSIX systems, time is calculated as a signed 32-bit integer that started since 1 January 1970 and will end on 19 January 2038 [7]. Therefore, a solution should be predicted by then to handle this overflow.
- *circular time model*: unlike the linear model, this approach can handle the "*apocalypse*" by offering an infinite system lifetime[1], since the time progresses in cycles from 0 to $2^n - 1$ and drops to 0 again to avoid the overflow.

In our case, the system time is implemented using a circular time model with 32-bit variable and 50μ resolution forcing each cycle to have a length of a few days. An absolute deadline d_i is calculated whenever a job is activated by adding the corresponding relative deadline D_i to the current time t: $d_i = D_i + t$. The current time can be evaluated by the *ticks* of Trampoline's system counter. For comparing two deadlines, we decided to use the ICTOH algorithm [8] that Carlini and Butazzo proposed and proved its ability to handle timer overflows with a small overhead compared to other techniques.

The idea of this algorithm is that, if we consider two deadlines d_i and d_j represented by n-bit unsigned integers, their comparaison can be performed by evaluating the difference between them as an unsigned n-bit integer and compare it to the half of the system's lifetime $P/2$. Thus, we have :

1. if $unsigned(d_i - d_j) > P/2$ then d_i is before d_j.
2. if $unsigned(d_i - d_j) < P/2$ then d_i is after d_j.
3. if $unsigned(d_i - d_j) = 0$ then d_i and d_j are simultaneous.

This difference is also the distance evaluated from d_i to d_j on the time circle in the direction of increasing values. Note that the results of this comparison is valid only if $|d_i - d_j| < P/2$. In other words, task timing constraints can not exceed $P/2$ ticks. For example, if we evaluate deadline events d_1, d_2 and d_3 in Fig. 2 in 32-bit system, we have: $dist_a = unsigned(d_1 - d_2) = F3332CCC > 7FFFFFFF = P/2$ and $dist_b = unsigned(d_3 - d_2) = 266666CC < P/2$, which means that d_1 preceeds d_2, and d_3 succeeds d_2.

4.2 Scheduler data structures

The data structures that we need are especially for storing the ready tasks according to their priority. In Global EDF scheduler, it's the absolute deadline that determines the execution order of jobs. Thus, the ready list should be sorted in an increasing absolute deadline order. It means that a newly activated job or a preempted one should be inserted in its exact place in the list. Note that a task can also have more than one execution instance (pending jobs), hence a priority level should be reserved for them. Also in case of system overload, the number of ready tasks must be bounded.

For that we consider two data structures for storing the ready jobs: *ReadyList* to store active jobs if their activation

[1]The maximum time the system can operate without causing a clock overrun, it can be calculated by multiplying the resolution with $2^n - 1$.

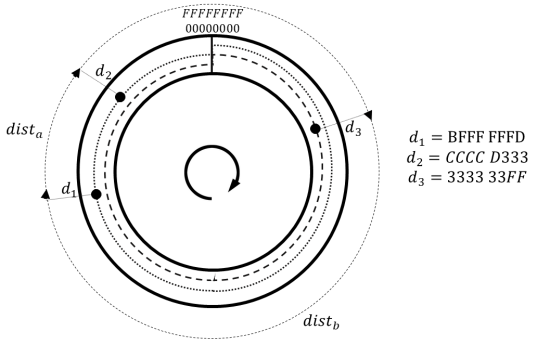

Figure 2: Events evaluated by ICTOH

count[2] is equal to one ; *PendingJobList* is used to store jobs that belong to tasks which have another job running or in the *ReadyList*. Note that each task has its own *PendingJobList*.

The choice of a data structure must essentially depend on the complexity of the task operations using the structure. There are three important operations that are frequently executed in every ready task list: inserting a task once it's activated, searching for the highest priority task and extracting a task for its execution. An empirical study of several data structures was made by Jones [9] comparing the complexity of insertion (*enqueue*) and extraction (*dequeue*) operations. This comparaison shows that a heap data structure can be one of the best alternatives for a dynamic priority policy for its efficiency. The requirements of this data structure in terms of complexity are: $O(1)$ for searching the highest priority task and $O(\log n)$ for inserting or extracting a task.

In our implementation, we are interested in a *min heap* in which deadlines are used as keys (values of the nodes) such that the *heap property* for each node is: if a node A is a parent of a node B, then $deadline(A) \leq deadline(B)$. This structure can be implemented by using a simple array such that: the first element contains the root, the next two elements of the array contain its children, the next four contain their four children, etc (see Fig. 3).

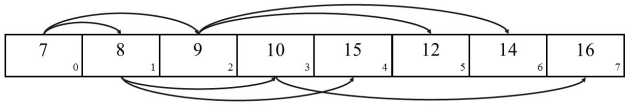

Figure 3: Example of a min heap implemented as an array

For the *PendingJobList* implementation, a simple FIFO is used to store pending jobs according to their activation's date. All the operations using this structure are performed in constant time $O(1)$. The insertion/extraction in this list will be further explained in this paper (cf. §4.3).

4.3 The scheduler

The *Scheduler* component of Trampoline is invoked by calling the function $tpl\_schedule\_from\_running()$ in one of the following situations: 1) task activation ; 2) task completion ; 3) event waiting ; 4) event setting. In our case we

[2]The activation count is used to store the number of jobs activated of the same task.

will rather focus on the first two cases by supposing that tasks are basic (tasks that can never be blocked by an event waiting). In partitioned scheduling, this function does the scheduling for the core concerned by the occured event. In Global scheduling, only one schedule is performed for all cores, and integrating such a scheduling mechanism consists of modification to the scheduler functions in Trampoline. Thus, we define three scheduling functions based on equivalent functions currently implemented in Trampoline :

1. *tpl_edf_activate_task*(): when a new job is activated, its absolute deadline is first calculated by summing the relative deadline with the current time. If the activation count indicates that the task has already a non finished job, the new activated job is stored at the end of the *PendingJobList* of the task. Otherwise this new job is stored in the *ReadyList* according to its absolute deadline. Task activation is a service that requires a rescheduling for which a boolean noted *need_schedule* is set as true at the end of this function.

2. *tpl_edf_terminate_task*(): this function is called only when the OSEK service *TerminateTask* is invoked. First, the *PendingJobList* of the terminated task is verified. If there is a pending job, it is stored in the *ReadyList* according to its absolute deadline. After that the - *need_schedule* boolean is set as true for rescheduling.

3. *tpl_schedule*(): this function is called when the boolean *need_schedule* is set as true. Two tests are performed in this function: 1) to see if there is a free processor while the *ReadyList* is not empty ; 2) to see if the task in the head of the *ReadyList* has a higher priority than a running task. In the first case, the task in the head of the list is extracted to be executed on the free processor. In the second case, the running task is preempted and put back in the *ReadyList*, and the higher priority one is extracted and left for execution.

5. CONCLUSION AND FUTURE WORK

In this paper we presented our integration of Global EDF in Trampoline RTOS and the choices we made for this implementation. Note that developing such scheduler is just a first step in our reseach, since our goal is to verify the implementation of several scheduling policies that were proven optimal in theory. Consequently our work is directed towards:

- **improving the implementation:** Trampoline in its current version implements a global locking mechanism to protect the acces to data structures. This forbids the parallel access to these resources by processors. This limits the global treatment of simultaneous scheduling events. In our work, we also consider studying whether this lock can be replaced by finer locking mechanisms.

- **modeling the scheduler:** in this part, we are interested in modeling all operations and interactions between the RTOS and the scheduling by abstracting the source code of the scheduler. The transition to a global scheduler will introduce new constraints and therefore a different behavior than a partitioned scheduler. For that we must complete Tigori's model with a finer representation of the scheduler's internal properties. Hence, we consider using timed automata or timed Petri nets to establish a model which is very close to the actual source code and allows to verify the implementation.

- **formal verification:** the goal is to elaborate proofs of the scheduler's correctness and the validity of scheduling properties after the implementation. For this, we plan to use model-checking tools like UPPAAL [11] and Roméo [12].

6. REFERENCES

[1] Theodore P Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. *Citeseer*, 2005.

[2] Sudarshan K Dhall and Chung Laung Liu. On a real-time scheduling problem. *Operations research*, 26(1):127–140, 1978.

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

[4] Geoffrey Nelissen, Vandy Berten, Vincent Nélis, Joël Goossens, and Dragomir Milojevic. U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 13–23. IEEE, 2012.

[5] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[6] Jean-Luc Bechennec, Mikael Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline an open source implementation of the osek/vdx rtos specification. In *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*, pages 62–69. IEEE, 2006.

[7] *The Open Group Technical Standard. Base Specifications, Issue 6*. IEEE, 2008.

[8] Alessio Carlini and Giorgio C Buttazzo. An efficient time representation for real-time embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 705–712. ACM, 2003.

[9] Douglas W Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[10] Kabland Tigori, Jean-Luc Béchennec, Sébastien Faucou, and Olivier Roux. Using formal methods for the development of safe application-specific rtos for automotive systems. In *CARS 2015-Critical Automotive applications: Robustness & Safety*, 2015.

[11] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

[12] Didier Lime, Olivier H Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for petri nets with stopwatches. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 54–57. Springer, 2009.

New Approaches to Contention-Sensitive Nested Locking in Real-Time Systems*

Catherine E. Nemitz
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

Nested lock requests in multiprocessor real-time systems can be handled by only a handful of synchronization protocols. These protocols trade off overhead and blocking under varying analysis assumptions. In some systems, a fine-grained contention-sensitive protocol has significantly lower worst-case blocking compared to its non-contention-sensitive counterparts, which yields improved schedulability provided overheads are low enough. In this work, we summarize three key schemes for handling nested requests and briefly discuss existing protocols. We then propose three approaches to reduce the often interdependent overhead and blocking for a new contention-sensitive protocol.

CCS Concepts

•**Computer systems organization** → **Real-time systems;** *Embedded and cyber-physical systems; Embedded software;* •**Software and its engineering** → **Mutual exclusion; Real-time systems software; Synchronization;** *Scheduling; Process synchronization;*

Keywords

multiprocessor locking protocols, nested locks, priority-inversion blocking, real-time locking protocols, contention-sensitive blocking

1. INTRODUCTION

The progression of multicore technologies has allowed increasing numbers of real-time applications to be conceived. To allow these applications to become realities, we must maximize the use of current hardware. For example, the automotive industry is pushing toward autonomous vehicles, which require hardware with low weight, power consumption, and size that can perform complex computations on input data. In particular, sensing data such as images or video streams may be processed and modified by several tasks, requiring resource access control for that shared memory. In addition, images may be processed by some combination of GPUs, which in turn may be considered resources.

*Work supported by NSF grant CNS 1717589. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

For any real-time application, synchronization protocols are required to provide efficient resource-allocation. An efficient synchronization protocol results in schedulability improvements, which allow more effective use of the hardware.

We focus on systems in which nested resource requests are allowed. That is, a job may require multiple resources simultaneously and thus will request the resources in a nested fashion. Our goal is to guarantee contention-sensitive worst-case blocking for all jobs. A job is considered to experience contention-sensitive blocking if the amount of its blocking is dependent only on the number of other requests for the same resource. We will refine this notion in Sec. 2.

Contributions.

After covering background material, we will present three ideas about how we propose to move contention-sensitive locking protocols forward to achieve better schedulability.

2. BACKGROUND

We begin by giving an overview of the systems we are considering and how such systems are analyzed. Then we discuss three broad schemes that are used by various locking protocols to grant access to multiple resources.

2.1 Models

Task model.

We assume the classic sporadic task model. We consider a task system of n tasks denoted $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Tasks are scheduled on m processors using a job-level fixed priority scheduler. We often consider an arbitrary job J_i of task τ_i.

Resource model.

We consider systems with n_r resources. An arbitrary resource is denoted ℓ_a. Unless a locking protocol specifies otherwise, resources may be requested in any order. In this work, we focus on resources that require mutual exclusion.

Request model.

When a job J_i requires access to a resource ℓ_a, it *issues* a *request*, denoted $\mathcal{R}_{i,a}$. If J_i issues only one request, we shorten this to $\mathcal{R}_i$. We say the request is *satisfied* when the job *holds* the resource. While the request is satisfied, it is in a *critical section*. We denote the critical section length of $\mathcal{R}_i$ as L_i and the maximum critical section length of any request L_{max}. Once a request *completes*, the job *releases* the resource. If job holds a resource and then requires another, we say that the requests for these resources are *nested*.

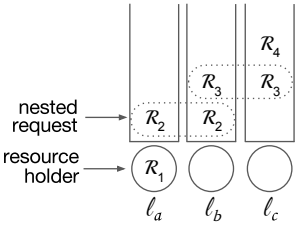

Figure 1: Example illustrating one potential ordering of requests $\mathcal{R}_1$ through $\mathcal{R}_4$. Here, $\mathcal{R}_1$ is satisfied and holds resource ℓ_a. The other requests are enqueued and waiting for access to their required resources.

(Later, we explore a technique that allows requests to be for multiple resources.) We denote the set of all resources J_i will require as D_i.

Nested requests occur in a variety of types of applications, and the depth of nested requests is typically between two and four (that is, between two and four resources are required simultaneously within the innermost critical section) [1, 3].

2.2 Analysis

We consider spin-based locking protocols and analyze such protocols on the basis of *priority-inversion blocking* (pi-blocking), which occurs when a job cannot execute because a lower priority job is holding a resource which the higher priority job requires. We consider the worst-case pi-blocking and take critical section lengths to be constant.

As mentioned above, we focus on contention-sensitive locking protocols. We denote the maximum amount of contention for a resource ℓ_a, that is, the highest possible number of active requests for that resource, as c_a. Note that this value is static for a given system. (This is in contrast to the dynamic concept of contention for ℓ_a, which is the number of active requests for that resource at a particular instance in time.) Given $\mathcal{C}_i = \max_{x \in D_i} c_x$ for a job J_i, we say that a locking protocol is contention-sensitive if the worst-case pi-blocking of any job J_i is bounded by $O(\mathcal{C}_i)$.

The key hindrance to contention-sensitive blocking is the possibility of *transitive blocking*, which occurs when a job experiences blocking because of job that it does not share any resources with. Such a situation is depicted in Fig. 1; request $\mathcal{R}_4$ conflicts only with $\mathcal{R}_3$, and yet neither of these requests are satisfied because of the chain of blocking caused by $\mathcal{R}_1$ and $\mathcal{R}_2$. This figure shows a system which allows resources to be requested together. $\mathcal{R}_1$ is satisfied and holds ℓ_a, depicted by the circle at the head of the queue. $\mathcal{R}_2$ has been enqueued for resources ℓ_a and ℓ_b. Likewise, $\mathcal{R}_3$ has been enqueued for resources ℓ_b and ℓ_c. Finally, $\mathcal{R}_4$ is enqueued and waiting for access to ℓ_c.

2.3 Handling nested requests

The following three schemes handle nested requests to provide mutually exclusive access to resources and prevent deadlock. Each method has its trade-offs, some of which affect analysis components such as the critical section length. To illustrate each approach, we use a short running example.

Example 1. *Consider a job J_1 that requires resource ℓ_a. In addition, consider J_2 that will require access to ℓ_a and then nested access to ℓ_b. Suppose there is also a job J_3 that requires access to ℓ_b and a job J_4 that requires access to ℓ_c.*

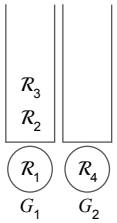

Figure 2: Example illustrating the effect of using static groups.

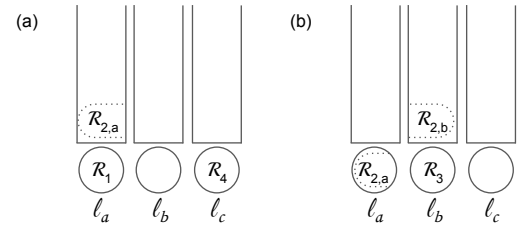

Figure 3: In (a), $\mathcal{R}_1$ holds ℓ_a, and $\mathcal{R}_{2,a}$ is waiting for ℓ_a. In (b), $\mathcal{R}_3$ holds ℓ_b and $\mathcal{R}_{2,a}$ holds ℓ_a, which allowed $\mathcal{R}_{2,b}$ to be issued for ℓ_b. The blocking of $\mathcal{R}_{2,b}$ thus contributes to the critical section length of $\mathcal{R}_{2,a}$, which in turn increases the worst-case blocking of requests for ℓ_a.

Static group locks.

This method requires static groups of resources to be formed by analyzing which resources are accessed in a nested fashion by some job. A job requiring any resource in the group must acquire the entire set. This approach allows a mutex to control access to any shared resource; each job will require only one group, so deadlock is impossible. The use of a mutex yields low overheads and inherently provides contention-sensitive blocking. However, it is important to distinguish that this notion of contention is relative to the created group of resources that a job requests. The static groups may be quite pessimistic, causing a job to contend with jobs that do not share resources but do share a group.

Example 1 (cont'd). *With the four jobs above, static groups $G_1 = \{\ell_a, \ell_b\}$ and $G_2 = \{\ell_c\}$ could be formed. Then jobs J_1, J_2, and J_3 will all issues requests for G_1, and job J_4 will issue a request for G_2, as shown in Fig. 2. Note that J_1 and J_3 now are considered to share a resource for the purposes of determining blocking, though they do not actually require access to the same resource. This is a small example of the increased pessimism that static group locks cause.*

Resource ordering.

In contrast to static group locks, resource ordering allows *fine-grained* locking; each job only acquires the resources it needs. In this approach, a total order on all resources is defined prior to system startup. Any nested requests must acquire resources in that order. This prevents deadlock and easily allows for a low overhead protocol within this scheme. However, this approach can easily inflate the critical section lengths beyond the given L_i, as illustrated by the following example. In fact, this inflation can be more than $m \cdot L_{max}$.

Example 1 (cont'd). *In this scenario, requests $\mathcal{R}_1$, $\mathcal{R}_{2,a}$, and $\mathcal{R}_4$ were released before $\mathcal{R}_3$. As shown in Fig. 3(a), $\mathcal{R}_1$ was immediately satisfied and holds ℓ_a. With the resource ordering imposed, J_2 must first issue a request for ℓ_a*

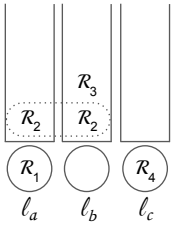

Figure 4: With DGLs, job J_2 issues a single request $\mathcal{R}_2$ for all resources it requires.

and then issue a separate request for ℓ_b, denoted $\mathcal{R}_{2,a}$ and $\mathcal{R}_{2,b}$, respectively. In Fig. 3(a), $\mathcal{R}_{2,a}$ is waiting for access to ℓ_a. $\mathcal{R}_4$ is satisfied and holds ℓ_c.

In Fig. 3(b), $\mathcal{R}_4$ has completed and $\mathcal{R}_3$ has been issued for ℓ_b and is satisfied. Later, $\mathcal{R}_1$ completed, and $\mathcal{R}_{2,a}$ became satisfied. J_2 then issued $\mathcal{R}_{2,b}$ and must wait for $\mathcal{R}_3$ to complete. It holds ℓ_a, so the time $\mathcal{R}_{2,b}$ blocks inflates the critical section length of $\mathcal{R}_{2,a}$.

Observe that resource ordering can cause huge amounts of blocking. A job requiring multiple resources may experience the following. Just before its first resource becomes available, requests may enqueue for its second resource. While it holds the first resource, it could experience the worst-case blocking for its second resource. This inflates the critical section, causing higher blocking than the expected L_i for any later request for that first resource. This build-up of blocking can be repeated for each nested resource access the job requires.

Dynamic group locks.

A third method for handling nested requests is by using *dynamic group locks* (DGLs). In this scheme, a job requiring nested resources issues a single request for all resources that it requires. This lengthens all inner critical section lengths to the length of the outermost access. Note that if a job conditionally acquires ℓ_a or ℓ_b but not both, under DGLs, it must request both resources. While holding both resources decreases potential runtime parallelism, it does not have an effect on the overall blocking; as discussed later as it pertains to static contention, we must consider the worst-case contention for each resource, and this job would be counted toward the contention of both resources regardless.

Several directions of work have been explored using the DGL scheme, as discussed in Sec. 3. One approach has moderate overheads and $O(m)$ blocking (that is, blocking bounded by the number of processors). Another approach has similar overheads and contention-sensitive blocking given certain analysis assumptions.

EXAMPLE 1 (CONT'D). *Returning to our four jobs, under DGLs, J_2 issues a single request for both ℓ_a and ℓ_b. (In order to prevent deadlock, protocols must ensure requests for multiple resources enqueue atomically into all required resource queues.) Fig. 4 shows one way in which the requests could enqueue. $\mathcal{R}_1$ is satisfied and holds ℓ_a. Thus, $\mathcal{R}_2$ is blocked. As depicted in Fig. 4, $\mathcal{R}_3$ was issued after $\mathcal{R}_2$ and must wait for access to ℓ_b. Regardless of when it is issued, $\mathcal{R}_4$ is satisfied immediately, as no other requests require ℓ_c.*

3. RELATED WORK

Standard mutex implementations, such as ticket locks and MCS locks function well with static groups locks and are inherently contention-sensitive [8].

Protocols that support fine-grained lock nesting by using resource ordering include the Multiprocessor Bandwidth Inheritance Protocol (M-BWI) [5], MrsP [4], and nested FIFO locks [2], the last of which has corresponding analysis that tractably bounds blocking.

The only protocols to use DGLs are those in the Real-time Nested Locking Protocol (RNLP) family [11, 10]. Two RNLP variants yield contention-sensitive blocking. The fast RW-RNLP provides contention-sensitive resource access only to read requests and non-nested write requests [9]. Nested write requests under the fast RW-RNLP are not contention-sensitive. Finally, the C-RNLP yields contention-sensitive blocking with the assumption that critical section lengths are the same for all resources [6].

4. NEW APPROACHES

We present three approaches to use with DGLs that we believe will be important in improving upon existing approaches toward nested lock requests. In particular, our goals are low overheads and contention-sensitive blocking for all requests, which will lead to better schedulability results.

4.1 Mutex usage

When a lock implementation requires maintenance of significant lock state, the simplest approach is to protect this state with a mutex that prevents concurrent lock calls from modifying the lock state simultaneously. This is the approach taken by the C-RNLP. While this is safe, it increases overheads by causing all requests to conflict on the lock-state mutex.

Therefore, our first approach to a new contention-sensitive locking protocol is to eliminate or reduce the usage of a lock-state mutex. Some lock structures allow this naturally or with only a slight addition of state-maintenance operations and thus overhead. For example, enqueing on multiple queues in a way that is seen as atomic simply requires that two requests for the same resources enqueue in the same order relative to each other for each resource. This is a condition that can be checked and preserved without requiring a mutex. Alternatively, even using a different mutex for each resource queue would reduce the amount of blocking that contributes to overhead, as only requests for the same resource, which already contend, would share a given mutex.

4.2 Static contention

Previous approaches to providing contention-sensitive resource access have focused on doing so with a dynamic view of contention; that is, a new request's worst-case blocking should be upper bounded by the number of active requests with overlapping resource requirements. However, this dynamic view of contention cannot be used in schedulability analysis. We must instead use the static measure of contention (the upper bound of the possible dynamic contention) in our analysis.

In light of this insight about the use of dynamic and static contention, we aim to explore the use of static contention in constructing a new contention-sensitive protocol.

A protocol designed around static contention may have less overhead; decisions regarding enqueuing for resources could be based on the static contention instead of computing the number of requests ahead of the current request. We are interested in exploring the trade-offs in such an ap-

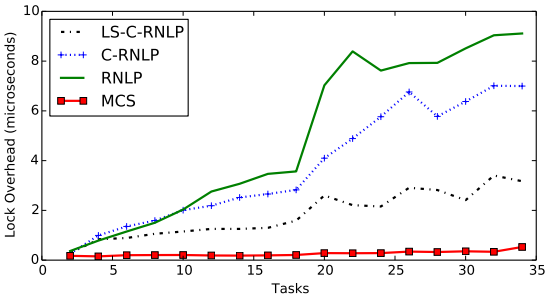

Figure 5: Lock overhead as a function of task count n for $n_r = 64$ and each job requesting four random resources from that set.

proach, which we expect would increase both schedulability and average blocking times (which could negatively impact non-real-time workloads running on the same platform as the real-time workload).

An alternate way to lower overheads could be to mix the ideas of DGLs and lock ordering. Jobs could issue requests under the DGL scheme, with all resources requested simultaneously. The lock state could then be updated with an ordered approach, in which resources are ordered by decreasing static contention. When enqueuing for these resources, requests could be required to wait until some threshold value is met before enqueuing in the subsequent queue. To clarify, a request might enqueue for ℓ_a, wait for $c_a - c_b$ time units, and then enqueue for ℓ_b, with the goal of becoming satisfied for all its resources at the same time.

4.3 Lock server

As with many ideas, this approach comes from a solution to a different problem. When some legacy applications are transfered to a multiprocessor context, a fundamental component that can slow its execution is the presence of lock requests. The idea of *remote core locking* improved performance; one core was dedicated to processing lock requests for one or more locks. This allowed the lock state and the memory locations protected by the lock to remain cache-hot. Requests were issued to this remote core by writing the lock identifier and the address of the critical section that needed to be executed to its shared cache space [7].

We propose a similar solution that we call a *lock server*. In contrast to remote core locking, a lock server maintains the lock state and executes the logic of lock and unlock calls but does not execute any of the critical section code on behalf of the request. This approach was motivated by previous work [6], in which we observed overhead trends that imply that the lock state bounces between different caches. In particular, notice the overheads presented in Fig. 5 of the RNLP and the C-RNLP. The tasks systems that generated these overheads (described in more detail below) was run on a 36 core machine with two sockets. Each task was pinned to a core, and while there were at most 18 tasks, only a single socket was used. However, once the second socket (with a separate cache) was in use, overheads drastically increased.

To do some preliminary testing of the hypothesis that a lock server would reduce overheads, we implemented the C-RNLP as a simple lock server (denoted LS-C-RNLP). Each request was issued to the lock server, which returned a location in memory on which to spin. The job then spun on its core until the value in that location in memory was set by the lock server to indicate that its request was satisfied.

We evaluated the LS-C-RNLP against the original C-RNLP, the RNLP, and the MCS on a dual-socket 18-cores-per-socket Intel Xeon E5-2699 platform. As mentioned above, each task was pinned to a core (using only a single socket when possible). These tasks repeatedly performed lock and unlock calls with a negligible critical section length in order to try to cause the worst-case overheads. Each task issued 1000 requests for a randomly chosen set of four resources of the available $n_r = 64$ resources. We report the 99th percentile of these overheads for varying numbers of tasks in the system in Fig. 5.

For the new contention-sensitive protocol we develop, we will test its overheads and the resulting schedulability of implementing it both with and without a lock server.

5. CONCLUSION

We explored three approaches to attaining a more universally applicable contention-sensitive protocol with increased schedulability results. In future work, we will explore these ideas. In particular, we are attempting to build the lock state data structures around the statically defined contention per resource. We are considering using DGLs for request issuance, but within the lock logic, employing a lock-enqueuing ordering based on decreasing static contention as a means of limiting the length of any transitive blocking chains.

6. ACKNOWLEDGMENTS

The author would like to thank Jim Anderson and Tanya Amert for their discussions and helpful feedback.

7. REFERENCES

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI 1998*.

[2] A. Biondi, B. Brandenburg, and A. Wieder. A blocking bound for nested FIFO spin locks. In *RTSS 2016*.

[3] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT 2007*.

[4] A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS 2013*.

[5] D. Faggioli, G. Lipari, and T. Cucinotta. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6), 2012.

[6] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS 2015*.

[7] J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC'12*.

[8] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization of shared-memory multiprocessors. *Transactions on Computer Systems*, 9(1), 1991.

[9] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. In *RTNS 2017*.

[10] B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *IPDPS 2014*.

[11] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS 2012*.

Ordering strict partial orders to model behavioural refinement

Mathieu Montin
Université de Toulouse ; INP ; IRIT
2 rue Camichel, BP 7122
31071 Toulouse Cedex 7, France
mathieu.montin@enseeiht.fr

Marc Pantel
Université de Toulouse ; INP ; IRIT
2 rue Camichel, BP 7122
31071 Toulouse Cedex 7, France
marc.pantel@enseeiht.fr

ABSTRACT

Behavioural refinement plays a key role in the development of correct–by–construction complex systems such as real time distributed systems. Indeed, behavioural models coupled with formal methods allow to assess the correctness of system models with respect to system requirements. In that purpose, behavioural refinements allow preserving the correctness of the models during the development phases, from the early specification to the final embedded system. Refinement is usually handled in an operational matter such that each level of abstraction is derived from notions coming from the closest higher level of abstraction. This vision however is unsuitable when coupled with denotational semantics where the solutions are not built but rather validated by the semantics. Our work targets the definition of a refinement relation compatible with this kind of semantics. This relation is integrated to CCSL where refinement is not a native construct of the language and whose semantics is given in a denotational manner.

CCS Concepts

•**Software and its engineering** → **Formal software verification;** •**Computer systems organization** → *Embedded software;* •**Theory of computation** → *Type theory;*

Keywords

Refinement; Behavioural models; Agda; CCSL

1. INTRODUCTION

Software is now ubiquitous and involved in complex interactions between the human user and the physical world in so-called cyber-physical systems (CPS). To handle the growing complexity of these systems, separation of concerns is mandatory. Two different kinds of separation are usually identified throughout their development: the horizontal and vertical separation. The first one corresponds to the various concerns in the system architecture which might be described in different domain specific modelling languages (DSMLs). The second one corresponds to the different steps in a development leading from the requirements to the implementation through the use of refinements. The horizontal separation is usually handled through the abstraction of the different parts of the system in a common behavioural language. However, most of these languages allow the expression of constraints between the different preoccupations of the system but lack the required expressiveness to handle vertical separation. In this paper, we propose a formal definition of the relation of refinement in a denotational context. It relies on an order between the strict partial orders that are used to bind together the different instants on which events occur. This work has been conducted using the Agda proof assistant and associated to CCSL denotational semantics, although no knowledge regarding both languages will be needed here as details cannot be given due to space limits but the whole development is available on the first author's web page.

2. A REFINEMENT EXAMPLE

The transition system depicted in Figure 1 is an example of a simple system, which can be alternatively enabled and disabled. While it is active, an action can be executed any number of times. We focus on event traces and event refinement and not state traces and refinement like [3] as we model time using instantaneous event in a synchronous manner.

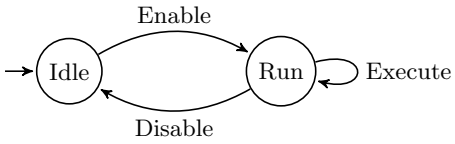

Figure 1: A simple system

By picturing our example with a transition system, we implicitly provided an operational semantics. Indeed, the transition system can be seen as a machine which builds a correct execution for the system. A denotational semantic however, does not give any way of creating such traces and instead provides predicates to assess the correctness of a given trace. This distinction is essential since most refinement strategies rely on operational semantics while we aim at handling systems through their denotational semantics. A possible trace for our example is depicted on Figure 2. t_{en}, t_{di} and t_{ex} respectively represent the occurrences of the "Enable", "Disable" and "Execute" transitions. This trace could have been generated from our transition system and would be validated by any denotational semantics describing our example.

This trace starts with the birth of the system and possibly goes on indefinitely, which makes this representation partial. In addition, this design places each event on the same timeline, thus ignoring horizontal separation. In order to make it visible, we represent each event on a specific timeline on

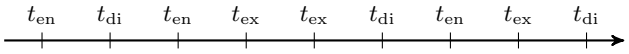

Figure 2: A trace on a single timeline

Figure 3. The instants on each timeline are totally ordered and those in the same vertical dotted lines are coincident. These notions will be elaborated when introducing the strict partial orders.

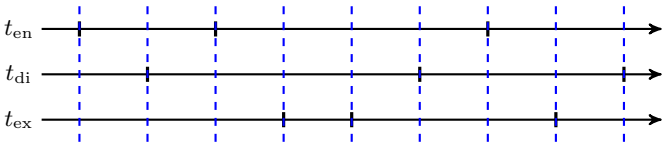

Figure 3: One timeline per event

The action executed by this system can be specified in various ways. In this paper, we imagine that it is connected to a light through the use of a memory containing a variable m. This variable will be assigned the values 1 or 0, and the light will be turned on and off accordingly. When the system is enabled (t_{en} transition), the light remains down until a button is pressed (t_{ex} transition) shuts it down. Pressing the same button will alternatively turn it off and on. Disabling the system (t_{di} transition) turns it off, as depicted on Figure 4.

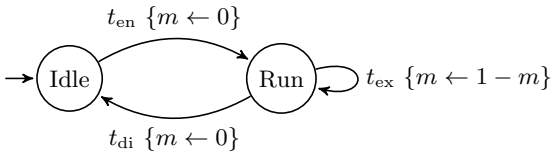

Figure 4: The system pilots a light

By specifying the system behaviour, we defined events to add to its traces. t_{m_0} and t_{m_1} respectively correspond to the variable m being assigned 0 and 1. These additions belong to horizontal separation since we added a new part to our system (the module linked to the light). One of these possible traces is depicted in Figure 5. As we add new events, refinement cannot be defined as a simple inclusion of traces like [5].

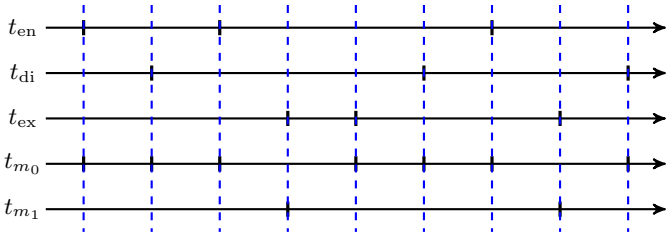

Figure 5: The new trace of the system

Some events are occurring simultaneously, for instance t_{en} always occurs on an instant coincident to an occurrence of t_{m_0}. Such relations between events can be defined in CCSL (a simple case of sub-clocking here), which has been handled in a previous work on the mechanization of this language.

It is important to note that when describing this system, we implicitly took a certain point of view regarding its definition. We deliberately ignored some low level concerns regarding the way such a memory is handled. This is a matter of vertical separation. The next sections of this paper will focus on a more concrete level of abstraction. But first we need to introduce some standard notions regarding time handling in asynchronous languages.

3. REPRESENTATION OF TIME

When considering executable languages, we observe different events which occur on given instants of time. Although the common vision of time is a straight line inducing a total order between each existing instant, asynchronous systems introduce uncertainties that weaken this order. Two instants are indeed not necessarily comparable, which leads to the use of partial orders to represent the existing links between them. Thus, each pair of instants is either:

- comparable, through a precedence relation $\prec$
- equivalent, through a coincidence relation $\approx$
- unrelated (neither comparable nor equivalent)

Some properties are required for these relations to form a strict partial order:

- $\approx$ is an equivalence relation
- $\prec$ is irreflexive regarding $\approx$
- $\prec$ is transitive
- $\prec$ respects the classes induced by $\approx$

4. BEHAVIOURAL REFINEMENT

4.1 Goal

Whenever a certain event occurs on a given instant, its occurrence is considered immediate and punctual in the timeline. However in some cases, such an event can be decomposed in smaller events which contradicts this property. In our example, the "Enable" event can be viewed as a sequence of sub–events, such as powering up the system, retrieving the address of m, computing the value of 0 (here there is no actual computation since 0 is an atomic value, but there could be in the case of a more complicated arithmetical expression) and storing this value at the right address. These events, except for the first one, are used to handle the computation and the storing of a value in a memory. Taking into account these events require to view the system at a more concrete level, in which case its representation as a transition system is depicted in Figure 6.

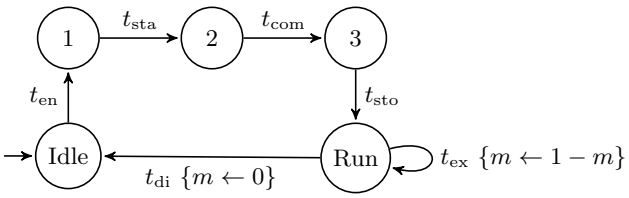

Figure 6: The refined system

The "Enable" transition has been refined in several transitions. t_{en} represents the powering of the system, t_{sta} the

stacking of the address of m, t_{com} the computing of the value of the expression 0 and t_{sto} the storing of the computed value at the stacked address.

Since both points of view we discussed are valid representations of our system, it should be possible to describe them in any concurrent language, without losing the link that binds them. This leads to the main goal of our proposal to model behavioural refinement, which is to describe a system at different levels of observation without losing the link between these levels. Thus, an instant at a certain level could be refined by several instants at a lower level, just like the "Enable" event was split into several different events.

Addressing this issue would allow system developers to focus on their specific view of the system rather than a common view shared among all of them. By representing it from the right angle, they could grasp their constraints even better without bothering about more concrete details. The system could then be solved at different levels with the guarantee that none of them will be compromising the others. Furthermore, this notion of refinement could be used to make explicit and prove simulations and bisimulations (or mostly weak bisimulations) between systems. In this case, the two specifications would not be different levels of observation of a system, but different ways of specifying its behaviour.

4.2 Different levels of refinement

In our example, the higher level of observation is represented on Figure 7 while the lower level is represented on Figure 8. In both these timelines, events not refined are omitted, for the sake of clarity. They do not influence the reasoning we are conducting, thus their omission is acceptable. The different instants have been annotated with natural numbers in order to manipulate them more easily. From the higher point of view, all the instants on which the subevents occur are equivalent to each other and to the containing event. Their underlying order has no impact on the trace of the system at this level.

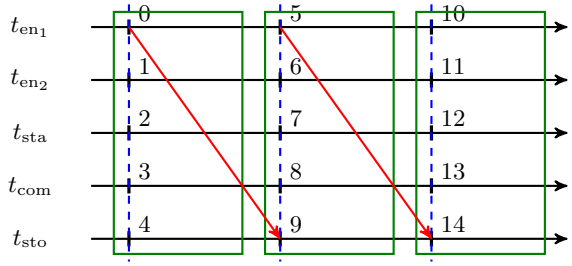

Figure 7: The annotated higher level of observation

For the lower level of observation, the different instants are ordered in such a way that they respect the specification in Figure 6. The vertical dashed lines represent the equivalence classes induced by the thinner strict partial order, while the rectangles represent the ones induced by the refined strict partial order.

The representation in Figure 7 allows us to to assess the coincidence and the precedence relation that bind its different instants, as subsets of $\mathbb{N} \times \mathbb{N}$. Since both these relations must be transitive, the coincidence must be symmetrical and they must form a strict partial order, we will omit the related

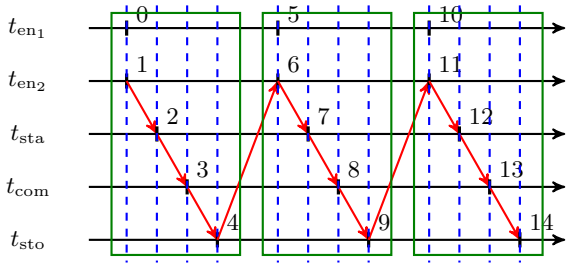

Figure 8: The annotated lower level of observation

elements which can be deduced from these properties.

Coincidence			Precedence
(0 , 1)	(0 , 2)	(0 , 3)	(0 , 5)
(0 , 4)	(5 , 6)	(5 , 7)	
(5 , 8)	(5 , 9)	(10 , 11)	(5 , 10)
(10 , 12)	(10 , 13)	(10 , 14)	

These traces are potentially infinite, thus we only give the visible subset of each relation. However, we can define them mathematically for any natural number in order to handle their infinite number. This is done by relying on the Euclidean decomposition by 5:

$$\forall (a, a') \in \mathbb{N}^2, \exists!\ (q, r, q', r') \in \mathbb{N}^4\ :\\ a = 5q + r \wedge r < 5 \wedge a' = 5q' + r' \wedge r' < 5$$

These relations are defined as follows:

$$\forall (a, a') \in \mathbb{N}^2, a \approx_2 a' \overset{d}{\iff} q = q'\\ \forall (a, a') \in \mathbb{N}^2, a <_2 a' \overset{d}{\iff} q < q'$$

The same work can be achieved for the lower level of observation, which is displayed on Figure 8. The relations extracted from Figure 8 are depicted in the table below. As previously explained, only the relevant couples are mentioned.

Coincidence	Precedence		
(0 , 1)	(1 , 2)	(2 , 3)	(3 , 4)
(5 , 6)	(4 , 5)	(6 , 7)	(7 , 8)
(10 , 11)	(8 , 9)	(9 , 10)	(11 , 12)
...	(12 , 13)	(13 , 14)	...

Here are the relations at the concrete level:

$$\forall (a, a') \in \mathbb{N}^2, a \approx_1 a' \overset{d}{\iff}\\ (q_1 = q_2) \wedge ((r_1, r_2) \in [0, 1]^2 \vee (r_1 = r_2 \wedge r_1 \notin [0, 1]))\\ \forall (a, a') \in \mathbb{N}^2, a <_1 a' \overset{d}{\iff}\\ (q_1 < q_2) \vee ((q_1 = q_2) \wedge (r_1 < r_2) \wedge (r_2 \neq 1))$$

Our example exhibits two different couples of relations, which should be in a situation of refinement.

4.3 Our proposal

As an attempt to formalize this approach, we propose to connect different levels of abstraction through the strict partial order they carry. We define the following relation $<_r$ to ensure the underlying relations fulfil the right conditions to maintain the integrity of the different representations regarding the semantics of refinement:

$$
\begin{array}{l}
\forall I \in \Omega, \forall (<_c, <_a, \approx_c, \approx_a) \in (I \times I)^4 : \\
(<_c, \approx_c) <_r (<_a, \approx_a) \overset{d}{\Longleftrightarrow} \forall (i_1, i_2) \in I : \\
\qquad i_1 <_c i_2 \Rightarrow i_1 <_a i_2 \vee i_1 \approx_a i_2 \qquad (1) \\
\qquad \wedge \quad i_1 <_a i_2 \Rightarrow i_1 <_c i_2 \qquad (2) \\
\qquad \wedge \quad i_1 \approx_c i_2 \Rightarrow i_1 \approx_a i_2 \qquad (3) \\
\qquad \wedge \quad i_1 \approx_a i_2 \Rightarrow i_1 \approx_c i_2 \vee i_1 <_c i_2 \vee i_2 <_c i_1 \qquad (4)
\end{array}
$$

$<_r$ is defined as a relation between pairs of relations ($<_c$,$\approx_c$) and ($<_a$, $\approx_a$) that represent the strict precedence and the equivalence composing the strict partial orders bound to both levels of abstraction. In this definition, the level annotated by the index c is the more concrete level and a is the more abstract. We state what it means for a pair of relations to refine another pair of relation. These relations are defined on the set I of instants. We can only compare pairs of relations that are bounded to the same set. This definition is composed of four predicates, each of which indicate how one of the four relations is translated into the other level of observation:

1. If a strictly precedes b in the lower level, then it can either be equivalent to it in the higher level or still precede it.

2. However, if a strictly precedes b in the higher level, then it can only still precede it in the lower level. This direction doesn't allow any loss of information.

3. On the contrary, if a is equivalent to b in the lower level, it can only stay equivalent in the higher level.

4. If a is equivalent to b in the higher level then we only assure that these two instants are still related in the lower level. We can gain information this way.

This definition can be extended to strict partial orders: A strict partial order refines another when their underlying relations are in a relation of refinement.

5. RELATED WORKS

Our work takes place in GEMOC that mixes both horizontal and vertical separation of concerns. Indeed, GEMOC allows to define the various DSMLs used to model the various parts in a CPS in the various phases of the development. GEMOC relies on the Clock Constraint Specific Language (CCSL) in order to model both the MoC for the various DSML [6, 8, 12] and the coordination between DSML using the BEhavioural COOrdination Language (BeCooL) [11].

Our approach is motivated by the lack (to our knowledge) of formal definition of behavioural refinement in a context of denotational semantics. Refinement has already been studied widely through operational semantics [15, 13], and is the core concept advocated in developing correct–by–construction systems with the B [1] and Event-B [2] methods.

Our proposal provides a mechanized relation of refinement in Agda and aims at being coupled to a previous work on a mechanisation of the semantics of CCSL in the same proof assistant, based on a paper denotational semantics [7]. Thus, this approach could be reused for any other concurrent languages. Formal mechanization of temporal languages has already been done using other formal methods, for example [10] uses Higher Order Logic in Isabelle/HOL; [9] and [14] use the Calculus of Inductive Constructions in Coq, see [4].

6. CONCLUSION

This paper presented a mathematical relation over strict partial order whose goal is to model behavioural refinement in a denotational manner. Each level of abstraction is associated to a specific strict partial order while our relation binds them together. This definition has been mechanized in the Agda proof assistant, which allowed us to prove several properties about it as well as connect it to the mechanization of CCSL we made in a previous work. The bridge between these contributions has allowed us to prove the preservation of several CCSL operators through our relation of refinement. This work will lead to an extension of CCSL with a refinement operator which will allow both comparing language semantics in GEMOC and conducting correct–by–construction developments more easily with CCSL. The whole development, including the parts about CCSL, is available online on the first author's web page.

7. REFERENCES

[1] J. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[2] J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[3] R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR '94, Concurrency Theory, 5th Intl. Conf., Uppsala, Sweden, Aug. 22-25, Proc.*, pages 367–384, 1994.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

[5] A. Cavalcanti and M. Gaudel. A note on traces refinement and the *conf* relation in the unifying theories of programming. In *Unifying Theories of Programming, 2^{nd} Intl. Symp., UTP 2008, Dublin, Ireland, Sep. 8-10, Revised Selected Papers*, pages 42–61, 2008.

[6] B. Combemale, J. DeAntoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, and R. B. France. Reifying concurrency for executable metamodeling. In *Software Language Engineering - 6th Intl. Conf., SLE 2013, Indianapolis, IN, USA, Oct. 26-28. Proc.*, 2013.

[7] J. Deantoni, C. André, and R. Gascon. CCSL denotational semantics. Research Report RR-8628, 2014.

[8] J. DeAntoni, P. I. Diallo, C. Teodorov, J. Champeau, and B. Combemale. Towards a meta-language for the concurrency concern in dsls. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, 2015.

[9] M. Garnacho, J. Bodeveix, and M. Filali-Amine. A mechanized semantic framework for real-time systems. In *Formal Modeling and Analysis of Timed Systems - 11th Intl. Conf., FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proc.*, 2013.

[10] R. Hale, R. Cardell-Oliver, and J. Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1/2), 1993.

[11] M. E. V. Larsen, J. DeAntoni, B. Combemale, and F. Mallet. A behavioral coordination operator language (bcool). In *18th ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, Sep. 30 - Oct. 2.*, 2015.

[12] F. Latombe, X. Crégut, B. Combemale, J. DeAntoni, and M. Pantel. Weaving concurrency in executable domain-specific modeling languages. In *Proc. of the ACM SIGPLAN Intl. Conf. on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, Oct. 25-27*, 2015.

[13] D. Murphy and D. Pitt. *Real-timed concurrent refineable behaviours*, pages 529–545. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[14] C. Paulin-Mohring. Modelisation of timed automata in coq. In *Theoretical Aspects of Computer Software, 4th Intl. Symp., TACS 2001, Sendai, Japan, October 29-31, 2001, Proc.*, 2001.

[15] G. Ramanathan. Refinement of events in the development of real-time distributed systems. *Theoretical Computer Science*, 133(2):341 – 359, 1994.

A hypervisor schedulability analysis for safety and security critical applications scheduled in arbitrary patterns of slots

Tristan Fautrel
Université Paris-Est
LIGM, UMR CNRS 8049
tristan.fautrel@u-pem.fr

Laurent George
ESIEE Paris
LIGM, UMR CNRS 8049
laurent.george@esiee.fr

Frédéric Fauberteau
Léonard de Vinci Pôle
Universitaire, Research Center
frederic.fauberteau@devinci.fr

ABSTRACT

This paper focuses on the problem of scheduling several applications on top of a hypervisor. The hypervisor is in charge of satisfying safety and security constraints enforced by space and temporal isolation. An application is run by one processor of a multiprocessor platform in dedicated slots composing a pattern. One processor can run several applications each assigned to a dedicated pattern of slots. A hypervisor distributes the time resource among several Virtual Machines (VMs) on a multiprocessor architecture. Each VM embeds an application consisting of a set of sporadic real-time tasks. These tasks are scheduled according to a preemptive Fixed-Task-Priority (FTP) policy in the slots assigned to their application. A task can be executed by one or several slots and a slot can execute several tasks. First, we derive an exact schedulability condition using response time analysis for sporadic tasks scheduled in a periodic arbitrary pattern of slots. Then, we investigate several pattern constructions derived from the scheduling of tasks with classical schedulings like Round Robin (RR), Weighted Fair Queueing (WFQ), Rate Monotonic (RM) and Earliest Deadline First (EDF). Finally, we compare by simulation the success ratios of several constructions of slot patterns.

1. INTRODUCTION

Multicore Commercial Off-The-Shelf (COTS) platforms are now considered as interesting powerful platforms for executing safety critical applications like those in the automotive industry. One challenge arises when several applications of different safety and security constraints are executed on the same platform. Satisfying both constraints can be obtained by enforcing space and temporal isolation between applications *e.g.* by using processor affinity possibilities, cache partitioning and cluster scheduling. The approach currently considered in the automotive industry to satisfy space and temporal isolation is to execute applications on top of a hypervisor in charge of managing platform resources so as to preserve this space and temporal isolation. This put a high level of constraints on the hypervisor that must be certified for a given set of safety and security constraints.

Hierarchical scheduling and compositional approaches [7] use the concept of servers in charge of executing components developed in isolation.

In this paper, we investigate one solution adopted in the industry (in particular for automotive application) based on a certified hypervisor. We suppose that each application is assigned to a specific processor of a multicore platform, using processor affinity and cache partitioning controlled by the hypervisor. A processor can execute several applications. Applications are modeled by sporadic tasks. The hypervisor defines for each processor a scheduling table of slots dedicated to the execution of all applications. A slot can execute several tasks depending on their Worst Case Execution Time (WCET) and a task can be executed on several consecutive slots assigned to its application.

In this paper we study the schedulability problem of sporadic tasks, associated to one application, when tasks are scheduled by one processor in a pattern of slots. Several patterns of slots, one per application, can be associated to one processor. Therefore each application may be studied independently once a pattern of slots for it is found.

The contribution of this paper is first to find a schedulability analysis for the local tasks of each application (scheduled by FTP) by taking into account the interference of the global scheduler which selects an application to execute on one processor at time t. The analysis and the model are similar to [4] except that we consider FTP instead of EDF as local scheduling policy. We then compare different approaches based on classical algorithms to find the patterns and then the global scheduler. Our approach differs from the Generalized Multi Frame approach [1] by the fact that slots assigned to one application can execute several tasks and not only one.

2. MODEL AND NOTATIONS

Our model is a two-level hierarchical scheduling scheme. In this model, we define patterns of time slots which corresponds to servers in the model described in [4]. Each pattern corresponds to a VM and is comprised of a task set that can be seen as an application. A table is a structure that contains all the patterns of slots and that corresponds to the entire system. By definition, a VM is scheduled in consecutive time slots defining a pattern of slots. Each slot can be used by one or several sporadic tasks.

Our system under study consists in one single processor of a multiprocessor platform managed by a hypervisor for executing a set of applications. On this system we schedule a task set τ of preemptive tasks, each task is assigned to one application. The i^{th} sporadic task is denoted τ_i and is characterized by the tuple (C_i, D_i, T_i) where C_i is its Worst Case Execution Time (WCET), D_i its arbitrary deadline and T_i its minimal inter-arrival time. The utilization of a task τ_i is defined by $\frac{C_i}{T_i}$ and is denoted U_i. These tasks are executed on a platform built on a hypervisor that schedules

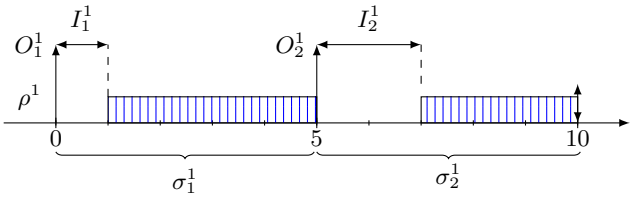

Figure 1: Pattern representing the execution of a VM and comprised of 2 time slots.

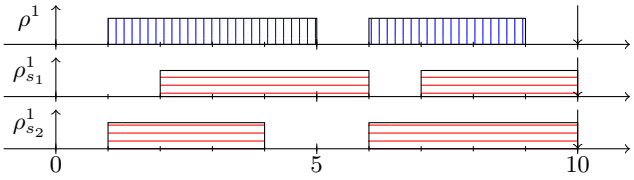

Figure 2: The same pattern shifted to represent the segments

several VMs. A given task is assigned to only one VM.

We represent the execution parameters of a VM by a periodic pattern of time slots of execution. Our system consists in a set of VMs that is represented by a table ρ including all the patterns. The pattern of the s^{th} VM associated to one application is denoted ρ^s. The tasks assigned to this application and then to the pattern ρ^s are denoted $\tau(\rho^s)$.

Each pattern is composed of a list of segments. The k^{th} segment of ρ^s is denoted σ_k^s and is composed of an idle slot section followed by a non empty time slot dedicated the the execution of tasks (*e.g.* filled rectangle from 1 to 5 in Figure 1). A segment σ_k^s is defined by the tuple (O_k^s, I_k^s) where O_k^s is the offset from the activation time of ρ^s and I_k^s the idle slot between O_k^s and the start time of time slot of σ_k^s.

The pattern ρ^s is characterized by the tuple $(\mathcal{C}^s, \mathcal{T}^s, \vec{\sigma}^s)$ where $\mathcal{C}^S$ is its execution time, $\mathcal{T}^s$ its period and where $\vec{\sigma}^s$ represents the set of segments that composes ρ^s. The parameters of the pattern are defined by the tasks that are associated to it. We only know its minimal utilization which must be greater than the sum of the utilization of the tasks.

In Figure 1, we show an example of a pattern consisting of two segments. The first segment σ_1^1 has an offset O_1^1 equal to 0 and an idle slot I_1^1 equals to 1. The second one, σ_2^1 has an offset O_2^1 equal to 5 and an idle slot I_2^1 equals to 2.

By using this representation of ρ^1, a pattern may contain segments without time slot. This is represented by ρ^1 in Figure 2. If the time slots of this pattern are represented, the third segment has for parameters an offset O_3^1 equal to 9 and an idle slot I_3^1 equal to 1 which leads to an empty time slot. If any of this case happen the pattern will be shifted to start at an idle slot and to finish at a time slot. Throughout this paper, every pattern may be shifted to match this representation if it ends by an idle time slot after its creation. When a pattern is studied, the others do not have to be taken into account. This is why a pattern may be shifted without any impact on the analysis of others.

In Figure 2 a pattern and its two possible shifts are represented. The pattern ρ^1 cannot be represented using our model since its representation does not end by a non empty time slot (ρ^1 ends by an idle slot). We can apply a right-shift of 1 (respectively of 5) time units to obtain the shifted representation $\rho_{s_1}^1$ (respectively $\rho_{s_2}^1$).

3. PATTERN SCHEDULABILITY ANALYSIS

In this section, we present an exact schedulability analysis for a sporadic task set scheduled in an arbitrary pattern of slots, by a FTP algorithm on a platform composed of a set of VMs. For the sake of space we will not recall the classical Worst Case Response Time (WCRT) presented in [5] and only the WCRT with multiple segments will be explained (which is more general than with only one segment).

3.1 Pattern Critical instant

In order to provide an exact schedulability analysis, the scenario that exhibits the worst case response time has to be identified. As presented in [5], the worst case response time is obtained when the analysis is performed by considering the minimum inter-arrival time for all tasks.

In 1973, Liu and Layland [6] proved that by using a FTP algorithm a critical instant ("a critical instant for a task is defined to be an instant at which a request for that task will have the largest response time") is when all the tasks are synchronous (all released at the same time). In our model the release time of a task can be out of the slots associated to its VM, this means that any instant may be the critical instant.

As we use the definition of the critical instant from Liu and Layland [6], we need to adapt this definition to our model. We have sporadic tasks. In order to have the highest load on the system, these tasks may be seen as periodic after a critical instant. The main difference between the Liu and Layland model and ours is fact task can only be run in the time slots of the VM they belong to. Tasks cannot be executed during idle slots during which their VM is inactive. This time may be seen as a task (for a given VM there is as many idle tasks as this VM has a segment. We may now introduce the k^{th} idle task of the pattern ρ^s: τ_k^{idle}. Its parameters are I_k^s for its duration (constant) and $\mathcal{T}^f$ for its period. We do not need a specific deadline so it may be equal to the period. The parameters and the activation time of this task cannot be changed.

This task is always executed as soon as it is released. As we used a FTP algorithm we can define the priority of this task as the highest. Thanks to this priority we assure that this task is executed as soon it is released and our model with the idle task is the same as in Liu and Layland [6].

THEOREM 1. *A candidate critical instant corresponds to a time when all the sporadic tasks are synchronous with the beginning of an idle task τ^{idle} and then periodic*

PROOF. The proof is divided in two cases. The first one is when the pattern contains only one segment. This case is trivial. By using the definition of the critical instant given by [6]. We have two kinds of task: the regular and the idle task (only one since there is only one segment). The only way to have a synchronous activations is then to activate the tasks at the same time as the only idle task.

The second case is when the pattern contains more than one segment. As said before, the critical instant is supposed to be when the sporadic tasks are synchronous. In our model we have multiple idle tasks for which we cannot change the activation time. Only the activation of the sporadic tasks may be changed. As the idle tasks have different activation

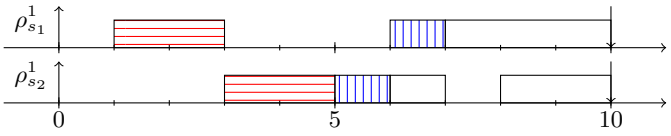

Figure 3: Same pattern with the same tasks synchronous with two different idle tasks.

times we cannot have a fully synchronous system. But we know for a given idle task that a critical instant can be found when the sporadic tasks are released synchronously with it.

That gives us candidates critical instants and not only one. These candidates are when the system is the most synchronous i.e. when all sporadic tasks are synchronous with the beginning of an idle slot (as other idle slots cannot be moved). □

In our model we therefore reduce at most as possible candidate critical instants. This method will drastically reduce the complexity of the computation of the response time of a sporadic task, by computing it $\mathcal{T}^{\int}$ times (for a task in the pattern ρ^s), the computation is done as many times as the pattern contains a segment.

We show in Figure 3 that with multiple segments the critical instant is not obviously when the tasks are synchronous with the greatest idle task. We used two tasks in this figure: $\tau_1(2, 10, 10)$ and $\tau_2(1, 10, 10)$. Here τ_1 has the greatest priority and τ_2 the lowest. The same pattern ρ^1 is represented shifted with on the two segments. The two shifts are represented by $\rho^1_{s_1}$ and $\rho^1_{s_2}$. With $\rho^1_{s_1}$, starts an idle task with cost 1 whereas $\rho^1_{s_2}$ starts with an idle task of cost of 3.

In this figure the execution of τ_1 is represented by the horizontal lines whereas the execution of τ_2 is represented by the vertical lines. We can see that τ_2 has a response time of 7 within $\rho^1_{s_1}$ and of 6 within $\rho^1_{s_2}$. The critical instant is then represented by $\rho^1_{s_2}$, which starts with the idle task with the lower cost. We cannot state of the exact critical instant when the pattern contains more than one segment without testing the WCRT of the candidates.

3.2 Worst Response Time with multiple segments

The case of multiple segments is more general than the case with only one segment. The analysis implies to consider a set of possible critical instants (each start time of an idle time slot) instead of just one. Thus, we represented the patterns as a list of segments with only an offset and an idle slot. Considering the pattern ρ_1 in Figure 2, the critical instants are at time 0 in the shifted representation $\rho^1_{s_1}$ and $\rho^1_{s_2}$. The worst-case response time r_i of a task τ_i of ρ^1 is given by the maximum response time among those computed using both representations. It is computed by applying the following equation:

$$r_i = \max_{\rho^i_s \in shift(\rho^i)} \left(r_i \left(\rho^i_s \right) \right) \tag{1}$$

where $shift(\rho^i)$ denotes the set of shifted representations of ρ^i and $r_i\left(\rho^i_s\right)$ denotes the response time of the task τ_i computed according to the shifted representation ρ^i_s.

The response time $r_i\left(\rho^i_s\right)$ for a shifted representation is computed by the method described in [8] and given by the following equation:

$$r_i \left(\rho^i_s \right) = \max_{q \in \{0 \ldots Q\}} \left(w_{i,q} - qT_i \right) \tag{2}$$

where q denotes the index of the job of the task τ_i and Q is the maximum number of jobs to consider to find the worst case response time such that $w_{i,Q} \leq (Q+1)T_i$.

The workload $w_{i,q}$ is computed by the following recurrence (adapted to our context):

$$w^{m+1}_{i,q} = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w^m_{i,q}}{T_j} \right\rceil C_j + \sum_{\sigma^s_k \in \vec{\sigma_s}} \left(\left\lceil \frac{w^m_{i,q} - O^s_k + I^s_k}{\mathcal{T}^s} \right\rceil I^s_k \right) \tag{3}$$

The last part of this equation computes the number of times that the idle slot had been scheduled for the k^{th} segment at time $w^m_{i,q}$.

4. SIMULATION

In this section we compare different algorithms used to generate the time slots of the patterns. Before comparing the algorithms, we present how we generate our task sets and patterns. The first generated objects are the tasks. To avoid a too long analysis, we reduce the hyper-period by using a part of the generator presented in [3]. In this paper, the authors present an algorithm to restrict the hyper-period of a generated task set. The values chosen to generate the periods in this algorithm gives most of the time harmonic periods. We will see why this point is important in our simulations. From this paper, we only use the algorithm in [3] to generate the periods.

Each period is computed by picking a set of numbers in a given matrix of prime powers. Hence we are sure that the hyper-period is bounded by the product of all prime powers of the matrix. For the remaining parameters, we use the UUniFast [2]. We generate 1000 task sets of 10 tasks by combining these two algorithms for each utilization value comprised between 0.3 and 0.8. Since the utilization value of a task τ_i is generated by UUniFast, its period T_i is randomly generated (using [3]) and its WCET C_i is directly derived from the equality $C_i = U_i \times T_i$. The arbitrary deadline D_i of τ_i is defined by $D_i = 10 \times T_i$.

Our simulated systems are composed of 10 tasks spread over 3 patterns. After the task generation, we proceed by the assignment of tasks to a pattern. Each pattern has at least one task since we uniformly generate a random integer between 0 and 2 that corresponds to the index of the pattern on which the generated task is assigned. When the number of tasks not yet associated to a pattern is less than the number of pattern with at least one task, we continuously generate a random number until this number corresponds to a pattern without tasks.

We have at this point the tasks generated and the patterns with at least one task. As we constructed our systems in order to have an utilization always less than 100%, the CPU is not fully used. To avoid this behaviour and increase the number of schedulable task set we will introduce a boosting operation to recover the unused utilization. The parameters of the patterns are deduced from the tasks associated to it. The boosting operation consists then in adding a specific amount of utilization to the utilization of a task to obtain a total utilization equals to 1. By doing this we will obtain a virtual set of tasks that we will use to generate the time slots of the pattern (by using a scheduling algorithm).

We boost the tasks by spreading the remaining utilization of the system over the tasks. Each task of the same pattern receives the same amount of utilization. The application

of boosting algorithms on a task set may imply deadline misses during the scheduling since the total utilization of the resulting task set is now equal to 1. But this schedule is only used to generate the patterns table and these deadline misses do not impact the scheduling of the original task set.

The number of tasks in the subset $\tau(\rho^s)$ (where $\tau(\rho^s)$ is the assigned subset of τ in the pattern ρ^s) is given by the term $|\tau(\rho^s)|$. To boost the tasks of τ according to the period, we introduce the term $T^{max}(\rho^s)$ that represents the largest period among the periods of tasks in $\tau(\rho^s)$. By definition, $T^{max}(\rho^s)$ is given by the following equation:

$$T^{max}(\rho^s) = \max_{\tau_i \in \tau(\rho^s)} (T_i)$$

We now compute the ratio α_T^s of remaining utilization assigned to ρ^s according to the period by the following equation:

$$\alpha_T^s = \frac{T^{max}(\rho^s)}{\sum_{\rho^u \in \rho} T^{max}(\rho^u)}$$

Algorithm 1: Boosting algorithm

Input: U^{rem} the remaining utilization
Input: ρ the table of patterns
Result: $\{U^\alpha\}$ the set of boosted utilization for each task

```
foreach ρ^s ∈ ρ do
    foreach τ_j ∈ τ(ρ^s) do
        U_j^α = U_j + (α_T^s × U^rem) / |τ(ρ^s)|;
    end
end
```

The complete algorithm for the boosting is described in Algorithm 1. The fraction $\frac{\alpha^s \times U^{rem}}{|\tau(\rho^s)|}$ corresponds to an equal distribution of the remaining utilization assigned to ρ^s between all tasks of $\tau(\rho^s)$. The result of this algorithm is the set $\{U^\alpha\}$ where the utilization U_j of each task τ_j of $\tau(\rho^s)$ is incremented by this fraction.

Once the set $\{U^\alpha\}$ is computed, one can derive the boosted parameters C_j^α and T_j^α for each task τ_j. Since the terms of all previous equations are rational numbers, C_j^α and T_j^α are also rational numbers. For the sake of readability, we do not discuss how to extract C_j^α and T_j^α from $\{U^\alpha\}$. The virtual set of boosted task may now be scheduled to find the time slots of each pattern.

Figure 4 shows our results of the schedulability of the generated tasks set according to the utilization. The different algorithms used to generate the time slots are: RM in both Preemptive (P) and Non-Preemptive (NP) modes, EDF in both P and NP modes, RR in P mode and WFQ in NP mode. We see that RR outperforms the other algorithms. The EDF and the RM algorithms come next with the same curve. As the period of the task generated are most of the time harmonic, the RM and EDF will give the same results in the preemptive mode. The other algorithms used in non-preemptive mode have really similar curves.

We can explain this results by looking at the formula which gives the WCRT. This formula adds to the classical response time the idle slots of every different patterns. So the more fair the algorithm is, the more task sets may be scheduled and that is why RR outperforms the others since it is the most fair algorithm used. This is also why the non-preemptive algorithms have such bad results compared to the preemptive ones.

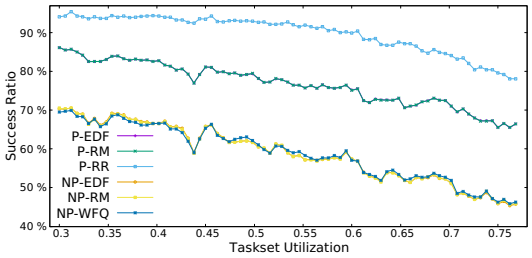

Figure 4: Simulation results

5. CONCLUSION

In this paper we proposed a schedulability analysis for applications run by a hypervisor. To preserve safety and security constraints, an application is allocated to one processor of a multiprocessor platform. One processor can execute several applications assigned to it in dedicated patterns of temporal slots managed by the hypervisor (one pattern per application). Each pattern is periodic and contains several slots that are not necessarily periodic in the pattern. Each slot can execute several tasks. We propose an exact schedulabity test for an application modeled by sporadic tasks and executed in the slots of an arbitrary pattern. Then we propose several pattern constructions derived from the scheduling of task with a well-known real-time schedulings (RR, WFQ, RM and EDF). Finally, we compared the success ratio of the different pattern construction heuristics.

As a further work, we would like to extend this work by expressing the problem of finding valid pattern of slots as a linear programming problem, taking into account all the constraints of the tasks.

Another extension will consist in considering the possibility to assign a pattern of slots for an application on several processors. We would like to compare global algorithms, semi-partitioned algorithms for the construction of pattern of slots.

6. REFERENCES

[1] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17:5–22, 1999.

[2] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1–2):129–154, 2005.

[3] J. Goossens and C. Macq. Limitation of the hyper–period in real-time periodic task set generation. In *Proc. of RTS*, pages 133–148, 2001.

[4] A. Guasque, P. Balbastre, and A. Crespo. Real-time hierarchical systems with arbitrary scheduling at global level. *Journal of Systems and Software*, 119:70–86, 2016.

[5] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of RTSS*, pages 201–209. IEEE, 1990.

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

[7] T. Nolte. Compositionality and cps from a platform perspective. In *Proc. of RTCSA*, volume 2, pages 57–60, 2011.

[8] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.