Towards Hard and Soft Real-time Operating Systems for Multicore Heterogeneous Architectures

Alessio Balsini

Real-time Systems Laboratory
Scuola Superiore Sant'Anna

A thesis submitted for the degree of
Doctor of Philosophy

Supervisor
Prof. Tommaso CUCINOTTA

Tutor
Prof. Luca ABENI

November 30, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Contributions

In the last decades, the unprecedented evolution of computing and networking devices opened the doors to the use of computing services in a number of different applications and the accessibility of personal and mobile devices to the majority of the people. The technological evolution grew together with the requirements imposed by the users, resulting in the need to provide, among others, high computational power, low latency, and limited power consumption.

The growing capabilities of the computing architectures, both in terms of computing and communication speed, the former with the reduction of chip sizes and the introduction of multicore and manycore computing platforms, the latter with the global Internet accessibility, enabled the development and diffusion of a variety of new different applications, resulting in new requirements and challenges to tackle.

It is not possible to think of the devices we have nowadays as the devices available ten years ago. The smartphones we have nowadays completely disrupt high-end personal computers of 10 years ago in all the aspects, from memory capacity, to computing speed, power consumption, and size. This capability improvement allowed software developers to realize more and more complex applications, spanning from multimedia players to video recording, image and voice recognition based on artificial intelligence, web browsing, and so on, thus requiring a full operating system to manage all the different workload and provide isolation for security and performance purposes.

This relentless evolution of the computing infrastructure led also to an increasing complexity of the computing architectures, that now involve billions of interconnected devices. And the more complex is the devices architecture, the more complex is the software to fully exploit the available computational capabilities. One example can be the need of spreading the workload among many distributed devices, like in cloud architectures, or the management of data streams generated by a number of interconnected embedded devices, that may be collected by a single server in the cloud, following the Internet of Things (IoT) paradigms.

The request for flexible computing architectures also gave birth to services that in the past were provided by custom hardware, like in the case of Network

Functions Virtualization (NFV).

In most of the mentioned cases, one of the fundamental requirements is to provide small response times or end-to-end latency, that often translates into the ability of providing real-time guarantees or, in other words, temporal predictability.

The problem is particularly challenging in hard real-time settings, such as in the automotive industry, that has been revolutionized in the last decades by the evolution of the computing technologies. It is nowadays usual to find in a vehicle tens, sometimes hundreds, interconnected microcontrollers, hosting complex software components implementing evolved sensing and control features that were just impossible to realize 20 years ago. Another big step in the computerisation of the vehicles has been due to the introduction of fully autonomous vehicles, that are forecast to dominate the roads in the upcoming decades. In this example, all the vehicle decisions are taken in real-time according to the events happening both inside and outside the car, and may also depend on the data exchanged through the Internet (e.g., maps, traffic, etc.), for which the elaboration of all of this information causes computationally intensive workloads. In the automotive field, also the requirements specification evolved and increased in difficulty, due to both the number of involved devices and the complexity of the requested functionalities. Nowadays there are research branches and companies involved in the management of the requirements, covering the process the goes from the disambiguation of the requirements to the testing and validation for compliance of whole systems (e.g., automatic code generation and formal methods), aiming at simplifying the connections between specifications, design, and validation. For these examples, the most critical control units of the vehicle introduce even harder temporal constraints, requiring predictable execution of the software and proper scheduling theory to provide real-time guarantees.

Another essential element that the mentioned computing architectures have in common, and that has a strong impact on the system performance, for example, soft or hard real-time response time, is the data placement. It is easy to observe how the placement of data affects especially the latency of computing platforms in different domains, from the proper management of cache memories and scratchpads within a single SoC, to the distribution of data among distributed databases in the case of cloud architectures.

In all the previously mentioned examples, one common factor that is gaining momentum is the use of heterogeneous hardware architectures, for which the computations are demanded to ad-hoc devices like Graphics Processing Unit (GPU), now evolved to General-Purpose computing on GPU (GPGPU), or other hardware accelerators implemented with Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA), Tensor Processing Unit (TPU), or different kind of general purpose processors, some with higher computational performance, others with higher energy efficiency. These computational units are nowadays commonly integrated in the same System-on-a-Chip (SoC). These new architectures provide a huge potential that can be exploited and, also if these provide advantages in the execution of the deployed applications, predictable execution must be preserved if real-time constraints are specified.

In these scenarios the use of open-source operating systems is becoming more and more popular, and this popularity is also demonstrated by the foundation of consortia like OSADL[1] aiming at bringing together experts to spread the use of open source software also in industrial environments where legal requirements, safety and security standards and certifications are mandatory. In this environments, the use of the Linux kernel is a constant.

With the awareness of that, most of the works that will be mentioned later have been validated with proper modifications or improvement to the Linux kernel.

Both in the cases of battery-powered mobile devices and in cloud data centers, the impact of the energy consumption cannot be neglected, the former to improve the life of the device, the latter to reduce the cost. The classical mechanisms to reduce the energy consumption by modulating the frequency of the processor or migrating the tasks to different computational units often do not consider how the policy affect the computational and latency performance. This thesis also addresses the problem of providing a low-latency, energy-efficient strategy in the management of multimedia processing on heterogeneous mobile devices, that may be easily extended to other platforms and applications, with mechanisms based on the processor utilization and the introduction of interfaces allowing for the communication between the applications layer and the kernel layer to exchange temporal requirements, so that the kernel has precise hints about the timing requirements of the application, and its evolution in time.

Contributions and Thesis Structure This thesis addresses several open issues that have been identified in the mentioned scenarios.

In the automotive field, it is of paramount importance to be able to provide a predictable execution of all the critical engine sensing and control tasks, often interacting with each other, not to mention the tough validation process often involving formal methods for which the requirements are expressed in formal languages. In these cases, the latency between the time at which an event occurs and the time at which the whole chain of actions is triggered plays a critical role in the proper functioning of the whole system. Chapter 3 provides an introduction on the components of a realistic model of an AUTOSAR multicore engine control unit, and provides a set of three novel contributions. First, Section 3.1 gives formal definitions of the different kinds of *latency* that can be measured among interdependent tasks, besides an analysis on the worst case response time. This work highlights the strong dependency between the worst case response times and the data and tasks placements among the different memories and cores, accordingly. Second, Section 3.2 goes further into details, proposing two different approaches to solve the problem of optimizing the data distribution among different memories to minimize the response times of the tasks. Finally, an important issue that has been identified in this area is the lack of a bridge between the expression of system requirements with formal languages and the verification phase, requiring the manual translation of the requirements into test cases for the system verification. To solve this issue, a support tool has been developed, as explained in Section 3.3.

With the need of providing temporal guarantees and isolation among processes running on the same machine, new scheduling mechanisms are currently available in modern operating systems, for example, the SCHED_DEADLINE

[1]More information available at: `https://www.osadl.org/`.

deadline-based scheduler in the Linux kernel. Chapter 4 presents several enhancements to the SCHED_DEADLINE scheduler based on the real-time scheduling theory, thus improving the system predictability in the execution of latency-sensitive workloads. In particular, Section 4.1 presents a developed hierarchical scheduling technique and compares the theoretical performance with the experimental results achieved with a real implementation in the Linux kernel. Section 4.2 applies the just mentioned hierarchical scheduler to the cloud computing, for which the proposed approach was able to guarantee predictable QoS and stable performance, compliant with the theory. These works highlighted a shortcoming in the scheduling algorithm on which SCHED_DEADLINE relies, in particular, it presents a wrong behavior when managing tasks that suspend their executions: a common problem that arises, for example, by the use of locks or blocking system calls. This issue is tackled in Section 4.3, which provides both a correct schedulability analysis for self-suspending tasks and the implementation of the newly developed scheduler in the Linux kernel.

The diffusion of heterogeneous architectures, involving different kinds of computational units within the same computing system, introduced a flexibility potential that is hard to be fully exploited. Chapter 5 describes different aspects of heterogeneous architectures. Section 5.1 presents a novel solution for dealing with dynamic workload changes in professional audio production scenarios, with reference to the Android platform on heterogeneous Arm big.LITTLE computing architectures. The introduced solution allows for a 40% in power saving during audio playback at low latency set-ups, when compared to the current Android low-latency mechanisms, but it also reduces the playback latency of one order of magnitude at the cost of a negligible increase in power consumption, enabling new interactive professional audio applications on Android, that have been deemed impossible on the platform, so far. The development of the just mentioned work, showed the lack of complete models of both energy consumption and computing times at the varying of computing architecture and CPU frequency, that could be used for a preliminary evaluation of scheduling design choices. This problem has been addressed in Section 5.2, which presents the two realistic models, implemented and validated within a real-time tasks simulator. Another popular and successful heterogeneous architecture of the last decades is achieved by coupling CPUs and FPGAs in the same chip. This architecture combines the flexibility of the software that can be executed by the CPU, and the computational performance of the FPGA, whose hardware can be dynamically programmed to implement specific functions. Since the FPGA has a limited number of reprogrammable logic units, it may happen that all the accelerators required by the same application cannot contemporaneously coexist in the FPGA, so Section 5.3 proposes a framework to support the real-time dynamic reconfiguration of the FPGA and its full implementation on Linux, tested on a real board.

Chapter 2

List of Publications

Related to Chapter 3

- A. Balsini, A. Melani, P. Buonocunto, and M. Di Natale, “Fmtv 2016: Where is the actual challenge?”, Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS’16), in conjuction with the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016), July 2016. [18]
- A. Biondi, P. Pazzaglia, A. Balsini, and M. D. Natale, “Logical execution time implementation and memory optimization issues in autosar applications for multicores”, Proceedings of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS’17), in conjuction with the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), July 2017. [38]
- A. Balsini, M. D. Natale, M. Celia, and V. Tsachouridis, “Generation of simulink monitors for control applications from formal requirements”, in 2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES), 2017, pp. 1–9. [16]
- D. Calvaresi, P. Sernani, M. Marinoni, A. Claudi, A. Balsini, et al., “A framework based on real-time os and multi-agents for intelligent autonomous robot competitions”, in 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), 2016, pp. 1–10. [47]

Related to Chapter 4

- L. Abeni, A. Balsini and T. Cucinotta, “Container-based real-time scheduling in the linux kernel”, ACM SIGBED Review - Special Issue on Embedded Operating Systems Workshop (EWiLi ’18) 2018. [5]
- T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, “Virtual network functions as real-time containers in private clouds”, in 11th IEEE International Conference on Cloud Computing (IEEE CLOUD 2018), Jul. 2018. [59]
- A. Biondi, A. Balsini, and M. Marinoni, “Resource reservation for real-time self-suspending tasks: Theory and practice”, in Proceedings of the 23rd

International Conference on Real Time and Networks Systems, ser. RTNS '15, Lille, France: ACM, 2015, pp. 97–106, isbn: 978-1-4503-3591-1. [34]

Related to Chapter 5

- A. Balsini, L. Pannocchi, and T. Cucinotta, "Modeling and simulation of power consumption for real-time embedded heterogeneous architectures", ACM SIGBED Review - Special Issue on Embedded Operating Systems Workshop (EWiLi '18), 2018 [19]
- M. Pagani, M. Marinoni, A. Biondi, A. Balsini, and G. Buttazzo, "Towards real-time operating systems for heterogeneous reconfigurable platforms", in Proc. of the 12th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2016), Toulouse, France, July 5, 2016. [147]
- A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, et al., "A framework for supporting real-time applications on dynamic reconfigurable fpgas", in 2016 IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 1–12. [31],
- M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, "A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration", in 2017 30th IEEE International System-on-Chip Conference (SOCC), 2017, pp. 96–101. [146]

Under Review

- A. Balsini, T. Cucinotta, L. Abeni, J. Fernandes, P. Burk, et al., "Energy-efficient low-latency audio on android", Submitted to Journal of Systems and Software, June, 2018.

Chapter 3

Scheduling Latency in Control Systems

This chapter focuses on problems related to providing real-time scheduling guarantees or at least scheduling latency bounds in automotive applications and many control systems.

Researchers have been working on the definition of reliable and theoretically sound algorithms for the verification of the real-time requirements for computing systems in automotive, especially in terms of scheduling latency of chains of interdependent tasks. This research topic is challenging both for the complexity of the involved hardware and the software architecture, but is also lacking the theory basics, like a proper, formal, definition of latency. Section 3.1 will present proper definitions of different kinds of latency, as well as the analysis and simulation for a realistic automotive computing environment.

The results presented in Section 3.1 also showed the close relationship between the worst-case response time of the tasks, how the data is distributed among different memories, and how the tasks are pinned among the different cores on a multicore environment. To tackle this research problem, two different solutions are proposed in Section 3.2, one obtained with a mixed-integer linear programming (MILP) formulation, and the other with genetic algorithms.

Finally, a significant trend in automotive is to simplify and better organize the software development, with the purpose of reducing the error probability and simplifying the certification process. This is achieved with an extensive use of automatic code generation tools relying on block diagrams. In this field, the validation of the developed models is often performed by defining the system requirements (e.g., the minimum value of a signal or the maximum delay of an event) with formal languages. Concerning this need, Section 3.3 presents a tool that simplifies the translation of the requirements expressed by the means of formal languages into Simulink block diagrams.

It is worth to mention that the importance and complexity of the just mentioned problems is also demonstrated by the interest from the industry, and the organization of ad-hoc workshops[1].

[1]The *Waters* workshop and the *Waters Challenge*, part of the *Euromicro Conference on Real-Time Systems*, are tracks concerning real-time in automotive organized with the support of Bosch.

3.1 Scheduling Latency for AUTOSAR Components: Definition, Analysis and Simulation

A particularly relevant work that arose during the studies undertaken in the research activities of this Ph.D., has been related to the problem of providing analysis methods for real-time multicore fuel injection applications.

We took the chance to dig deep into these issues by participating into the FMTV'16 challenge[2], proposed in the context of the WATERS 2016 workshop within the Euromicro Conference on Real-Time Systems, which proposed to research groups as a case study and benchmark to compare different analysis methods for real-time multicore fuel injection applications. The nature of the problem is clear enough and the challenge can be met by a set of conventional analysis techniques. However, the formulation of the problem and its practical solution are more than likely to reveal a number of additional issues that go from the model of the application, to analysis techniques that consider with much better precision the details of the hardware platform, to the need for synthesis and optimization methods.

This challenge consists of a timing analysis problem in which the AUTOSAR model of a set of cooperating tasks in a fuel injection application is deployed onto a 4-core platform. The objective of the challenge is to apply different analysis methods (worst-case, simulation-based and possibly stochastic) to models of the system with an increasing level of accuracy with respect to the memory placement of communication variables. At the simplest level, memory access times are simply neglected; next, different access times are assumed under the hypothesis of global or local memory allocation; and, finally, the problem of optimizing the placement of the memory items is presented.

From the architecture standpoint, two solutions to the problem are presented, developed as a result of this Ph.D. activity: (i) the simulation of the time behavior using the open source scheduling simulator RTSIM [21], and (ii) the analysis of the task set for its worst-case behavior, using a set of formulas derived from the problem description and obtained by adaptation of classical results.

The results of these two analysis methods are provided (with an additional discussion on how to tackle the memory access time problem), as well as several issues that are worth discussing. Among those:

- The **definition of response times when the system contains chains of tasks or runnables communicating asynchronously**. The challenge refers to a set of definitions (*reactive* and *age*) for which an application-level justification is not clear enough and for which, despite being formally presented in [82], solutions in analytical closed form or as algorithms have been presented only recently [24].

- Next, while the challenge has the merit of restoring to the foreground the consideration of hardware features and issues, its **description of the hardware architecture details** is still incomplete and simplistic. For example, the FIFO arbiter controlling accesses to shared memory is likely to be integrated within the crossbar or possibly placed after it, but this information can only be guessed and would affect the access times to

[2]More information available at: `https://waters2016.inria.fr/challenge/`.

memory. More details of the hardware architecture and the assumptions that have been made in this work will be provided later in Section 3.1.2.

- Finally, and most important, this work highlights **the runnables placement problem**, which is most likely the most relevant design issue for a time critical multicore system like this.

3.1.1 System Model and Notation

The challenge model is in large part compliant with the AUTOSAR metamodel and adopts from it definitions and most of the semantics for activation and communication of functions (runnables in AUTOSAR). An attempt at the formal characterization of the challenge model is the following.

A task τ_i is composed of an ordered sequence of n_i runnables $\rho_{i,1}, \dots, \rho_{i,n_i}$, each of which has its execution time defined as a statistical distribution $\mathcal{C}_i$, which is defined as a truncated Weibull distribution for most if not all the runnables in the model. For the purpose of worst-case analysis, the worst-case execution time (WCET) $C_{i,j}$ and a best-case execution time $c_{i,j}$ may be computed from the distribution $\mathcal{C}_i$.

The scheduling of each task is also controlled by its scheduling mode (cooperative or preemptive) and its priority π_i, with preemptive tasks having higher priority than cooperative tasks, and cooperative tasks only preempting each other at runnable boundaries.

The model also defines deadlines that apply to tasks and task chains. For tasks, deadlines bound the worst case completion time with respect to the activation and match the common definition of a relative deadline D_i. Also, all tasks are assumed to be periodic or sporadic, with a period or a minimum inter-arrival time T_i. When applicable, relative deadlines are constrained to be smaller than or equal to periods, i.e., $D_i \leq T_i$. In the end, each task is assumed to be defined by a tuple (C_i, c_i, D_i, T_i), where $C_i = \sum_{j=1}^{n_i} C_{i,j}$, $c_i = \sum_{j=1}^{n_i} c_{i,j}$.

The worst-case response time of the j-th runnable of task τ_i is denoted as $R_{i,j}$, while $r_{i,j}$ denotes its best-case response time. $hp^P(i)$ and $hp^C(i)$ denote the set of preemptive and cooperative tasks, respectively, having priority greater than τ_i. The union of the two disjoint sets is denoted as $hp(i) = hp^P(i) \cup hp^C(i)$.

As for end-to-end chains, the assumed model is based on the asynchronous propagation of information by means of shared data variables. These variables (labels in the model) are read and written by the runnables.

Figure 3.1 illustrates the three effect chains that are analyzed in the context of the challenge[3].

The following semantics have been considered for end-to-end latency calculation (from [82]):

- *Last-to-First* (L2F): it considers the delay between the last input that is not overwritten until the first output generated with the same input;
- *First-to-First* (F2F), or *Reactive*: it considers the delay between the first input that may be overwritten until the first output generated with the next different input;

[3] Note that, in the third chain, Label 2197 is replaced with Label 646 to fix a mistake in the model (Label 2197 is not read, nor written by the last two runnables in the chain, while Label 646 is the only one that satisfies the read/write relation imposed by the chain).

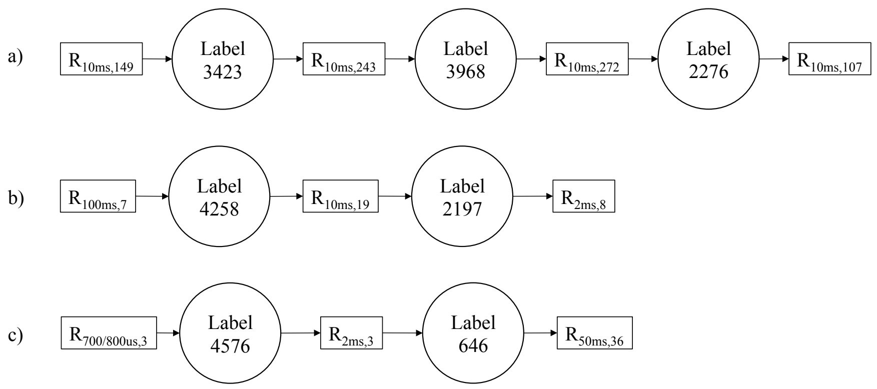


Figure 3.1: Effect chains in the model.

- *Last-to-Last* (L2L), or *Maximum Age*: it considers the delay between the last input that is not overwritten until the last output, considering duplicates.

The problem with this definition is that it is hardly formal, and even in the original reference there seems to be no single point in which a formal definition appears. Hence, the following definitions are used.

Assume a chain of periodic communicating runnables $\Gamma = \{\rho_1, \rho_2, \ldots \rho_n\}$. Also, assume $a_{j,h}$ denoting the h-th activation of runnable ρ_j, $f_{j,h}$ its finishing time, and $I_{j,h}$ and $O_{j,h}$ are the sets of input and output values that are respectively read from and written to the labels accessed by the h-th instance of ρ_j.

Then, the L2F latency of the chain Γ is the maximum value $f_{n,r} - a_{1,p}$ (finishing time of the r-th instance of ρ_n minus the activation time of the p-th instance of ρ_1), such that for some p, q, r:

$$\begin{aligned}\forall i = 1, \ldots, (n-2)\;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r}\\ \text{and }\; I_{i+1,q} \neq I_{i+1,q-1} \;\text{ and }\; I_{i+2,r} \neq I_{i+2,r-1}.\end{aligned}$$

Similarly, the F2F latency of the chain Γ is the time interval between the latest $a_{1,p}$ and the earliest $f_{n,r+1}$ such that for some p, q, r:

$$\begin{aligned}\forall i = 1, \ldots, (n-2)\;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r}\\ \text{and }\; I_{i+1,q} \neq I_{i+1,q+1} \;\text{ and }\; I_{i+2,r} \neq I_{i+2,r+1}.\end{aligned}$$

Finally, the L2L latency of the chain Γ is the maximum value $f_{n,r} - a_{1,p}$ (finishing time of the r-th instance of ρ_n minus the activation time of the p-th instance of ρ_1), such that for some p, q, r:

$$\forall i = 1, \ldots, (n-2)\;\; O_{i,p} = I_{i+1,q} \;\text{ and }\; O_{i+1,q} = I_{i+2,r}.$$

Figures 3.2 and 3.3 exemplify the definitions in the case of undersampling and oversampling effects, respectively. In particular, referring to the chain $\{\rho_1, \rho_2, \rho_3\}$ in Figure 3.2, the end-to-end delay by the L2F semantics corresponds to the time interval between the activation $a_{1,1}$ and the finishing time of the runnable activated at time $a_{3,1}$; the end-to-end delay by F2F corresponds to the time interval $[a_{1,1}, f_{3,2}]$. By L2L, it is measured as for the L2F semantics (i.e., $f_{3,1} - a_{1,1}$). In case of oversampling, as shown in Figure 3.3, the end-to-end delay can be measured by the L2F semantics as $f_{3,2} - a_{1,1}$; by F2F it is $f_{3,5} - a_{1,1}$, while the L2L semantics accounts for the same data read by multiple runnable instances (e.g., in the time interval $f_{3,4} - a_{1,1}$).

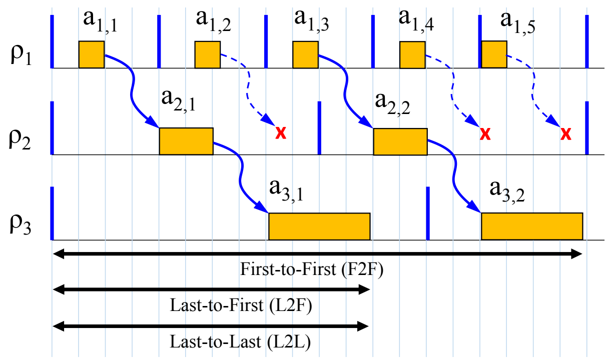


Figure 3.2: End-to-end delay in the case of undersampling.

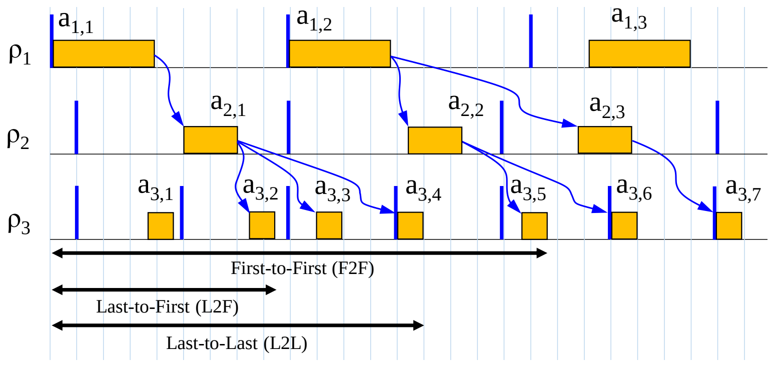


Figure 3.3: End-to-end delay in the case of oversampling.

The definition ambiguity leaves open a fundamental issue: **what is the actual meaning and relevance (in application terms) of such definitions**?

3.1.2 Worst-case Latency Analysis

This section discusses the analytical approach to compute the worst-case response times for tasks and chains, with and without considering the timing for the access to shared or local memory.

Analysis Without Memory Access Times

For any *preemptive* task, the worst-case response time of runnable $\rho_{i,j}$ is given by the fixed point iteration of the following formula (starting with $R^0_{i,j} = \sum_{h=1}^{j} C_{i,h}$):

$$R_{i,j} = \sum_{h=1}^{j} C_{i,h} + \sum_{k \in hp(i)} \left\lceil \frac{R_{i,j}}{T_k} \right\rceil C_k. \tag{3.1}$$

The above formula quantifies the higher-priority interference suffered by $\rho_{i,j}$ by considering the synchronous periodic arrivals of higher-priority tasks.

For cooperative tasks, the worst-case response time needs to consider also the blocking time by lower-priority cooperative runnables and the fact that the last runnable does not suffer any preemption by higher-priority cooperative tasks once it has started executing. In addition, by analogy with the limited preemptive scheduling with fixed preemption points [45], it is not enough to compute the response time of the first job after the critical instant. In particular, the computation must be carried out for all jobs $s \in [1, K_i]$ falling within the so called Level-i Active Period L_i, computed as:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \sum_{k \in hp(i)} \left\lceil \frac{L_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}, \tag{3.2}$$

such that $K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$. Therefore, in case of a cooperative task τ_i, it is possible to compute the worst-case finishing time of the s-th job of $\rho_{i,j}$ by the fixed point iteration of the following formula:

$$f_{i,j}^s = \sum_{h=1}^{j} C_{i,h} + B_{i,j} + (s-1)C_i + \sum_{k \in hp^P(i)} \left\lceil \frac{f_{i,j}^s}{T_k} \right\rceil C_k + \sum_{k \in hp^C(i)} \left(\left\lfloor \frac{f_{i,j}^s - C_{i,j}}{T_k} \right\rfloor + 1 \right) C_k,$$

where

$$B_{i,j} = \max_{\substack{q \in lp^C(i) \\ h=1,\dots,n_q}} C_{q,h}$$

represents the maximum blocking time imposed by lower-priority cooperative tasks.

Then, the worst-case response time of $\rho_{i,j}$ can be computed as:

$$R_{i,j} = \max_{s \in [1, K_i]} f_{i,j}^s - (s-1)T_i. \tag{3.3}$$

Worst-case Start Time Computation Another quantity of interest for the end-to-end latency computation is the worst-case start time $S_{i,j}$ of runnable $\rho_{i,j}$. The calculation is the same for both the case of preemptive and cooperative tasks, and is given by:

$$S_{i,j} = \epsilon + \sum_{h=1}^{j-1} C_{i,h} + \sum_{k \in hp(i)} \left\lceil \frac{S_{i,j}}{T_k} \right\rceil C_k, \tag{3.4}$$

where ϵ is an arbitrarily small constant.

Best-case Response Time Computation For preemptive tasks, the best-case response time of runnable $\rho_{i,j}$ is [41]:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp(i)} \left(\left\lceil \frac{r_{i,j}}{T_k} \right\rceil - 1 \right) c_k. \tag{3.5}$$

For cooperative tasks, a lower-bound on the best-case response time can be computed by considering a zero blocking-time from lower-priority tasks and the minimum amount of interference from higher-priority tasks [41, 42]:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp^P(i)} \left(\left\lceil \frac{r_{i,j}}{T_k} \right\rceil - 1 \right) c_k + \sum_{k \in hp^C(i)} \left\lfloor \frac{r_{i,j} - c_{i,j}}{T_k} \right\rfloor c_k. \tag{3.6}$$

End-to-end Latency Calculation

The end-to-end latencies have been computed according to the semantics reported in Section 3.1.1. For each chain, first the end-to-end latency is computed by the Last-to-First (L2F) semantics, and then is extended to obtain the latencies by the F2F and L2L semantics.

Last-to-First Semantics The end-to-end latency of chain $\rho_1, \dots, \rho_N$, according to the L2F semantics and as shown in Figures 3.2 and 3.3, can be computed as:

$$\sum_{i=1}^{N-1} (R_i + \min(T_{i+1} - r_{i+1}, T_i)) + R_N. \tag{3.7}$$

First-to-First Semantics With respect to the L2F semantics, the F2F semantics requires to add one cycle delay for the first runnable in the chain, in order to consider the previous input. Therefore, the end-to-end latency of chain $\rho_1, \dots, \rho_N$, according to the F2F semantics, can be computed as:

$$T_1 + \sum_{i=1}^{N-1} (R_i + \min(T_{i+1} - r_{i+1}, T_i)) + R_N. \tag{3.8}$$

Additionally, the F2F semantics considers previous inputs that are overwritten. In order to compute how many times in the worst case an input is overwritten between consecutive stages of the chain (i.e., between runnables ρ_i and ρ_{i+1}), it must be found the largest possible integer $\overline{n} \geq 1$ that satisfies:

$$T_{i+1} + S_{i+1} - r_{i+1} \geq \overline{n} T_i + r_i - R_i. \tag{3.9}$$

This relation guarantees that the longest interval between two consecutive reads is greater than the shortest interval between $\overline{n}$ consecutive writes. If the above relation holds (i.e., input overwriting takes place), the end-to-end latency of chain $\rho_1, \dots, \rho_N$ is computed as:

$$T_1 + \sum_{i=1}^{N-1} (R_i + \overline{n} T_i) + R_N. \tag{3.10}$$

Last-to-Last Semantics With respect to the L2F semantics, the L2L also considers subsequent outputs that are overwritten. In order to compute how many times in the worst case an output is overwritten between consecutive stages of the chain, it must be found the largest possible integer $\widehat{n} \geq 1$ that satisfies:

$$T_i - r_i + R_i \geq \widehat{n}T_{i+1} - r_{i+1} + S_{i+1}. \tag{3.11}$$

This relation guarantees that the longest interval between two consecutive writes is greater than the shortest interval to perform $\widehat{n}$ consecutive reads. If the above relation holds (i.e., output overwriting takes place), the end-to-end latency is computed as:

$$\sum_{i=1}^{N-1} (R_i + \widehat{n}T_{i+1} - r_{i+1}) + R_N. \tag{3.12}$$

Otherwise, the end-to-end latency by the L2L semantics is as the one obtained under the L2F semantics.

Analysis with Memory Access and Arbitration Times

In the proposed model, the four cores contend for access to a shared global memory (GRAM) with FIFO arbitration. Each read/write access to GRAM costs 9 cycles (there is no caching effect). Therefore, in the worst case each memory access might be blocked by pending accesses from other cores, i.e., each access can be delayed for $9(m-1) = 27$ cycles. Adding up the memory access cost for the current request, a worst-case memory-access penalty of 36 clock cycles is obtained. By exploiting the knowledge of how many labels are read/written by each runnable, it is possible to compute the worst-case memory access latency for its read/write phases.

In the best case, memory accesses do not experience any delays from other cores, leading to a best-case memory-access time of 9 clock cycles. Accordingly, the best-case memory access latency for the read/write phases can be computed.

Such values need to be added to the execution time of each runnable, to which the analysis described in Section 3.1.2 can be applied identically.

The worst-case estimate of $9(m-1)$ cycles implies that the 9 cycles access cost is repeatedly applied on each FIFO access, which is most likely a pessimistic estimate given the lack of detailed information on the hardware (memory) configuration. Careful consideration of the memory access costs **require a model of the hardware more detailed than what is typically available in scheduling analysis papers**.

End-to-end Latency Calculation

The end-to-end latency calculation can be performed as described in Section 3.1.2, with the following differences.

Last-to-First Semantics Equation (3.7) is replaced by:

$$\sum_{i=1}^{N-1} (R_i - r_{i+1}^{read} + \min(T_{i+1}, T_i)) + R_N, \tag{3.13}$$

where r_i^{read} denotes the best-case response time of the read phase of ρ_i.

First-to-First Semantics Equation (3.9) is replaced by:

$$T_{i+1} + S_{i+1} - r_{i+1}^{read} \geq \overline{n} T_i + r_i - R_i. \tag{3.14}$$

Last-to-Last Semantics Equation (3.11) is replaced by:

$$T_i - r_i + R_i \geq \widehat{n} T_{i+1} - r_{i+1}^{read} + R_{i+1}^{read}, \tag{3.15}$$

where R_i^{read} denotes the worst-case response time of the read phase of ρ_i, which can be computed similarly as in Section 3.1.2.

Experimental Evaluation

In order to make the system analyzable, the WCETs of those tasks that were not deemed schedulable by this analysis were scaled down by considering the largest scaling factor $\sigma \in (0, 1]$ that guarantees schedulability. In particular, starting from $\sigma = 1$, WCETs are iteratively scaled down in steps of 0.01 until the system became schedulable according to the proposed analysis.

Effect Chain 1 In the effect chain 1: (i) all runnables belong to the same task (*Task_10ms*, allocated to core 3), hence all runnables are bound to the same rate; (ii) there is backward communication between the third and the fourth runnable, which implies a one cycle delay until the last datum is read. Therefore, the worst-case end-to-end latency of this effect chain by L2F can be computed as:

$$L_1^{L2F} = T_{10ms} + R_{10ms,107} = 13376\ \mu s. \tag{3.16}$$

Given that all runnables belong to the same task, this result is valid also when considering the L2L semantics. As for the F2F semantics, the analysis needs to consider a one cycle delay for the first runnable, that is:

$$L_1^{F2F} = 2T_{10ms} + R_{10ms,107} = 23376\ \mu s. \tag{3.17}$$

Effect Chain 2 Unlike the previous chain, runnables in this chain belong to different tasks with different rates. In this case, the end-to-end latency calculation should also consider the over-sampling effect between pairs of consecutive runnables. By the L2F semantics, applying Equation (3.7), returns:

$$\begin{aligned} L_2^{L2F} &= R_{100ms,7} + \min(T_{10ms} - r_{10ms,19}, T_{100ms}) \\ &+ R_{10ms,19} + \min(T_{2ms} - r_{2ms,8}, T_{10ms}) \\ &+ R_{2ms,8} = 52222\ \mu s \end{aligned}$$

As for the F2F semantics, due to the over-sampling effect, there are no input overwritings (Condition (3.9) is never verified), hence the end-to-end latency is simply given by:

$$L_2^{F2F} = L_2^{L2F} + T_{100ms} = 152222\ \mu s.$$

Finally, the end-to-end latency computation for the L2L semantics requires to verify Condition (3.11) for any pair of consecutive runnables. In this case, $\widehat{n} = 13$ is obtained for the first stage and $\widehat{n} = 5$ for the second, which yields:

$$\begin{aligned} L_2^{L2L} &= R_{100ms,7} + 13 \cdot T_{10ms} - r_{10ms,19} + R_{10ms,19} \\ &+ 5 \cdot T_{2ms} - r_{2ms,8} + R_{2ms,8} = 180222\ \mu s. \end{aligned}$$

Chain	L2F I	L2F II	F2F I	F2F II	L2L I	L2L II
1	13376	13383	23376	23383	13376	13383
2	52222	52796	152222	152796	180222	180796
3	41953	42448	127553	130040	41953	43248

Table 3.1: End-to-end latency upper bounds (μs) for the first (I) and second (II) challenge.

Effect Chain 3 Also in this case, runnables belong to different tasks with different rates. Task periods have increasing values, leading to an under-sampling effect.

By the L2F semantics, applying Equation (3.7) returns:

$$L_3^{L2F} = R_{700/800us,3} + \min(T_{2ms} - r_{2ms,3}, T_{700/800us}) + R_{2ms,3} + \min(T_{50ms} - r_{50ms,36}, T_{2ms}) + R_{50ms,36} = 41953\ \mu s$$

Due to the sporadic nature of the first runnable, $T_{700/800us} = 800\ \mu s$ is assumed in order to maximize the latency.

The end-to-end latency by the F2F semantics requires adding one cycle delay with respect to L2F and to verify Condition (3.9) for any pair of consecutive runnables. In this case, $\overline{n} = 2$ is obtained for the first stage (again, assuming $T_{700/800us} = 800\ \mu s$) and $\overline{n} = 43$ for the second, which yields:

$$L_3^{F2F} = T_{700/800us} + 2 \cdot T_{700/800us} + R_{700/800us,3} + 43 \cdot T_{2ms} + R_{2ms,3} + R_{50ms,36} = 127553\ \mu s.$$

Finally, the end-to-end latency for the L2L semantics is equal to the L2F case, because no output is overwritten due to the under-sampling effect.

Similar calculations are performed to compute end-to-end latencies accounting for memory effects, as described in Section 3.1.2.

Table 3.1 summarizes the obtained end-to-end latencies calculated according to the different semantics adopted, for each of the two challenges.

3.1.3 Model Simulator

The analysis by simulation of the challenge model has been performed by a purposely developed extension [138, 167] to the C++ RTSIM [21] scheduling simulator.

Data Acquisition

The engine control application model that is the subject of the challenge is defined by an XML file that can be parsed to obtain the model data. The model information is then stored in data structures internal to the simulator C++ classes.

Some of the model information requires a preliminary elaboration, such as the execution time that is represented by parameters of a Weibull distributions: the lower bound (b), the upper bound (B), the mean (η), and the probability of having values greater than the upperbound (ρ). These parameters must be converted to compute the standard Weibull parameters: scale (λ) and shape (k).

The transformation has been performed considering that the cumulative distribution function (CDF), given an uniformly distributed random variable x, is null for $x < 0$, and for $x \geq 0$ is defined as $CDF(x) = 1 - e^{-(x/\lambda)^k}$. By considering that for $x = B - b$, it is possible to obtain $CDF(B-b) = 1 - \rho$, so after performing some substitution it is possible to define $\lambda = \frac{\sqrt[k]{-\ln\rho}}{B-b}$. The mean value of a Weibull distribution is calculated as $\eta = \lambda\Gamma\left(1 + \frac{1}{k}\right)$, and, by substituting the first result in the second equation, the result is $\frac{\sqrt[k]{-\ln\rho}}{B-b}\Gamma\left(1+\frac{1}{k}\right) - \eta = 0$.

The approach followed by the simulator described in this chapter to obtain an approximation of the k parameter is to minimize the absolute error of the previously described function

$$\min_{k>0}\left|\frac{\sqrt[k]{-\ln\rho}}{B-b}\Gamma\left(1+\frac{1}{k}\right) - \eta\right|. \tag{3.18}$$

The function minimum is obtained by using the GNU Scientific Library [90].

Cores, Kernels and Schedulers

In RTSIM the main entity for scheduling simulations is the Kernel. Each Kernel has an associated Core. Once a Kernel is instantiated, the programmer assigns a Scheduler to it. Among the different available schedulers, the one used for the challenge is the fixed priority scheduler. In RTSIM, partitioned multicore scheduling is obtained by instantiating multiple Kernel objects, one for each core.

Tasks

Each task τ_i in RTSIM is defined by its parameters: the activation time of first job ($a_{i,0}$), its relative deadline (D_i), its period or minimum inter-arrival time (T_i), and the sequence of instructions it executes, each defined by an execution time specification (deterministic or random). For any periodic task, the activation time of each job is computed by adding T_i to its last activation time. For sporadic tasks, a random value in the range $[T_i, T_i^{max}]$ is added to the last activation time, where T_i^{max} denotes the maximum inter-arrival time of τ_i. Relative deadlines are set equal to T_i. As for the job instructions, each task executes a sequence of runnables. According to the RTSIM syntax, the new instruction `runnable(runnableName)` is defined, and the code of each task is of the form:

```
runnable(r1);runnable(r2); ...  runnable(rN);
```

Runnables

Cooperative tasks preempt lower priority cooperative tasks only at runnable borders, while higher priority preemptive task can preempt any lower priority task and runnable. In the case of cooperative tasks, preemption within runnables is prevented by locking and unlocking a core-specific mutex dedicated to cooperative tasks before and after calling a runnable. The resulting job code for a cooperative task is:

```
...  lock(muxC);runnable(rX);unlock(muxC); ...
```

When a job calls a runnable instruction, the operations performed, in order, are the following: updating end-to-end statistics associated to labels reading events, virtually executing the runnable computations, updating end-to-end statistics associated to labels writing events.

Results

All the simulation runs performed for the challenge system produced the following: (i) complete traces of the task scheduling events; (ii) F2F and L2L end-to-end delays of each chain; (iii) response times of all runnables involved in each chain. The system simulation was performed collecting sample runs for different initial offsets of the tasks. For periodic tasks, the initial offsets are uniformly selected in the interval $[0, T_i]$, while for sporadic tasks they are chosen in $[0, T_i^{max}]$. The execution of the tasks has been simulated for a total virtual time of one hour. The simulation required 28 minutes and 50 seconds on a system with an *Intel i7-2630QM* core running at 2 GHz and 8 GB of DDR3 RAM running at 1333 MHz.

Figure 3.4 represents the distribution of the F2F and L2L latencies for each chain. For the first chain, the maximum end-to-end delays measured by simulation are 22377 μs for F2F and 12377 μs for L2L. For the second chain, the maximum end-to-end delays measured by simulation are 109.26 ms for F2F and 107.26 ms for L2L. In the end, the simulation returns 61324 μs for F2F and 3139.4 μs for L2L as maximum end-to-end delays for the third chain.

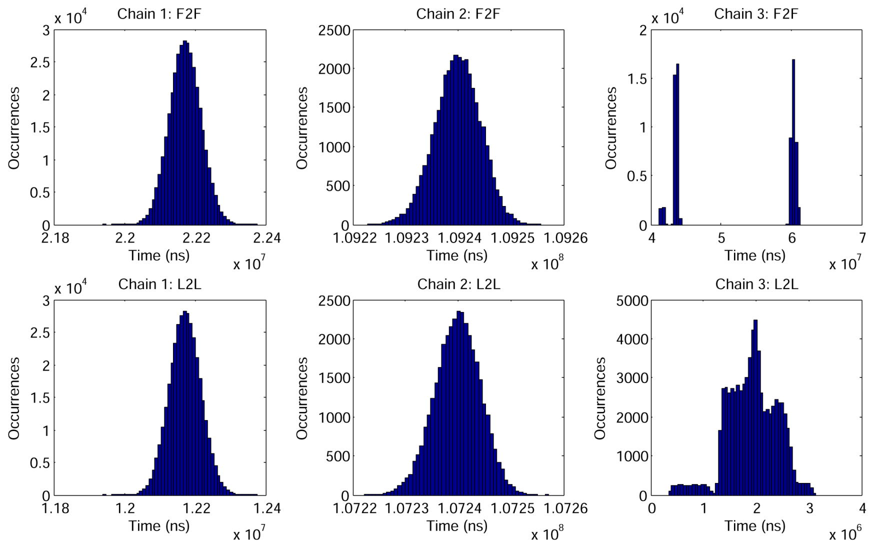


Figure 3.4: End-to-end delays obtained by simulation.

Additionally, Table 3.2 establishes a comparison between the worst-case response times of the runnables by the worst-case latency analysis of Section 3.1.2 (WCRT), which takes into account the scaling factors previously computed to guarantee schedulability, and the maximum response times observed during the simulations (SIM).

Runnable	**WCRT I**	**WCRT II**	**SIM I**
$R_{10ms,149}$	5176	5144	3556
$R_{10ms,243}$	7919	7903	5431
$R_{10ms,272}$	8896	8879	6139
$R_{10ms,107}$	3376	3383	2377
$R_{100ms,7}$	39647	39865	6992
$R_{10ms,19}$	770	781	577
$R_{2ms,8}$	142	150	122
$R_{700/800us,3}$	30	33	27
$R_{2ms,3}$	49	53	46
$R_{50ms,36}$	39074	39562	11151

Table 3.2: Worst-case response times (μsec) for the first (I) and second (II) challenge, and simulated results.

3.1.4 Conclusions and Open Challenges

This chapter proposed two solutions for the timing verification problem of the FMTV challenge. The first approach builds a mathematical model of the system and calculates worst-case latencies by adaptation of existing response time analysis techniques. Upper bounds on the end-to-end latencies are derived by first ignoring and then including memory access times. Then, a simulator of the given AUTOSAR model has been built on RTSIM to compute end-to-end latencies of the selected effect chains.

Using both analytical methods and simulation helps in the identification of the boundaries for the real worst-case response time: since the analytical methods are often pessimistic due to the assumptions made to simplify the system modeling, the returned values are likely upper bounds, while the simulation is not guaranteed to return the unlikely combination of events that generates the actual worst case response time.

While researching on this topic, we identified the importance of the memory access delay and the substantial impact of the data placement in the resulting end-to-end latencies. These latencies strictly depend on the hardware architecture, which is often not considered in the modeling of the system used for the analysis or simulation of the whole environment. This simplification is an open challenge not addressed in this work, which represents a significant gap for the applicability of the theory to real platforms. Section 3.2 goes more in-depth into this open challenge, addressing the data placement problem for a simple multicore hardware platform.

3.2 Data Placement Optimization to Minimize End-to-end Latency

Another relevant research activity of this thesis concerning computing architectures in automotive, as the follow-up of the problems solved in Section 3.1, has been related to the optimization of the data placement among the different memories of a multicore architecture.

The FMTV'17 challenge[4] represents an opportunity to go into detail with the open research problems of real automotive applications. In addition to the system presented in Section 3.1, FMTV'17 provides details of the hardware platform and imposes the use of the later-described Logical Execution Time (LET) model, aiming at investigating on the synthesis and optimization methods to exploit the available hardware resources and preserving the predictability of the software.

This section highlights some of the problems and issues that relate to the implementation of the LET model and then presents and compares two approaches for optimizing the placement of the labels in memory, including the time analysis methods that will be used for the system. The section concludes with a discussion on the next steps and other fundamental issues that are related to the general problem of optimizing the placement of computations in multicore platforms.

This challenge consists of a timing analysis problem in which the AUTOSAR model of a set of cooperating tasks in a fuel injection application is deployed onto a 4-core platform. The objective of the challenge is to study the possible conditions for the implementation of the Logical Execution Time (LET) model in the runnables communication and to provide methods for the analysis of the memory allocation of the communication variables (labels) in the model. The variables need to be allocated in the available memory spaces (local and global) of the platform.

The LET model was introduced as part of the Giotto programming paradigm [94] as a method to eliminate output jitter and provide time determinism in the code implementation of controls. In essence, LET delays the program output of a task (or any function executed inside the task) at the end of the task period, trading delay for output jitter.

The analysis of the LET implementation is performed under the assumption of the mechanisms and tools that are typical of an AUTOSAR process. In AUTOSAR, the computation functions are called runnables and the communication implementation is provided by a layer of code automatically generated by tools: the Run Time Environment (RTE). The consideration of the AUTOSAR process greatly influences the implementation options. The implementation of the LET model can follow two different approaches, that will be better described later: one that is compatible with the current mechanisms and tools of the standard AUTOSAR process, the other with a simple extension to the AUTOSAR implicit communication implementation (providing for a much more efficient solution).

Concerning the label placement optimization problem, real-time tasks may interfere with each other when accessing shared memory banks. This chapter also presents a simple method to bound the worst-case latency of those tasks. Using the provided bound for memory latency, two algorithms were developed to solve the problem: a simple Genetic Algorithm solution and a MILP formulation.

The results of these two optimization methods are provided with an additional discussion on how to tackle the runnable placement optimization problem, which is most likely the most relevant design issue for this kind of systems.

[4]More information available at: `https://waters2017.inria.fr/challenge/`.

3.2.1 System Model and Notation

The challenge model is similar to the one introduced in Section 3.1.1, and represents a case study from the Amalthea EU project, in large part compliant with the AUTOSAR metamodel. As such, the model adopts from AUTOSAR definitions and most of the semantics for the activation and communication of functions (runnables in AUTOSAR). An attempt at the formal characterization of the challenge model is the following, which recalls and extends what described in Section 3.1.1 for the current specific purpose.

Task and Runnable Model

A task τ_i is composed of an ordered sequence of n_i runnables $\rho_{i,1}, \dots, \rho_{i,n_i}$, each of which has its execution time $\mathcal{C}_{i,j}$, defined as a truncated Weibull distribution. For the purpose of worst-case analysis, the worst-case execution time (WCET) $C_{i,j}$ and a best-case execution time $c_{i,j}$ may be computed from the distribution $\mathcal{C}_{i,j}$. Each runnable $\rho_{i,j}$ may read or write labels from a set $\mathcal{L} = \{l_1, l_2, \dots, l_p\}$. Each label l_i is characterized by a type and a size (an integer number of bytes). Each task is defined by a tuple $\tau_i = \{\mathcal{C}_i, T_i, \mathcal{L}_i, D_i\}$, where $\mathcal{C}_i$ is the execution time distribution of the task, simply computed as the convolution of the distribution of the task runnables (by extension C_i and c_i are the worst and best case task execution times); T_i is the period or minimum inter-arrival time of the task activation event(s); $\mathcal{L}_i$ denotes the set of labels accessed by τ_i; and D_i is the relative (to the activation time) deadline. When applicable, relative deadlines are constrained to be smaller than or equal to periods, i.e., $D_i \leq T_i$. $N_{i,v}$ denotes the number of times τ_i accesses label $\ell_v \in \mathcal{L}_i$.

In the worst case (the reasoning also applies to other types of analysis but here only the worst-case analysis is discussed), the execution time C_i of a task may be expressed as the sum of the runnable execution times in the task. The execution times provided with the challenge do not include the execution cost to read and write the memory labels.

As in Section 3.1.1, the scheduling of each task is also controlled by its scheduling mode (cooperative or preemptive) and its priority π_i, with preemptive tasks having higher priority than cooperative tasks, and cooperative tasks only preempting each other at runnable boundaries. $R_{i,j}$ denotes the worst-case response time of the j-th runnable of task τ_i, while $r_{i,j}$ denotes its best-case response time. $hp^P(i)$ and $hp^C(i)$ denote the set of preemptive and cooperative tasks, respectively, having priority greater than τ_i, and $hp(i) = hp^P(i) \cup hp^C(i)$ denotes the union of the two disjoint sets.

Platform Model

The reference platform consists of $m = 4$ identical processors $P_1, \dots, P_m$, four local memories $M_1, \dots, M_m$ (one for each core), and a global memory M_{m+1}. The platform disposes of a crossbar switch that provided point-to-point communication channels between each core and each memory. Concurrent accesses to memory are arbitrated with a FIFO queue.

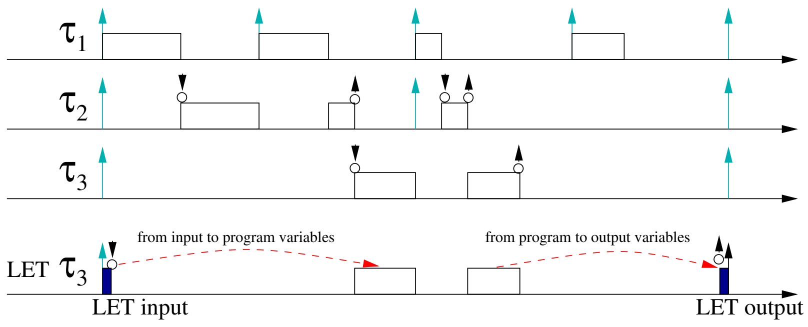

τ_1
τ_2
τ_3
LET τ_3
from input to program variables
from program to output variables
LET input
LET output

Figure 3.5: The LET model of execution.

Task and Label Allocation Model

The allocation of the tasks is *fixed* and given in the provided Amalthea model. $P(\tau_i)$ denotes the processor to which τ_i is allocated; $\Gamma(P_k)$ denotes the set of tasks allocated to processor P_k; and $\Gamma(\tau_i)$ denotes the set of tasks allocated to the same processor to which τ_i is allocated. An allocation of the labels is also provided in the Amalthea model. The following notation is used when discussing the label allocation. $\mathcal{M}_k$ denotes the set of labels allocated to memory M_k. λ^R denotes the delay introduced by task during a conflict to a remote memory, while λ^L denotes the delay introduced by task during a conflict to its local memory. The maximum time needed to access a word into memory M_x from processor P_x is denoted by

$$\Delta_{k,x} = \begin{cases} \delta^L & \text{if } k = x \text{ (local memory),} \\ \delta^R & \text{otherwise;} \end{cases}$$

where δ^R denotes the time needed to access a remote memory (GRAM or local RAM of another processor), and δ^L denotes the one needed to access the local memory M_k, with the assumptions $\lambda^L = \delta^L$ and $\lambda^R = \delta^R$. Based on the challenge information, the memory access and conflict times are $\lambda^L = 5$ ns (1 cycle); $\lambda^R = 45$ ns (9 cycles).

Finally, with respect to a given allocation of the labels, the time $MA_{i,v}$ needed to access label $\ell_v \in \mathcal{L}_i$ from task τ_i is defined as $MA_{i,v} = \Delta_{k,x}$ where $P_k = P(\tau_i)$ and M_x is the memory in which ℓ_v is allocated. The same terminology applies to runnables.

The LET Model of Execution

The Logical Execution Time model was probably first presented as part of the Giotto project [94]. The objective of the LET model is to add time determinism to periodic computations by eliminating the output jitter.

The LET execution model can be summarized as depicted in Figure 3.5. In the figure, the output of task τ_2 (denoted by the upward arrow at the end of the box representing the task execution) has a significant jitter. Because of variable interference from τ_1, it occurs late in the first task instance and much earlier in the second. The LET solution is shown in the bottom timeline for task τ_3 (taken as an example). The input of the task data is performed at the task activation, and the output is performed at the end of the task execution period. All task inputs are stored in local variables at the task activation. Similarly, all outputs need to be stored in local variables and will be actually output only by the LET code at the end of the cycle. This requires to allocate memory for local variables mirroring all input and output variables.

Several mechanisms can be used to enforce the LET synchronization of input and output operations, as a hardware or software implementation. In essence, LET is a sample and hold mechanism with synchronized execution of the input and output part. As such it is not too dissimilar to mechanisms used to enforce flow preservation in the implementation of synchronous models [191, 202]. When LET is implemented in SW (the hardware implementation would not affect the design analysis or the challenge goals) assuming a typical AUTOSAR development model (for more information on the related assumptions please refer to [72, 83]), there are two main options:

- LET is implemented as part of the Run-Time Environment (RTE) with support from the basic SW;
- LET is implemented at the application level by a set of dedicated runnables.

In both cases, since it requires a dedicated set of tasks (and the corresponding scheduling configuration), the LET implementation will most likely be modeled as a set of RTE or application-level input and output tasks. Since it is required that the input and output operations of these tasks are executed as close as possible to the start and end of period instants, these tasks should be characterized by a very short WCET and a very high priority level. This has several implications that are further discussed in the implementation section.

- There may be more than one task dedicated to the input and output sampling for LET execution. If this is the case, then these tasks will internally preempt each other and the design of this additional set of tasks may be a subproblem in its own.
- The execution of the output task (or action) at the end of the period may be very difficult to obtain with conventional scheduling strategies ("as late as possible execution" is typically not supported). In this case the output task needs to be actually executed at the beginning of the next cycle, possibly in conjunction with the corresponding input task (in a back-to-back fashion).

3.2.2 Timing Analysis with Memory Contention

This section presents a response-time analysis for tasks under partitioned fixed-priority scheduling that explicitly accounts for the delay introduced by *memory accesses* and their corresponding *memory contention.* The same analysis can be extended to runnables in a seamless manner.

Under the assumption of constrained deadlines, the worst-case response time of a task τ_i is bounded by the least positive fixed-point of the following recurrent equation:

$$R_i = W_i + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lceil \frac{R_i}{T_j} \right\rceil W_j + MC_i(R_i) \tag{3.19}$$

where $W_i = C_i + \sum_{\ell_v \in \mathcal{L}_i} N_{i,v} \cdot MA_{i,v}$ (i.e., the worst-case execution time of the task plus the cost for accessing its labels) and $MC_i(R_i)$ represents the delay due to memory contention incurred by τ_i and all the high-priority tasks, which *transitively* affects the response time of the task under analysis.

Since memory contention is resolved according to the FIFO policy, a safe bound on the term $MC_i(R_i)$ can be obtained by simply *inflating* the terms W_i to account for $m-1$ contentions for each memory access. However, this approach may lead to excessive pessimism, thus resulting in very coarse upper-bounds on the response times.

The *inflation-free analysis* [32, 192] is here used to bound the blocking times for a synchronization protocol for multiprocessor systems. This analysis explicitly accounts for each memory access that may originate a contention while task τ_i (under analysis) is pending. To this end, it is now possible to bound the maximum number of accesses $NRA_{k,x}(t)$ issued by tasks executing on the remote processors $P_k \neq P(\tau_i)$ to each memory M_x in an *arbitrary* time window of length t, that is:

$$NRA_{k,x}(t) = \sum_{\tau_j \in \Gamma(P_k)} \sum_{\ell_v \in \mathcal{L}_j \cap \mathcal{M}_x} \left\lceil \frac{t + R_j}{T_j} \right\rceil N_{j,v}. \tag{3.20}$$

Note that the above equation considers the sum over *all* the tasks allocated to P_k, as they can produce memory contention independently of their priority (FIFO arbitration). The term $\lceil (t + R_j)/T_j \rceil$ is a safe bound on the maximum number of jobs of $\tau_j \in \Gamma(P_k)$ in any time window of length t [32, 192].

Similarly, it is also possible to bound the number of accesses $NLA_{i,x}(t)$ issued by the local processor $P(\tau_i)$ to each memory M_x in a *busy-period* of length t where τ_i is pending, that is

$$NLA_{i,x}(t) = N_{i,v} + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \sum_{\ell_v \in \mathcal{L}_j \cap \mathcal{M}_x} \left\lceil \frac{t}{T_j} \right\rceil N_{j,v}. \tag{3.21}$$

Due to the FIFO arbitration and the fact that the memory accesses are non-interruptible, it follows that **(i)** each memory access issued by a remote processor can delay *at most* one access issued by the local processor and **(ii)** each access issued by the local processor can be delayed by *at most* one remote access *per processor*; hence the following bound holds:

$$MC_i(t) = \sum_{P_k \neq P(\tau_i)} \sum_{x=1}^{m+1} \min \{NRA_{k,x}(t), NLA_{i,x}(t)\} \cdot \Lambda_{k,x}, \tag{3.22}$$

where the term $\Lambda_{k,x}$ is provided to distinguish the delay introduced by the memory contentions as a function of each pair (P_k, M_x), and is defined as

$$\Lambda_{k,x} = \begin{cases} \lambda^L & \text{if } k = x \text{ (local conflict)}, \\ \lambda^R & \text{otherwise.} \end{cases}$$

Equation (3.22) can be used in Equation (3.19) to bound the response times of the tasks. The term $NRA_{i,k,x}(t)$ depends on the response time of the tasks allocated to the remote processors: this additional recursive dependency can be

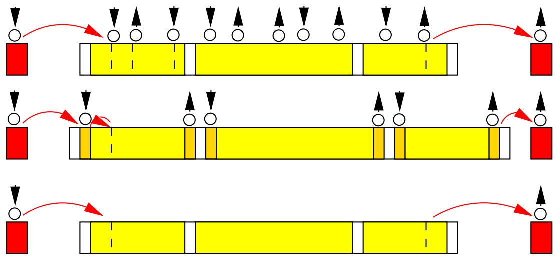


Figure 3.6: Code implementation of the LET model of execution with explicit or implicit communication.

addressed with an iterative loop in which Equation (3.19) is solved for all the tasks until *all* the response-time bounds R_i converge. Such an iterative loop starts with $R_i = C_i$ for all tasks τ_i.

3.2.3 Implementing and Analyzing the Logical Execution Model in AUTOSAR

The discussion on the implementation of the LET model cannot be undertaken without the joint consideration of the typical AUTOSAR model for the generation of task code and the execution of runnables. AUTOSAR has two models of communication. In the explicit model (top of Figure 3.6) the copy of the data in the communication variables is performed at the time each runnable invokes the communication API function. The implementation of the LET model in this case, would require the definition of two LET runnables that act as proxies for the read and write operations. The reader and writer runnable should execute according to the pattern defined in the following section.

In the implicit model, even if a read or write operation is invoked by the runnable in the middle of its execution, the actual code implementing the read from and write into the shared variables is automatically generated as part of the RTE code at the beginning and at the end of the runnable code. The result of the read operation is sampled at the beginning of the runnable execution and then stored in a local variable for the duration of the runnable execution. Similarly, the write value is locally stored in a variable and then output by RTE code after the runnable execution (shown in the middle of Figure 3.6, the darker rectangles before and after the runnable execution represent the RTE code). If the RTE generation tools are not modified, the LET implementation in this case would require yet another set of runnables and an additional set of variables, which is clearly a source of additional memory and time overhead.

However, it is relatively straightforward to see how a simple modification of the RTE generation process for the implicit communication model would be the best solution. A simple RTE generation option could result in moving the input and output to the LET tasks rather than the runnable boundaries. The RTE generator could generate the LET input and output tasks together with the other RTE-generated code.

Reader and Writer Tasks, Definition and Analysis

The general architecture and scheduling of the input and output tasks in a LET model is discussed at length in [95]. In their work, input and output tasks are scheduled together with mode change tasks assuming a time-triggered schedule with jitter constraints for the input and output operations.

In the case the tasks are implementing AUTOSAR runnables, the input and output tasks can serve all the tasks executing at the same rate. Of course, if a task has runnables executing at multiples of the task period, the corresponding input and output sections can be skipped when unnecessary. The input and output LET tasks may be scheduled using the AUTOSAR time-triggered mode when available, in order to ensure the output task is executed right before the end of the period of the tasks it serves. In a priority-based schedule, like the one assumed in the challenge, the output and input tasks may be joined and executed back to back at the beginning of each period.

To arbitrate among the input and output operations for the tasks executing at different rates there are several options between two extremes: one is to have a single LET task executing at the greatest common divisor of the task periods (most likely inefficient for the challenge model). The other extreme is to have a LET task for each period. Of course, partial groupings may be possible and may be more efficient in some cases.

A LET task for each period is assumed for this model. LET tasks have higher priorities than the other tasks (to enforce the precedence constraints) and their relative priorities are assigned according to Rate-Monotonic.

3.2.4 The Challenge Model

The provided challenge model has a set of special characteristics that affect the analysis and optimization methods and strongly characterize the obtained results.

First and foremost, the tasks are allocated on the cores according to a specific strategy. The first core only executes interrupt service routines. The same is true for the fourth core, that also executes a 10ms periodic task. The second core only executes (most likely details are not provided) a variable rate task (triggered at predefined angles in the engine rotation), and a very high rate 1ms periodic task. All the other periodic tasks are executed by the third core.

Using the worst-case execution times provided in the model, the system is definitively in *overload* with the following per-core utilizations: 0.97, 1.336, 1.068 and 1.179. The large overload in the second core is attributed to the modeling strategy adopted for the `Angle_sync` task. That task can be deemed as an engine-triggered task with variable activation rate and speed-dependent adaptive behavior. The provided model most likely considers a minimum inter-arrival time for the maximum engine speed, and a WCET computed for the most time consuming operating mode. This is pessimistic and explains the overload. The explicit consideration of the *adaptive variable-rate* (AVR) task model [35] would improve the analysis precision.

To optimize the system configuration based on the worst case behavior, the feasibility must be restored and a suitable cost function must be defined. To restore feasibility, the mean execution time is considered in place of the worst-case.

As a cost function, the maximum normalized (with respect to the deadline) response time of the tasks is selected as in the function:

$$\mathcal{C} = \max_{\tau_i \in \Gamma(P_k), \forall P_k} \frac{R_i}{D_i}, \tag{3.23}$$

3.2.5 End-to-end Latency

The analysis provided in [35] is adopted to compute the end-to-end latency of the effect chains.

However, in order to consider the influence of sporadic computational activities, the best-case response time computation for a runnable $\rho_{i,j}$ must be corrected as follows:

$$r_{i,j} = \sum_{h=1}^{j} c_{i,h} + \sum_{k \in hp(i)} N_k^{act} c_k \tag{3.24}$$

where N_k^{act} is defined as:

$$N_k^{act} = \begin{cases} \lceil \frac{r_{i,j}}{T_k} \rceil - 1 & \text{for periodic tasks;} \\ 0 & \text{for sporadic tasks.} \end{cases} \tag{3.25}$$

The ISRs have been considered as sporadic tasks: this choice has been adopted because the maximum inter-arrival time of the ISRs provided in the Amalthea model seems too close to the minimum one.

In this chapter, the computation of end-to-end latencies is provided only for the case of explicit communication. The case for LET-based communication is straightforward (modulo some minimal interference caused by high-priority LET tasks).

Effect Chain 1 In the effect chain 1, all runnables belong to the same task (*Task_10ms*, allocated to core 3). As there is backward communication between the third and the fourth runnable, this adds a one cycle delay until the last datum is read. Therefore, the worst-case end-to-end latency of this effect chain by L2F can be computed as:

$$L_1^{L2F} = T_{10ms} + R_{10ms,107} \tag{3.26}$$

This result is valid also when considering the L2L semantics. As for the F2F semantics, the analysis needs to consider a one cycle delay for the first runnable, that is:

$$L_1^{F2F} = 2T_{10ms} + R_{10ms,107}. \tag{3.27}$$

Effect Chain 2 Runnables in this chain belong to different tasks with different rates. In this case, the end-to-end latency calculation should also consider the over-sampling effect between pairs of consecutive runnables. By the L2F semantics, is obtained:

$$\begin{aligned} L_2^{L2F} &= R_{100ms,7} + \min(T_{10ms} - r_{10ms,19}, T_{100ms}) \\ &+ R_{10ms,19} + \min(T_{2ms} - r_{2ms,8}, T_{10ms}) + R_{2ms,8} \end{aligned}$$

As for the F2F semantics, due to the over-sampling effect, there are no input overwritings, hence the end-to-end latency is simply given by:

$$L_2^{F2F} = L_2^{L2F} + T_{100ms}.$$

Finally, the end-to-end latency computation for the L2L semantics requires to verify Condition (8) from [35], for any pair of consecutive runnables.

$$\begin{aligned} L_2^{L2L} &= R_{100ms,7} + \widehat{n}_1 \cdot T_{10ms} - r_{10ms,19} + R_{10ms,19} \\ &+\widehat{n}_2 \cdot T_{2ms} - r_{2ms,8} + R_{2ms,8}. \end{aligned}$$

Effect Chain 3 Also in this case, runnables belong to different tasks with different rates. Task periods have increasing values, leading to an under-sampling effect.

By the L2F semantics is obtained:

$$\begin{aligned} L_3^{L2F} &= R_{700/800us,3} + \min(T_{2ms} - r_{2ms,3}, T_{700/800us}) + \\ &R_{2ms,3} + \min(T_{50ms} - r_{50ms,36}, T_{2ms}) + R_{50ms,36}\ \mu s \end{aligned}$$

Due to the sporadic nature of the first runnable, is assumed $T_{700/800us} = 800\ \mu s$ in order to maximize the latency.

The end-to-end latency by the F2F semantics requires to add one cycle delay with respect to L2F and to verify Condition (8) from [35] for any pair of consecutive runnables.

$$\begin{aligned} L_3^{F2F} &= T_{700/800us} + \overline{n}_1 \cdot T_{700/800us} + R_{700/800us,3} + \\ &\overline{n}_2 \cdot T_{2ms} + R_{2ms,3} + R_{50ms,36} = 75559\ \mu s. \end{aligned}$$

Finally, the end-to-end latency for the L2L semantics is equal to the L2F case, because no output is overwritten due to the under-sampling effect.

3.2.6 Optimizing the Placement of Memory Labels

This section discusses possible approaches to compute the optimal placement of label and label copies (for LET) in memory. Two possible solutions (MILP and Genetic Algorithm) are presented for the case of explicit communication and LET-based communication.

Genetic Algorithm

Due to the extremely large set of labels to be positioned, a metaheuristic has been chosen to find a sufficiently good solution. A Genetic Algorithm (GA) approach has been found to be the most suitable candidate for this problem. Hereafter the structure of the algorithm is briefly presented.

Using the common nomenclature for GAs, a possible label placement is defined as an *individual* $\mathcal{I}$. Each individual is encoded as an ordered string of 10000 RAM identifiers (*genes*), representing the position of each label in the memories. The set of individuals (called *population*) is firstly initialized randomly. At every step each individual is evaluate with a *fitness* function that is the cost function identified for the challenge: $F(\mathcal{I}) = \mathcal{C}(\mathcal{I})$ i.e., the maximum normalized response time among all tasks with the labels positioned as in $\mathcal{I}$.

At every iteration, the solutions are reordered by following their fitness values $F(\mathcal{I})$ (the smaller the better) and divided in three subsets: (i) reproductive survivors (*elite*), (ii) non-reproductive survivors, and (iii) extinct individuals. The next generation is created by selecting random couples of *parents* between the elite group, that generate new individuals using a *crossover* function: this strategy swaps randomly selected blocks of genes between the parents and save the resulting solutions as a new individuals. At the same time, extinct individuals are removed from the population.

The exploration of new individuals in the solution space is guaranteed by using also a certain number of *mutation* functions, which randomly change a limited (and casually chosen) number of genes in the population. Each function has different activation probabilities and consist in random changing label positions, moving labels from one memory to another one, and spreading labels from one memory to all the others. On the other hand, in order to maintain a sort of *elitism*, a (small) number of clones of the best solutions are copied in the next generation without mutating.

MILP Formulation

It is worth discussing two approximations that have been applied to the analysis of Section 3.2.2 in order to express the response-time bounds in a linear form to formulate the problem as a *mixed-integer linear program* (MILP).

First, instead of searching for the least positive fixed-point of Equation (3.19), the approximated response-time analysis proposed by Park and Park in [153] is adopted. Under rate-monotonic scheduling, the authors showed (with an experimental evaluation) that their approximation introduces an extremely limited error ($\leq 1\%$) with respect to the exact response-time analysis. By extending Theorem 4 in [153] to cope with the analysis presented in Section 3.2.2, the response time of a task τ_i (if schedulable with a constrained deadline) is bounded by:

$$R_i = \min_{t \in S_i} \left\{ r_i = W_i + \sum_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lceil \frac{t}{T_j} \right\rceil W_j + MC_i(t) : r_i \leq t \right\}, \tag{3.28}$$

where

$$S_i = \left\{ \bigcup_{\substack{\tau_j \in hp(\tau_i) \\ \tau_j \in \Gamma(\tau_i)}} \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j, T_i \right\}.$$

This approximation results very useful for encoding the response-time bound into a MILP as **(i)** it allows getting rid of the typical *integer* variables that are needed to model the term with the ceiling of Equation (3.19) (note that the terms in the set S_i are all *constants*, hence that term is in turn a constant); and **(ii)** it allows avoiding the need for *quadratic* constraints, which is implied by the fact that the terms W_j must be optimization variables (note that their values depend on the label placement).

Second, to avoid requiring additional integer variables, the term $NRA_{k,x}(t)$ of Equation (3.20) has been over-approximated by replacing R_j with D_j.

Finally, it is worth mentioning that the lower-bounds of the response times were used to reduce the number of MILP variables (and the corresponding constraints) that must be provided to encode Equation (3.28). The lower-bounds have been computed by accounting for one clock cycle for each access to a label, which corresponds to the best case where labels allocated in local memory and no contention is possible. Such bounds allow reducing the elements into the set S_i.

A binary variable $A_{v,x}$ has been provided to each couple of label ℓ_v and memory M_x, with the interpretation that $A_{v,x} = 1$ iff ℓ_v is allocated to M_x. Such variables have been constrained such that $\sum_{x=1}^{m+1} A_{v,x} = 1$ holds for each label $\ell_v \in \mathcal{L}$.

The actual worst-case execution time W_i of a task τ_i allocated to processor P_k can then be expressed with the following linear constraint:

$$W_i \geq C_i + \sum_{x=1}^{m+1} \sum_{\ell_v \in \mathcal{L}_i} N_{i,v} \cdot \Delta_{k,x} \cdot A_{v,x}.$$

The objective of the MILP formulation is to *minimize* Equation (3.23).

3.2.7 Experimental Evaluation

LET Model Implementation

For the purposes of this challenge, the LET model has been implemented only for the runnables involved in effect chains; the only jitter-sensitive parts of the system.

The effect chains are composed of 10 runnables and 7 labels. The approach proposed in this chapter for the LET implementation requires adding high-priority LET tasks dedicated to copying and writing data implied in the effect chain. Runnables belonging to the same task need only one collective task, thus only 5 LET tasks must be added to the system. As every label needs two local copies (one for reading one for writing), the total number of labels is increased by 14.

Optimal Label Placement: MILP Formulation

The MILP formulation has been solved with IBM CPLEX on a machine equipped with an 8-core processor Intel(R) Xeon(R) E5-2609v2 running at 2.50GHz. The solver is able to immediately found (very first iterations) a feasible solution for the label placement.

For the case of explicit communication, the optimal placement is computed in about 1 hour and 20 minutes, but the solver is able to provide a sub-optimal solution with a guaranteed gap to the optimum lower than 1% in *less than two minutes*. The value of the objective function for the optimal solution is 0.8505. Recomputing the objective function with the analysis presented in Section 3.2.2 returns 0.849555: this result confirmed the effectiveness of the approximation adopted in the MILP formulation. Using the label placement provided in the Amalthea model of the challenge, the objective function is 1.32634: hence, this solution provides an improvement that is larger than 35%.

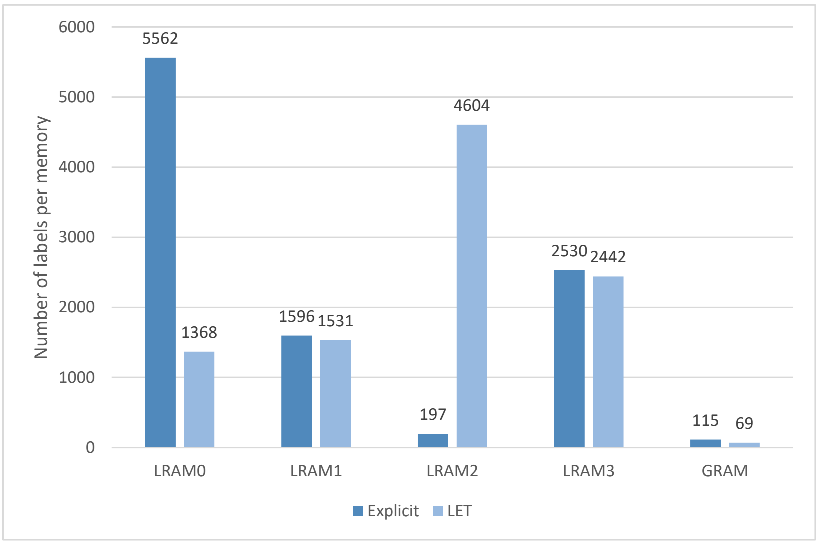


Figure 3.7: Placement of the labels for the case of explicit (dark bars) and LET-based (light bars) communication.

The label placement for the optimal solution is illustrated in Figure 3.7 (dark bars). As shown in the graph, most of the labels are allocated in the local memory of the first core (LRAM0).

For the case of LET communication, the optimal placement is computed in about 1 hour and 50 minutes. Similarly to the first case, the solver is able to provide a sub-optimal solution with a guaranteed gap lower than 1% in *less than seven minutes*. Using the label placement provided in the Amalthea model of the challenge, and placing the 14 labels required for implementing the LET communication as in the presented solution (as they do not exist in the challenge data), the objective function is the same for the case of explicit communication.

Surprisingly, the label placement is completely different from the one computed for the case of explicit communication (see Figure 3.7, light bars), as most of the labels are allocated in the local memory of core 2 (LRAM2). This result is attributed to the fact that the analysis is dominated by the `Angle_sync` task and the 1ms periodic task on core 1: as a consequence, the optimization algorithms will mostly optimize the labels used by these tasks, while the placement of the other labels is almost indifferent (with possibly few exceptions due to memory conflicts).

Optimal Label Placement: Genetic Algorithm

The Genetic Algorithm approach has been implemented in C++ and executed on an Intel(R) i7 4790K running at 4GHz. The population has been set to 200 individuals $\mathcal{I}$, all initialized randomly, and the termination condition of the algorithm has been defined to complete 30000 iterations. A feasible solution (with objective function < 1) is usually reached after the first 2000 iterations. The algorithm takes approximatively 5 seconds per iteration, with a completion time of less than 40 hours. For each communication semantic were produced approximately 20 distinct simulations.

For the case of explicit communication, the best results obtained with the Genetic Algorithm is an objective function of 0.85161, while for the LET communication the value is 0.85173. The solutions obtained are only slightly worse than the ones found with the MILP formulation, but required a large running time to be computed.

End-to-end Latency

End-to-end latencies of the effect chains have been computed using the optimal label placement that has been obtained with the MILP formulation. The best and worst case response times of the runnables under explicit communication are reported in Table 3.3, while the corresponding latencies of the effect chains are reported in Table 3.4.

Table 3.3: Response times (μsec) for the runnables in the effect chains under explicit communication.

Runnable name	**Best RT**	**Worst RT**	**Period**
Runnable10ms149	2077.68	4118.33	10000
Runnable10ms243	3315.99	6328.08	10000
Runnable10ms272	3644	7175.57	10000
Runnable10ms107	1367.16	2745.87	10000
Runnable100ms7	152.335	13820.5	100000
Runnable10ms19	298.715	650.74	10000
Runnable2ms8	55.465	117.12	2000
RunnableSporadic700us800us3	17.815	27.11	700
Runnable2ms3	20.025	42.385	2000
Runnable50ms36	1070.3	13089.6	50000

Table 3.4: End-to-end latencies for the effect chains under explicit communication.

Chain index	**Latency type**	**Latency value (μs)**
1	L2F	12746
1	F2F	22746
1	L2L	12746
2	L2F	26234
2	F2F	126234
2	L2L	134234 ($\widehat{n}_1 = 11$ and $\widehat{n}_2 = 5$)
3	L2F	15959
3	F2F	75559 ($\overline{n}_1 = 2$ and $\overline{n}_2 = 30$)
3	L2L	15959

Under LET-based communication, once a label placement that guarantees the task schedulability is found (as done by the proposed optimization algorithms), the end-to-end latencies can be computed in a straightforward manner. Further investigation may target the integration of the end-to-end latencies as constraints in the MILP formulation, with the objective of computing label placements that guarantee specific timing constraints related to the effect chains.

3.2.8 Discussion of Results, Conclusions and Open Challenges

This chapter presented the methods and algorithms to provide an AUTOSAR-compliant implementation of the *logical execution time* (LET) model and to optimize the placement of memory labels in the system.

The methods seem to be are applicable, perform quite satisfactorily, and can provide early indication on the quality of a given mapping configuration (or find a very effective one). However, there are several limitations in the challenge configuration that affect the results.

First and foremost, given the cost function and the current task set (and its core allocation), the analysis is dominated by the `Angle_sync` task and the 1ms periodic task on the same core. This means that the optimization algorithms will in practice mostly optimize the labels used by these tasks and be almost indifferent to the others (this explains why the LET solution and the solution without LET appear quite different with very similar cost values).

Also, this reinforces the concept, already illustrated in the Section 3.1, that the runnable and task placement dominates all other considerations and is therefore an interesting open challenge for future research. A discussion of possible formulations and the additional issues and problems can be found in [33].

3.3 Formal Language Verification

The increasing complexity of embedded systems requires an improved capability of detecting and fixing errors. The availability of a modeling environment like Simulink allows the verification by simulation or model checking of system properties and of the correct behavior of the design. This verification is possible upon condition that the requirements are expressed in a formal way.

Test and verification in Simulink is often a time-consuming process that requires the systems developers to translate requirements in model blocks for the verification. The capability of performing such translation is seldom available and prone to translation and interpretation errors.

This chapter presents a monitor generation tool and a Simulink library that enable a methodology to translate requirements in structured natural language into formal Signal Time Language (STL) constraints, leading to the automatic generation of Simulink monitors that check at run-time the desired properties. The tool automatically creates and connects the monitor blocks to a target Simulink model.

3.3.1 Introduction

Model-based development of embedded systems is today an established industrial practice. The use of models allows a precise and formal definition of the behavior with respect to time and also allows to raise the level of abstraction of the controller logic allowing verification by model checking and by simulation.

Simulink by the Mathworks [137] is among the most popular modeling environments. The Simulink models used for the representation of cyber-physical systems are based on a synchronous reactive semantics. The model of the controlled (physical) system is defined by a system of differential equations, integrated in continuous time, while the model of the controller is typically discrete-time.

Simulink allows for model verification of discrete-time models using formal proofs through the Simulink Design Verifier add-on (which is internally based on the Prover engine [160]) and supports checking assertions at simulation time using a simple library with basic assertion blocks.

This leaves the system developers with the task of bridging the gap between the requirements (often expressed in natural language) and the definition of monitors that check the requirements constraints at simulation time and possibly also at run-time. This process can be divided in steps. First, the requirements need to be translated from the natural language into a formal language. To ease this translation, the natural language can be constrained to be semantically as close as possible to a suitably selected formal language. Once the requirements are expressed formally, the language can be used to verify the correctness of the system model offline by model checking or theorem proving, or the constraint formulas can be parsed to automatically generate monitors that check them on-line (while the simulation is running) or off-line on the execution traces.

A temporal logic is a language in which formal specifications can be written for computer systems. In the late 70s, Amir Pnueli [158] introduced temporal logic to reason formally about the temporal behaviors of reactive systems. In the Linear Temporal Logic LTL [158] and the Computation Tree Logic or CTL [80], time is implicitly represented as an enumerated sequence of reaction steps occurring in a discrete time space. These temporal logics were developed to check properties in (typically hardware) systems with boolean, discrete-time signals and focused on the verification, specification, and synthesis of concurrent systems.

Other models and languages were later developed [109, 130] to improve the expressive power of LTL and CTL and to define and verify properties in real (continuous) time as applied to hybrid systems.

Today, there are several examples of temporal logic, differing in the model of time, the semantics of reactions and the language that can be used to define properties and constraints. The Property Specification Language PSL [79] is an extension of LTL in which constraints are composed of boolean expressions written in the host language (often VHDL or Verilog) together with temporal operators and sequences to describe the relationship between states over time. The Metric Temporal Logic (MTL) [109] allows reasoning over Boolean signals over dense-time domains and the Signal Temporal Logic (STL) [130] was proposed in the context of analog and mixed-signal circuits as a specification language for constraints on real-valued signals defined in continuous time.

The verification of timed properties using these languages has been studied in depth and so the possibility of using techniques for monitoring the properties off-line on system traces or on-line using monitors at simulation time. The general verification problem is discussed in several surveys and books such as [131].

Other works discuss the application of formal verification (by model checking) to systems with STL constraints. A recent work on this subject is [76].

A formal language like STL also offers the option of generating monitors for checking the properties of a simulated system. Offline techniques for monitoring STL properties on execution traces are discussed in [75]. This is an example of timed pattern matching, which consists in finding all segments of a continuous-time boolean signal that match a pattern defined by a timed regular expression. This problem has been formulated and solved in [185] via an offline algorithm that takes the signal and expression as inputs and produces the set of all matches.

Another possibility is the automatic generation of monitors that can be used to check properties at run-time, that is, while the simulation is running. In the context of timed regular expressions [14] an online matching algorithm has been presented in [186], but an on-line monitor generation technique is still not explicitly available for STL.

Finally, while the use of a formal temporal logic allows in principle the use of automatic verification techniques, bridging the gap between informal requirements and formal statements is not an easy task. Libraries and automatic implementation techniques can be used to ease the use of STL formulas in designs. In [102] Kapinsky et al propose the use of STL to verify typical control constraints in automotive applications modeled in Simulink, However, despite the title of the work, a library implementing the sample STL constraint presented in the chapter is not described, nor it is available.

As for the problem of translating informal requirements into formal (possibly STL) constraints, several approaches are possible. It is possible to parse the natural language to extract formal predicates (as in [84] or [87], with a more recent discussion of the possible approaches in [143]), or to restrict the natural language (using editors or forms) in such a way that only a readable form of formal statements (typically constructed by replacing the formal language operators with natural language tokens) is allowed. A comparison of the two approaches is presented in [68].

An example of controlled composition of natural language tokens is described also in [133], in which the requirements formulation approach is coupled with the proposal of a contract language for the expression of requirements.

Contributions The purpose of this work is not to provide a formal language or a formal extension of existing methods, but rather to provide a usable tool and library to improve the applicability of existing languages, methods and techniques.

This chapter presents an open source tool that generates Simulink monitor blocks for the validation at simulation time of a given models against constraints expressed according to a restriction of the STL language. The blocks are generated according to rules expressed as STL formulas and the monitor generation makes use of a set of library blocks that provide an implementation of the STL operators and an implementation of the typical control constraints described in [102].

Thanks to the availability of the source code and the modular structure of the project, the user can customize the tool and the library by directly accessing the software classes. It is thus possible to modify or improve the tool to extend

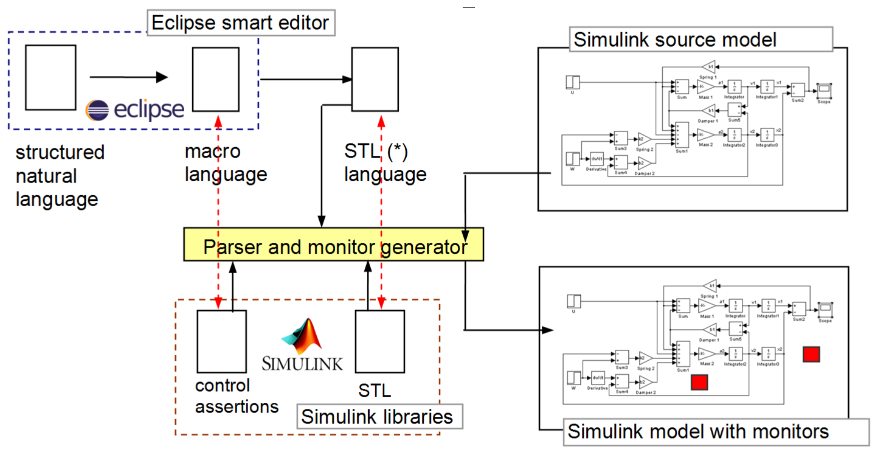


Figure 3.8: The framework for monitor generation from requirements.

the supported formal languages, or support other environments in addition to Simulink.

3.3.2 From Requirements to Monitors

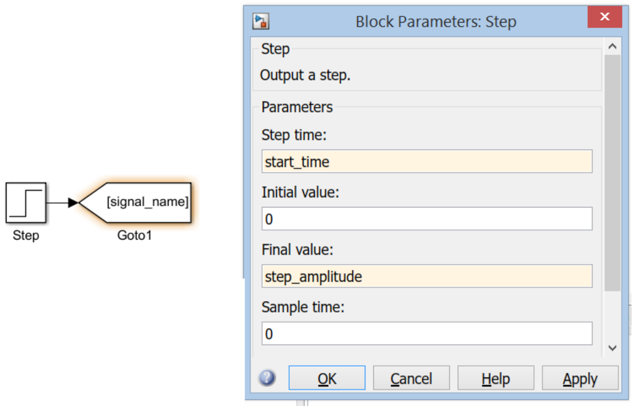
The work described in this chapter is part of a general framework that is meant to improve the quality of the requirements and automate their translation into runtime Simulink monitors. A graphical description of the methodology is shown in Figure 3.8 The framework is centered on the availability of STL specifications (actually using a restriction of the STL language) that express the constraints to be verified on the system. From the STL specifications, a tool automatically generates monitors that are automatically connected to the model signals to check the correct behavior of the system at simulation time. The monitors are generated using elements from a purposely developed Simulink library (freely available from [17]), that provides, among other things, a practical implementation of the library proposed in [102].

The following sections describe the methodology and the tools to generate the monitors from STL statements. However, this section provides a short description of the other stages of the process to provide some context to this work. These stages and tools (currently under development) are a first step to address the problem of bridging the gap from natural language requirements to the generation of monitors in Simulink.

To restrict the scope of the work to a manageable size, it initially targets typical control requirements, of the type presented in [102]. A typical requirement expressed in natural language (for a control application) is the following.

The Driver Subsystem (DRV) shall accelerate the motor from zero to x1 rpm in less than t1 sec, with an overshoot of less than x2 rpm.

A customized Eclipse editor that supports the user in writing structured requirement by separating assumptions from assertions has been developed. The editor provides syntax highlighting, context help and direct access to a library of symbols (of signals, subsystem and parameter identifiers) in an attempt to enforce the definition of requirements in a structured language with predefined



Block Parameters: Step

Step

Output a step.

Parameters

Step time: start_time

Initial value: 0

Final value: step_amplitude

Sample time: 0

Figure 3.9: Step signal generator.

natural language sentences or tokens, and the use of names from a data dictionary.

The informal requirement shown as an example then becomes.

R1. Acceleration bound and overshoot

Assumption:

The system inertia `sys_inertia` ***is less than or equal to*** `i1` ***and*** `reference` ***is a*** `step with amplitude` `A`.

Assertion:

Inside the Driver Subsystem (`DRV`), the speed (`spd`) signal `shall rise from 0 to x1 in less than t1` ***and*** `the overshoot shall be less than x2`.

In the new requirement formulation, the fixed size font indicates names of signals or parameters, the fixed size font in bold indicates macros; the italics bold indicates operators (logic and comparison) and the bold sans serif indicates a scoping operator. Finally, with limited additional reasoning or processing, the requirement can be rewritten using macros as in the following (DRV/spd indicates the signal with name spd defined inside the subsystem DRV).

R1. Acceleration bound and overshoot

Assumption:

`UPPERBOUND(sys_inertia, i1); STEP(reference, 0, A).`

Assertion:

`RISETIME(DRV/spd, 0, t1, 0, x1) and OVERSHOOT(DRV/spd, 0, x2).`

At this point the macros expressing the specification can either be translated into STL or, for simplicity, be directly transformed into signal generator or assertion checker blocks. For example, the signal generator macro

`STEP(signal_name, start_time, step_amplitude)`

could be implemented with the library source subsystem of Figure 3.9.

Similarly, the macro:

`OVERSHOOT(sig_name, start_time, oversh_bound)`

can be easily translated in STL or implemented using the library blocks described in Section 3.3.6.

3.3.3 The STL Language

In STL, a formula ϕ is evaluated on a sequence of inputs $\mathcal{X} = (x_1, x_2, \dots, x_n)$ at a (continuous) time instant t, resulting in the evaluation of $(\mathcal{X}, t)$ pairs.

An STL formula ϕ can be:

- p: a proposition that evaluates a state variable.
$$(\mathcal{X}, t) \models p \Leftrightarrow p\,[t] = \text{TRUE}.$$
- $\neg\phi$ (Negation): the logical negation of ϕ.
$$(\mathcal{X}, t) \models \neg\phi \Leftrightarrow \neg\left((\mathcal{X}, t) \models \phi\right).$$
- $\phi_1 \wedge \phi_2$ (And): the logical *and* between ϕ_1 and ϕ_2.
$$(\mathcal{X}, t) \models \phi_1 \wedge \phi_2 \Leftrightarrow (\mathcal{X}, t) \models \phi_1 \wedge (\mathcal{X}, t) \models \phi_2.$$
- $\bigcirc\phi$ (Next): a temporal operator that evaluates ϕ at the subsequent input value.
$$(\mathcal{X}, t) \models \bigcirc\phi \Leftrightarrow (\mathcal{X}, t+1) \models \phi.$$
- $\phi_1 \mathcal{U} \phi_2$ (Until): a temporal operator that is satisfied if ϕ_1 holds until ϕ_2 becomes true.
$$(\mathcal{X}, t) \models \phi_1 \mathcal{U} \phi_2 \Leftrightarrow$$
$$\exists t' \geq t : (\mathcal{X}, t') \models \phi_2 \wedge \forall t'' \in [t, t'), (\mathcal{X}, t'') \models \phi_1.$$

From the previous primitive operators, it is possible to derive other temporal operators:

- $\Diamond\phi = \text{TRUE } \mathcal{U}\phi$ (Eventually): the condition is verified at least once.
$$(\mathcal{X}, t) \models \Diamond\phi \Leftrightarrow \exists t' \geq t : (\mathcal{X}, t') \models \phi.$$
- $\Box\phi = \neg\left(\Diamond\neg\phi\right)$ (Globally): the condition is always verified.
$$(\mathcal{X}, t) \models \Box\phi \Leftrightarrow \forall t' \geq t : (\mathcal{X}, t') \models \phi.$$

In STL, temporal operators may be bounded inside an implicit $[0, +\infty)$ or explicitly specified time interval. The Until operator with an interval bound has the meaning

$$(\mathcal{X}, t) \models \phi_1 \mathcal{U}_{[a,b]} \phi_2 \Leftrightarrow$$
$$\exists t' \in [t+a, t+b] : (\mathcal{X}, t') \models \phi_2 \wedge \forall t'' \in [t, t'], (\mathcal{X}, t'') \models \phi_1,$$

from which is possible to obtain the following relations.

$$\Diamond_{[a,b]}\phi = \text{TRUE } \mathcal{U}_{[a,b]}\phi.$$

$$\Box_{[a,b]}\phi = \neg\left(\Diamond_{[a,b]}\neg\phi\right).$$

Language Implementation

Each STL formula or *STLFormula*, can be one of the following:

- *BoolExpr*: an expression resulting in a boolean value.
- *!STLFormula*: the logical negation of an *STLFormula*.
- `{` *STLFormula* `}` `AND` `{` *STLFormula* `}`: a logical *AND* operation between two *STLFormula*s.
- *STLUntil*: the *Until* temporal operator.
- *STLGlobally*: the *Globally* temporal operator.
- *STLEventually*: the *Eventually* temporal operator.

Temporal Operators. The STL temporal operators can be written in a text syntax that can be parsed.

The *Until* operator is expressed in the following ways:

- `{` *STLFormula* `}` `U_`*TimeExpr* `{` *STLFormula* `}`: timed *Until*.
- `{` *STLFormula* `}` `U` `{` *STLFormula* `}`: untimed *Until*.

On the other hand, the *Globally* and *Eventually* temporal operators, is expressed as follows:

- `[]` `{` *STLFormula* `}`: untimed *Globally*.
- `[]_`*TimeExpr* `{` *STLFormula* `}`: timed *Globally*.
- `<>` `{` *STLFormula* `}`: untimed *Eventually*.
- `<>_`*TimeExpr* `{` *STLFormula* `}`: timed *Eventually*.

Expressions. The previously mentioned *TimeExpr* defines the time interval in which the temporal operator is evaluated. It can be any kind of interval: closed [*Expr*,*Expr*], left open (...], right open [...), or open (...).

The *Expr* keyword identifies an expression with integer or floating point value:

BoolExpr is an expression with true or false evaluation, and can be one of the following,

- *Expr CmpOp Expr*: a comparison expression.
- *BoolExpr BoolOp BoolExpr*: a logical expression.
- *BoolFunction*: a function that returns a logical value.
- *BoolVal*: a constant logical value.

Operators. The operators recognized by the tool can be the basic mathematical, comparison, or boolean operators.

Values. *Val* or *BoolVal* represent values that can be either a variable defined by the user, the name of a signal or parameter belonging to the Simulink model or a constant value.

Functions. The language also allows using predefined functions such as `abs(` *Expr* `)` for the absolute value of *Expr*, `diff(` *Expr* `)` for the left-derivative of *Expr*, and `step(` *Expr* `,` *Expr* `)`, which returns TRUE when the first expression is recognized to be a step function with a height of at least the value defined by the second expression.
Timed Behaviors. In STL, timed formulas can be nested such as, for example:

```
<>_[0, T] { q AND []_[a, b] { p } }.
```

The proposition p is nested one level deeper than proposition q. The meaning is that there has to be one time instant t in $[0, T]$ (the outer Eventually condition) such that q is satisfied in t and for all the system evolutions starting from time t, the condition p is verified at some time between $t + a$ and $t + b$.

In a runtime monitor implementation, the evaluation of the global condition with p depends not only on the time range of its temporal operator, but also on the time t in which q is satisfied. If t_q is the time at which q is satisfied, the time range in which p is evaluated becomes $[a + t_q, b + t_q]$. The nested time interval $[a, b]$ is therefore not an absolute time, but is relative to the time instant identified by the outer clause.

Language Restriction

To generate online monitors, the following restrictions to the STL language are proposed:

- The maximum level of nesting for temporal operators is two.
- If there is a nested temporal operator, the condition on which the outer operator is evaluated must be a conjunction and at least one of the terms of the conjunction must be a proposition (not a temporal operator).
- If T_b is the maximum value for all the endpoints of the intervals defined in the inner (nested) temporal operators, then the terms of the conjunction that are not temporal operators can only be true at time instants that are separated by a time interval always greater than T_b.

For example, in:

```
<>_[0, T] { q AND []_[a, b] { p } },
```

the outer temporal operator is defined on the conjunction

```
q AND []_[a, b] { p }.
```

In order to correctly generate a monitor from this formula, the proposition q can only be true at time instants that are separated by more than b time units.

The purpose of the restrictions is to simplify (or make altogether possible) the online monitor definition and generation. However, despite these restrictions the language is still powerful enough to handle the typical control requirements defined for the library in [102].

Examples

The following examples show how it is possible to express some simple system constraints using the (restricted) STL language. The language is used to express the condition resulting in the violation of the constraint (and the corresponding activation of the assertion block).

```
/* Doors must never be open while the
 * elevator is moving */
<> {doorOpen == TRUE AND elevatorSpeed != 0} ;

/* The elevator must never exceed given
 * speed and acceleration limits */
<> {abs(elevatorSpeed) > maxSpeed OR
    abs(diff(elevatorSpeed)) > maxAccel} ;

/* If the elevator is called at the fourth
 * floor, it must reach the destination in
 * less than 100 time units */
<> {floorRequest == 4 AND
    []_[0,100] elevatorFloor != 4 } ;
```

Other examples of typical control specifications in STL can be taken from [102] and are used for the synthesis of the control monitor blocks described in Section 3.3.6.

3.3.4 The Monitor Generation Tool

The tool presented in this chapter consists of a MATLAB/Simulink front-end, implemented as scripts in the Matlab language, and an STL parser and Monitor code generator, implemented in C++.

As shown in Figure 3.10, the parser and generator tool takes as input two files, one containing the list of requirements and a Data Dictionary description containing (among others) information about all the subsystems, signals, and parameters defined in the requirements and having a corresponding definition in the Simulink model. The Data Dictionary file (currently a *.csv* Excel file) can be automatically synchronized with the definitions in the Simulink model by one of the Matlab scripts in the framework.

The tool parses the two files and outputs a new file containing the Matlab code that is used to generate the Simulink Monitor blocks for the runtime validation of the STL rules in the requirements.

This section provides a high-level description of the main tool subsystems and the input/output files by following the logical flow that the user follows to generate the monitors.

Data Dictionary File and Requirements File

The tool provides a Matlab function called `syncDD()`, that takes as input parameters the name of the Simulink model to be synchronized and the name of the Data Dictionary file. The function ensures that all the Simulink names of signals, subsystems and parameters are in the DD file and, if not, it updates the DD. The DD also contains the definition of all the constants (with values possibly computed as expressions of other constant values).

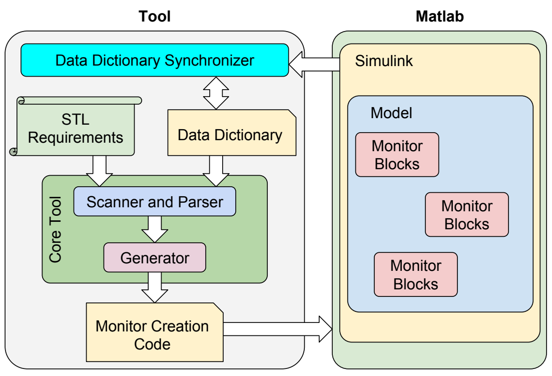


Figure 3.10: Block diagram representing the elements involved in the project.

The requirements file (from the editor or written manually by the user) is composed of a sequence of STL formulas to be monitored.

A label can be associated with each STL formula to ease the identification of the constraint that is checked by each monitor, as in:

```
maxExceeded : <> { x > maxValue };
```

Moreover, the tool accepts single-line and multi-line code comments expressed using the C language syntax.

Monitor Block Code Generator

When the STL requirements are parsed by the tool all the identifiers encountered in the requirements are checked to be valid constant values or signals as defined in the DD file.

When the the parsing of the requirements is completed, the tool generates in an output file the Matlab code containing the instructions to generate the Simulink monitor blocks.

Monitor Block Creation

The Matlab code generated by the parser is finally used to create the Simulink monitor blocks. The tool provides a Matlab function called `addMonitorBlock()`, which takes as parameters the name of the Simulink model in which the block will be added and the position where the monitor block is located.

The function creates the monitor block in the model and connects its inputs to the signals in the model using pairs of From/Goto blocks.

Model Validation

After the creation of the monitor blocks, the model can be validated by launching a Simulink simulation. In the default monitor creation process, each output port of a monitor is connected to an assertion block. Whenever a requirement is violated, the simulation is aborted and an error showing the violated condition is prompted to the user.

3.3.5 Parsing and Generation Tool

This section describes the implementation details of the subsystems described in Section 3.3.4.

STL Requirements Parser

The requirements file is first processed by the Flex (The Fast Lexical Analyzer) [114] Flex passes every STL language token detected in the source file to the syntax parser, implemented with the GNU Bison tool [114].

Constant values are computed and replaced, and the variable names and their values are stored in a (standard library) `map` data structure.

Each token recognizable by the parser has a corresponding C++ class, derived from a pure virtual `TreeNode` class that provides the following members and data:

- `left`, `right`: pointers to `TreeNode` classes.
- `generate()`: pure virtual function that creates the associated Matlab code.

When the Bison parser identifies a token, it creates an object from the C++ class representing the associated operator or expression and, if needed, sets the `left`, `right` (or both) data fields in order to create a binary tree of parsed objects.

Each class derived from `TreeNode` must provide an implementation of `generate()` (defined as pure virtual in the generic parent class). This function generates a Simulink block container that implements the clause expressed by the associated language token and then recursively calls `generate()` on its children nodes, creating the associated sub-blocks. The set of recursive calls at all the tree nodes, results in the generation of the Matlab code for the creation of the hierarchy of nested Simulink blocks inside the monitor.

The monitor block generated by the tool has a sub-block for each formula, as shown in the example of Figure 3.11. Those sub-blocks output a signal with boolean value representing the validity of the associated formula ($true/1$ when verified, $false/0$ otherwise). The output of the block is meant to be connected to the *Assertion* block provided by the Simulink standard library after being complemented by a NOT. The Assertion block takes as input a signal and, as default behavior, stops the simulation and prompt an error message when it receives a truth value. The block can be configured also to continue the simulation but signal the assertion violation with a prompt.

In order to make the timed temporal operators valid only in the time interval that is defined for them, they are provided with an additional boolean input port. The timed temporal operator is enabled only when this boolean value is true. This input port is connected with a time comparison block that outputs a true value only when the simulation time is in the given range.

The implementation of timed relationships between STL formulas is performed by extending the time range structure. Considering the Timed Behaviors description provided in Section 3.3.3, the time instant when the untimed terms in the inner conjunction (q in the example) are all satisfied is stored in a memory block and added to the blocks containing the interval edges. See for example, the implementation of the timed Until monitor of Figure 3.13.

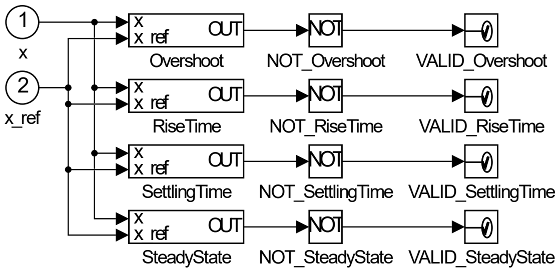


Figure 3.11: Validation bock content.

Simulink Functions

In Matlab, the signals and parameters defined inside the model can be extracted with the `getSignalsList()` and `getParameterList()` functions.

These function open the Simulink model passed as a parameter and scan it All the signals and parameters in the model are searched in the .csv DD file and, if missing, they are added to it.

Another important Matlab function provided by the tool is the one responsible for the creation and insertion of the monitor block in the Simulink model. The *AUTOGEN_testBlock.m* file is created by the parser tool, and used for the generation of the monitor blocks in the Simulink model by calling the `addValidationBlock()` function in Matlab.

The function takes as input parameter:

- The name of the Simulink model in which the validation block must be added.
- The name to be assigned to the validation block.
- The position of the validation block, expressed as the coordinates of the edges: left, top, right, bottom.

The function creates an empty block, with the requested position and name, as a monitor block container and runs the *AUTOGEN_testBlock.m* script to creates its content and the connections with the input model signals.

3.3.6 The Simulink Libraries for Monitoring STL and Control Constraints

To simplify the code generator, some of the standard functions that can be internally used by the validation block are developed as a Simulink library called STL Library and implemented in the file *STLLib.slx*, and a Control Monitor library in the file *CtrlMonitorLib.slx*.

STL Library

This library provides Simulink blocks implementing the STL temporal operators and the *AND* operator: *Eventually*, *Always*, *Until*, and *ANDSTL* (shown in Figure 3.12).

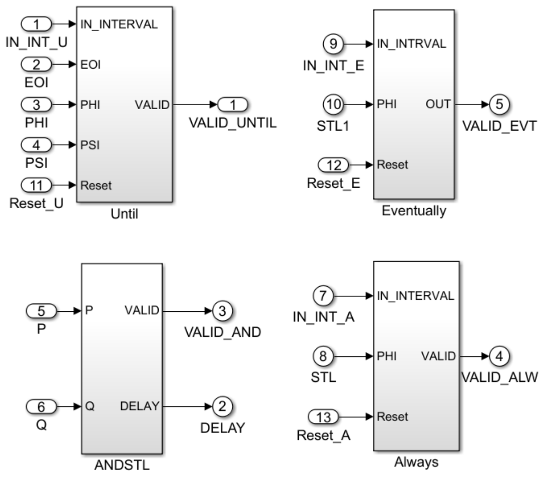


Figure 3.12: Library for the generation of STL monitors.

Consider, for example, the timed until block (labeled as UNTIL in the Figure, the other blocks follow similar conventions). The block contains an implementation of the clause $\phi U_{[SOI,EOI]}\psi$. The block inputs are: IN_INTERVAL that needs to be set to true if the current time is inside the interval $[SOI, EOI]$, false otherwise; the EOI value, the current evaluations for the ϕ and ψ formulas, and a RESET input.

Figure 3.13 shows the internals of the timed Until block. All the monitor blocks keep their output constant after a violation of the rule is detected. However, to facilitate their use in simulations concatenating several test cases, a reset input is also provided. This is implemented by the set-reset block at the end of the chain, on the far right.

The definition of the timed until $\phi U_{[SOI,EOI]}\psi$ is (from Section 3.3.3)

$$\begin{aligned}(\mathcal{X},t) \models \phi\mathcal{U}_{[SOI,EOI]}\psi \Leftrightarrow \\ &\exists t' \in [t+SOI, t+EOI] : (\mathcal{X},t') \models \psi\wedge \\ &\qquad\qquad \forall t'' \in [t,t'], (\mathcal{X},t'') \models \phi,\end{aligned}$$

with the availability of the signal IN_INTERVAL, the condition becomes

$$\begin{aligned}&\exists t' \text{ such that: IN\_INTERVAL} \wedge (\mathcal{X},t') \models \psi\wedge \\ &\qquad\qquad \forall t'' \in [t,t'], (\mathcal{X},t'') \models \phi,\end{aligned}$$

In the model implementation of Figure 3.13 the top left part is in charge of the implementation of the first conjunction (highlighted in red); whereas the bottom part (in blue) implements the final clause of the conjunction.

Control Monitor Library

To simplify the creation of monitors for typical control systems, is has also been defined a library of control monitors (shown in Figure 3.14.) The library

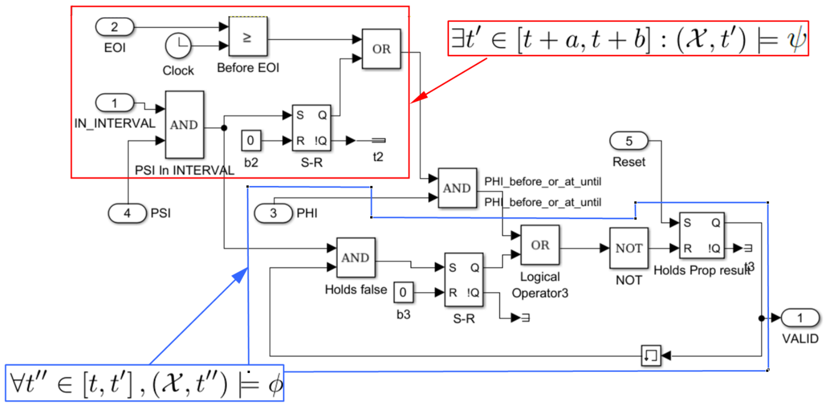


Figure 3.13: The STL library block for checking the timed Until condition.

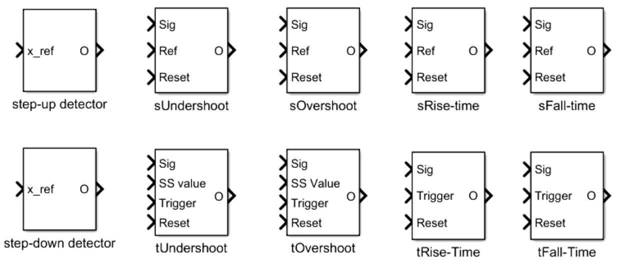


Figure 3.14: Library for the generation of Control monitors.

has blocks for checking overshoot (undershoot) and rise time (or fall time) constraints on triggers derived from generic inputs signals (steps, but also ramps). These conditions are verified on a selected input signal (typically a system variable or an input/output of the controller) with respect to another reference or trigger input. The library is constructed in layers. A set of blocks checks the conditions upon reception of a generic trigger signal. Other blocks are built on this set including the logic that detects the trigger from conditions on a generic signal.

Each block of the control monitor library (such as the overshoot block, shown in Figure 3.15) is built on top of (using) the STL library blocks.

For the implementation of the Control Monitor library, the STL formulations provided in [102] were used. For example, the STL encoding of the overshoot condition (with the corresponding block implementation of Figure 3.15) is:

$$\Diamond\_[0, T]\,(step(x_{ref}, r) \wedge \Diamond(x - x_{ref} > c))$$

3.3.7 Usage Example

This section presents a simple example model showing how the tool can be used to generate monitor blocks and what is the final result. The example system is

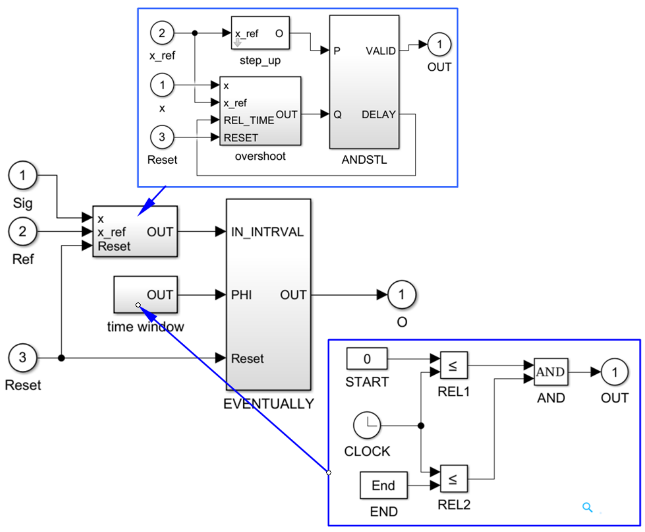


Figure 3.15: Overshoot monitor block internals as defined in the Control Monitor library.

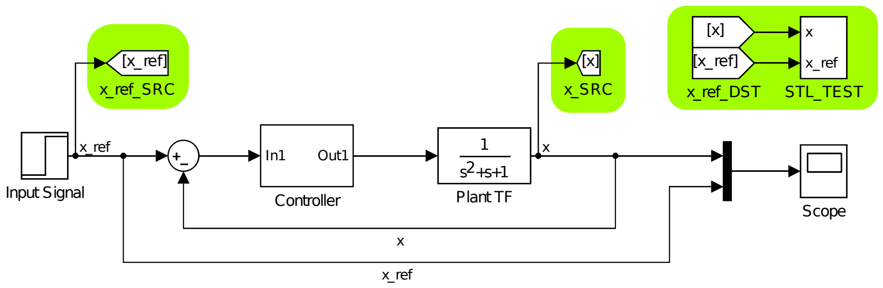


Figure 3.16: Example of Simulink block diagram of a model with a closed loop controller.

a Simulink model of a dual pole system controlled in a closed-loop, as shown in Figure 3.16.

The model is originally as shown in the bottom side, without the highlighted monitor blocks that are automatically added by the framework tool, as described in the next sections.

The function

```
>>> getSignalsList('SimulinkModelExample');
```

is executed from the Matlab prompt to verify that the signals in the model are contained in the DD file. The DD file also contains the constants are parameters used in the STL constraint formulas. The set of relevant variables and symbols in the DD file is shown in Table 3.5.

Listing 3.1 shows the example requirements file with the list of STL formulas representing the system requirements:

name	value
T	10
r	5
c	3
zeta	0.5
mu	0.95
steadyStateValue	10
s	3
beta	0.02
a	0.01

Table 3.5: Values in the DD file for the Simulink example with the STL constraints.

- *Overshoot*: after the detection of a step in the input signal, the output value of the system exceeds the reference value for more than a given quantity.
- *RiseTime*: after the detection of a step in the input signal, the system is not able to reach a specified value in a given time.
- *SettlingTime*: after the detection of a step in the input signal, the system is not able to keep the output bounded in a given range after a given time.
- *SteadyState*: when the system reaches its steady state condition, the value it outputs differs from the reference signal for more than a given value.

```
Overshoot : <>_[0, T] { step(x_ref, r) AND
  <> { x - x_ref > c } };

RiseTime : <>_[0, T] { step(x_ref, r) AND
  []_[0,zeta] { x < mu * steadyStateValue } };

SettlingTime : <>_[0, T] { step(x_ref, r) AND
  <>_[s, T] {abs(x - x_ref) > beta * x_ref} };

SteadyState : <>_[T, T] { abs(x - x_ref) > a};
```

Listing 3.1: Example of requirements file

The tool executes by passing as a first argument the requirements file and as a second argument the path of the folder containing the DD file.

After the execution of the tool, the *AUTOGEN_testBlock.m* file is created in the same path of the signals file.

To insert the validation block in the given Simulink model, the following function can be executed from the Matlab prompt:

```
>>> addValidationBlock('SimulinkModelExample', 'STL_TEST',
    [60,240,90,280]);
```

The result of the the `addValidationBlock()` function is the creation of the monitor blocks highlighted in Figure 3.10 and their connection to the specified input and output signals by means of From/Goto blocks..

3.3.8 Conclusions and Open Challenges

This chapter presented a framework for the generation of monitor Simulink blocks for model validation at simulation time. The STL formal language is used as reference for the definition of the model requirements. The chapter shows an overview of the tool and the supporting libraries, including implementation details and a practical example of its usage on a real model.

A still open challenge is to extend the tool by integrating it in a complete environment that supports the user to describe the model requirements in a formal language with a syntax closer to the natural languages.

Chapter 4

Extensions to Reservation-based Scheduling

In recent years, there has been a growing interest in supporting component-based software development of complex real-time embedded systems. Techniques such as machine virtualization have emerged as interesting mechanisms to enhance the security of these platforms, while real-time scheduling techniques have been proposed to guarantee temporal isolation of different virtualized components sharing the same physical resources. This combination also highlighted criticalities due to overheads introduced by hypervisors, particularly for low-end embedded devices. This led to the need of investigating deeper into solutions based on lightweight virtualization alternatives, such as containers.

In this context, Section 4.1 proposes the use a real-time deadline-based scheduling policy built into the Linux kernel to provide temporal scheduling guarantees to different co-located containers. The proposed solution extends the SCHED_DEADLINE scheduling policy to schedule Linux control groups[1], allowing user threads to be scheduled with fixed priorities (i.e., SCHED_FIFO or SCHED_RR) inside the cgroup scheduled by the SCHED_DEADLINE scheduling class. Since the proposed mechanism can be configured through the cgroups interface, it is also compatible with widely used tools such as LXC, Docker and similar, that are already using cgroups. This solution is compatible with existing hierarchical real-time scheduling analysis, and some experiments demonstrate consistency between theory and practice.

The advantages of this approach are presented in Section 4.2, showing preliminary results from on-going research for ensuring stable performance of co-located distributed cloud services in a resource-efficient way. It uses a hierarchical real-time CPU scheduling policy to achieve a fine-grain control of the temporal interference among real-time services running in co-located containers. The section also evaluates the performance of the presented solution by applying the method to a synthetic task set running on Linux within LXC containers. The Linux kernel used in the experiments has been modified to implement the

[1]The control groups (`cgroups`) infrastructure provides mechanisms for aggregating/partitioning sets of tasks into hierarchical groups with specialized behaviour, e.g., cgroups can be used to limit memory or CPU resources for defined sets of tasks. More information available at: `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`.

above mentioned hierarchical real-time scheduling policy.

The use of these scheduling paradigms has shown the limitations of classical task models, like the periodic one proposed by Liu&Layland, and it is pushing for the adoption of more realistic task models and the development of new schedulability analyses to guarantee their timing constraints. Self-suspending tasks are representative of enhanced task models considering explicit suspensions of the execution, happening when a task has to interact with an external device (e.g., through I/O operations) or to access shared resources. Real-time analysis of such a task model cannot neglect to take also into account temporal isolation techniques like bandwidth reservations and hypervisors, required to manage the complexity of actual software and the need of a modular development. The chapter concludes presenting in Section 4.3 the novel H-CBS-SO scheduling algorithm, which provides temporal isolation for real-time self-suspending tasks. The implementation of the algorithm is also proposed as an extension for the Linux kernel. Finally, experimental results are presented, aiming at evaluating the performance of the implementation in terms of run-time overhead.

4.1 Container-Based Real-time Scheduling in the Linux Kernel

Component-based software design and development is a well-known approach used in software engineering to address the complexity of the applications to be developed and to improve the software re-usability. Using this technique, complex software systems are decomposed into *components* described by well-defined interfaces; each component can be developed independently, and various components can be integrated later on the hardware target platform.

When developing real-time applications, the “traditional” software interface of each component must be complemented with non-functional attributes, e.g., those describing its timing requirements. In real-time literature, component-based design has been supported by modeling a component-based application as a scheduling hierarchy, as done for example in the Compositional Scheduling Framework (CSF) [26, 174, 175], that has been extended to support multiple CPUs and/or CPU cores in various ways [28, 29, 44, 78, 199].

In the past, these frameworks have been generally implemented by running each software component in a dedicated Virtual Machine (VM) and using a hypervisor scheduler as the root of the scheduling hierarchy [54, 110, 194]. In some situations, especially in embedded devices, it can be interesting to reduce the overhead introduced by a full VM, and to use a more lightweight virtualization technology such as container-based virtualization.

This section presents a CSF implementation based on containers; specifically, Linux control groups and namespaces. To support CSF analysis, this technology, on which many widely used programs such as Docker, LXC, LXD and similar tools are based, has been extended by implementing a theoretically sound 2-levels scheduling hierarchy. The SCHED_DEADLINE scheduling policy, implementing the CBS [6] algorithm, is used as the root of the scheduling hierarchy, and the standard fixed priority scheduler (SCHED_FIFO or SCHED_RR scheduling policy) is used at the second level of the hierarchy. The main difference with respect to previous implementations [49] is that the standard SCHED_DEADLINE code-base in the mainline Linux kernel is extended, rather than reimplementing an independent reservation-based scheduler.

The proposed approach is compatible with existing real-time analysis (some experiments show that the obtained results are compatible with MPR analysis [78], but different kinds of analysis and design techniques can be used as well) and with commonly used software tools, that do not need to be modified, for example an unmodified LXC installation has been used for the experiments.

4.1.1 Definitions and Background

As previously mentioned, isolation among software components can be achieved by running each component in a separate VM. By using appropriate scheduling techniques, it is possible not to limit isolation to spatial isolation, but to implement temporal isolation too, meaning that the worst case temporal behavior of a component is not affected by the other components[2].

Virtualization and Containers

The software components of a complex real-time application can be executed in different kinds of dedicated VMs (a VM per component), based on different virtualization technologies and providing different performance and degrees of isolation among components. For example, some VMs are based on *full hardware virtualization* whereas other VMs are based on *container-based virtualization* (or *OS-level virtualization*).

When full hardware virtualization is used, the VM software implements and emulates all the hardware details of a real machine (including I/O devices, etc.) and the guest OS running in the VM can execute as if it was running on real hardware. In theory, any unmodified OS can run in the VM without being aware of the virtualization details, but this is generally bad for performance and some kind of *para-virtualization* is generally used.

When OS-level virtualization is used instead, only the OS kernel services are virtualized. This is done by the host kernel, which uses virtualization of its services to provide isolation among different guests. This means that the kernel is shared between the host OS and the guest OSs; hence all the guests have to be based on the same hardware architecture and kernel. As an example, this technique allows for using the same hardware platform to run multiple Linux distributions based on the same kernel and it is often used to run multiple Linux applications/containers on the same server, like Docker. Every distribution will be isolated from the others, having the impression to be the only one running on the kernel.

The Linux kernel supports OS-level virtualization through *control groups* (also known as `cgroups`, a mechanism originally inspired by *resource containers* [20]), and *namespaces*, that can be used to isolate various kernel resources. Userspace tools like Docker or LXC are used to set-up the control groups and namespaces as appropriate to execute guest OSs or isolated applications inside them.

[2]While in clouds or large-scale servers temporal isolation is sometimes implemented by dedicating entire CPU cores to components, in embedded systems the number of available CPU cores is too small and this solution is often not appropriate.

Most of the previous implementations of the Compositional Scheduling Framework focused on full hardware virtualization, directly executing guest machine language instructions on the host CPUs to improve performance and relying on special CPU features to make this possible; more technically, to make the CPU fully virtualizable [159]. The software responsible for controlling the execution of guest code, the *hypervisor*, contains a scheduler that is responsible for selecting the VM to be executed and implements the root of the scheduling hierarchy. In previous works, this scheduler has been modified to support real-time resource allocation [49, 194].

When using container-based virtualization, on the other hand, the host kernel is responsible for scheduling all of the tasks contained in the various VMs, and implements the root of the scheduling hierarchy. Since the host scheduler can easily know if a guest is executing a real-time task (using the POSIX SCHED_FIFO or SCHED_RR policy) or not, it is easy to schedule only real-time tasks through a CPU reservation implemented by the SCHED_DEADLINE policy serving the VM. This is another advantage of container-based virtualization that will be discussed later.

Real-time Scheduling

A real-time application can be built by integrating a set of *components*, where each component $\mathcal{C}$ can be modeled as a set of real-time tasks $\mathcal{C} = \{\tau_1, ...\tau_n\}$. A real-time task τ_i is a stream of activations, or jobs, $J_{i,j}$ ($J_{i,j}$ is the j^{th} job of task τ_i) with job $J_{i,j}$ arriving (becoming ready for execution) at time $r_{i,j}$ and executing for a time $c_{i,j}$ before finishing at time $f_{i,j}$ (the finishing time $f_{i,j}$ depends on the arrival time $r_{i,j}$ and the scheduling decisions).

The Worst Case Execution Time (WCET) of task τ_i is defined as $C_i = \max_j\{c_{i,j}\}$, while the period (or minimum inter-arrival time) is defined as $T_i = r_{i,j+1} - r_{i,j}$ (or $T_i = \min_j\{r_{i,j+1} - r_{i,j}\}$).

Each job is also characterized by an *absolute deadline* $d_{i,j}$, representing a temporal constraint that is respected if $f_{i,j} \leq d_{i,j}$. The goal of a real-time scheduler is to select tasks for execution so that all the deadlines of all the jobs are respected.

As previously mentioned, a component $\mathcal{C}$ is executed in a dedicated VM having m virtual CPUs[3] π_k (with $0 \leq k < m$). When container-based virtualization is used, the host scheduler is responsible for:

1. selecting the virtual CPUs that execute at each time;
2. for each selected virtual CPU, select the task to be executed.

In this chapter, virtual CPUs are scheduled using a reservation-based algorithm: each virtual CPU π_k is assigned a *maximum budget* (or *runtime*) Q_k and a *reservation period* P_k, and is reserved an amount of time Q_k every P_k for execution. In more details, the CBS algorithm [6], as implemented in the SCHED_DEADLINE scheduling policy [112] is used.

[3]If full hardware virtualization is used, the number of virtual CPUs m can theoretically be larger than the number of physical CPUs M; if container-based virtualization is used, $m \leq M$.

Overview of the HCBS Algorithm

In this section, the rules of the Hard Constant Bandwidth Server (HCBS) [36, 136], the algorithm on which the SCHED_DEADLINE scheduler is implemented, are described in detail using its budget-based formulation and its fundamental properties are recalled.

The rules of a HCBS server with period P and maximum budget Q (bandwidth $\alpha = Q/P$) are summarized below. At any time t, the server is characterized by an absolute deadline $d(t)$ and a remaining budget $q(t)$. When a job executes, $q(t)$ is decreased accordingly. The HCBS server can be characterized by three states: Idle, Ready and Suspended.

Below is reported the algorithmic rules describing the HCBS server:

Definition 1 (HCBS Algorithm)**.**

- ***Rule 1*** *Initially, each server is Idle with* $q = 0$ *and* $d = 0$*.*
- ***Rule 2*** *When the HCBS server is Idle and a job arrives at time* t*, a replenishment time is computed as* $t_r = d(t) - q(t)/\alpha$*:*
  1. *if* $t < t_r$*, the server becomes Suspended and it remains suspended until time* t_r*. At time* t_r*, the server becomes Ready, the budget is replenished to* Q *and* $d \leftarrow t_r + P$*.*
  2. *otherwise, if the server becomes Ready, the budget is immediately replenished to* Q *and* $d \leftarrow t + P$*;*
- ***Rule 3*** *When* $q = 0$*, the server becomes Suspended and is suspended until time* d*. At time* d*, the server becomes Ready, the budget is replenished to* Q *and the deadline is postponed to* $d \leftarrow d + P$*.*
- ***Rule 4*** *When the server has no more pending workload it turns to the Idle state, holding the current values for both budget* q *and deadline* d*.*

To address the schedulability analysis of the HCBS server, the following result is recalled:

Theorem 1 (Theorem 1 in [136])**.** *Given a set of* n *HCBS servers* (Q_i, P_i)*,* $1 \le i \le n$*, all the servers are schedulable under EDF if and only if:*

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \le 1. \tag{4.1}$$

The HCBS server can be used to achieve *temporal isolation* among a set of real-time tasks.

Theorem 2 (Theorem 2 in [136])**.** *Given a set of* n *classical sporadic tasks having implicit deadline, each task is associated with a HCBS server* $\mathcal{S}_i$*. For each HCBS server* $\mathcal{S}_i$*, are defined* $Q_i = C_i$ *and* $P_i = T_i = D_i$*. Then, this set of tasks, each one executing upon a dedicated HCBS, is schedulable with EDF if and only if the test of Equation (4.1) holds.*

The advantage of associating each task to an HCBS server is enabling the protection of the system from execution overruns of the tasks. In other words, scheduling the system without the reservation servers, an execution overrun of a task can impact on the schedulability of the whole system, potentially allowing deadline misses on other tasks that are not exceeding their WCET. On the contrary, when each task executes upon a reservation server, all the overruns are protected by the budget exhaustion mechanism that stops the execution of the task. In this way, only the task that is experiencing an overrun is affected in terms of schedulability, while it is possible to guarantee the deadlines of the other tasks.

Proposition 1 (Isolation property of the HCBS)**.** *Given a set of n tasks, each one running upon a dedicated HCBS server as in Theorem 2, such that condition 4.1 holds, if a task experiences an execution overrun (i.e., it exceeds its WCET) then the schedulability of the other tasks will not be affected.*

4.1.2 Real-time for Linux Containers

The solution presented in this chapter is based on implementing a hierarchical scheduling system (such as the one described in Section 4.1.1) in a way that is compatible with the most commonly used container-based virtualization solutions. Unmodified LXC has been used for the experiments as an example.

A Hierarchical Scheduler for Linux

Independently from the used userspace tool, container-based virtualization in Linux is implemented by combining two different mechanisms: cgroups and namespaces; the userspace tools like LXC and Docker are only responsible for setting up the namespaces and cgroups needed to implement a VM.

Namespaces are used to isolate and virtualize system resources: a process executing in a namespace has the illusion to use a dedicated copy of the namespace resources, and cannot use nor see resources outside of the namespace. Hence, namespaces affect the visibility and accessibility of resources.

Control groups are used to organize the system processes in groups, and to limit, control or monitor the amount of resources used by these groups of processes. Hence, cgroups affect the resource scheduling and control. In particular, the real-time control group can be used to limit somehow and control the amount of time used by SCHED_FIFO and SCHED_RR tasks in the cgroup. Traditionally, it allows for associating a *runtime* and a *period* to the real-time tasks of the control group (hence, it could potentially be used for implementing a scheduling hierarchy), but its behavior is not well defined. Hence, the resulting scheduling hierarchy is not easy to analyze; for example, it implements a runtime "balancing" mechanism among CPUs[4], it uses a scheduling algorithm similar to the deferrable server [182] algorithm (but not identical, hence problematic for the analysis) and it has a strange behavior for hierarchies composed of more than two levels. Moreover, real-time tasks are migrated among CPUs without looking at the runtimes available on the various cores.

[4] For more information, refer to the `do_balance_runtime()` function in `https://github.com/torvalds/linux/blob/master/kernel/sched/rt.c`.

In the presented approach, the software interface of the real-time control groups is re-used, changing the scheduler implementation to fit in the CSF. A HCBS-based reservation-based algorithm, SCHED_DEADLINE, was already available in the mainline kernel, so that has been chosen to schedule the control groups. Linux implements global scheduling between available CPUs by using per-core ready task queues, named *runqueues*, and migrating tasks between runqueues when needed. For example, "push" and "pull" operations are used to enforce the global fixed priority invariant between fixed priority tasks, i.e., the M highest priority tasks are scheduled. For each CPU, there is a runqueue for non-real-time (SCHED_OTHER) tasks, one for fixed priority tasks (the real-time runqueue) and one for SCHED_DEADLINE tasks (the deadline runqueue). Each task descriptor contains some *scheduling entities* that are inserted in the runqueues: one for SCHED_OTHER, one for SCHED_FIFO/SCHED_RR and one for SCHED_DEADLINE, the deadline entity.

In this scheduler, a deadline entity is added to the real-time runqueue associated with every CPU in a control group. This way, the real-time runqueues of a control group can be scheduled by the SCHED_DEADLINE policy. When SCHED_DEADLINE schedules a deadline entity, if the entity is associated with a real-time runqueue, then the fixed priority scheduler is used to select its highest priority task. The usual migration mechanism based on push and pull operations is used to enforce the global fixed priority invariant, so the m highest priority tasks are scheduled.

The resulting real-time control group now implements a 2-levels scheduling hierarchy with a reservation-based root scheduler and a local scheduler based on fixed priorities: each control group is associated with a deadline entity per runqueue (that is, per CPU), and fixed priority tasks running inside the cgroup are scheduled only when the group's deadline entity has been scheduled by SCHED_DEADLINE. Each deadline entity associated with a real-time runqueue is strictly bound to a single physical CPU.

Currently, all the deadline entities of the cgroup (one per runqueue/CPU) have the same runtime and period, so that the original cgroup interface is preserved. However, this can be easily improved if needed (for example, using an "asymmetric distribution" of runtime and period in the cgroup cores can improve the schedulability of the cgroup real-time tasks [118, 199]). On the other hand, if deadline entities with different parameters are associated with the runqueues of a cgroup, then the resulting container is characterized by virtual CPUs having different speeds, so the push and pull operations need to be updated.

Notice that this container-based implementation of a hierarchical scheduler has an important advantage compared to full virtualization: when the runtime of a virtual CPU (vCPU) is finished and the vCPU is throttled, the scheduler can migrate a real-time task from that vCPU to others (using the "push" mechanism). With machine virtualization, instead, the guest OS kernel cannot know when a vCPU is throttled and its tasks must be migrated to other vCPUs; hence, it may throttle some real-time tasks while some other vCPUs still have an available runtime that is left unused.

As an example, consider a task set $\Gamma = \{\tau_1, \tau_2\}$, with $C_1 = 40ms$, $T_1 = 100ms$, $C_2 = 60ms$, $P_2 = 100ms$ running in a VM with 2 vCPUs having runtimes $Q_1 = Q_2 = 50ms$ and periods $P_1 = P_2 = 100ms$. Assume that τ_1 is scheduled on the first vCPU and τ_2 is scheduled on the second one. At time

$40ms$, the job of task τ_1 finishes, leaving $10ms$ of unused runtime on the first vCPU, and at time $50ms$ the job of task τ_2 has consumed all the runtime of the second vCPU, hence the vCPU is throttled. If the VM is implemented using traditional hardware virtualization, the guest scheduler has no way to know that migrating τ_2 on the first vCPU the job would still have some runtime to use, while using the container-based approach the host scheduler can push τ_2 from the second vCPU of the control group to the first one, allowing it to use the remaining $10ms$ of runtime and hence to finish in time.

All the control groups are associated with well-specified runtimes Q_i and periods P_i, and the SCHED_DEADLINE scheduling policy enforces that the real-time tasks of a cgroup cannot use a fraction of CPU time larger than Q_i/P_i. Hence, it has been decided to implement only a simple 2-levels scheduling hierarchy. Since the real-time control group interface allows to build deeper hierarchies, nesting cgroups inside other cgroups, this feature has been implemented by "flattening" deeper cgroup hierarchies: a cgroup with runtime Q_i and period P_i can contain children cgroups with runtimes Q^i_k and periods P^i_k only if $\sum_k Q^i_k/P^i_k \leq Q_i/P_i$. Every time a child cgroup is created, its utilization Q^i_k/P^i_k is subtracted from the parent's utilization Q_i/P_i (technically, the parent's runtime is decreased accordingly) and the child cgroup is associated with a deadline entity that is inserted in the "regular" SCHED_DEADLINE runqueue (the same runqueue where the parent's deadline entity is inserted). In this way, the temporal isolation properties and the reservation guarantees of each cgroup are preserved, while userspace tools can still create nested cgroups (for example, LXC does not create its cgroups in the root cgroup, but in a dedicated "LXC" cgroup).

Schedulability Analysis

The presented kernel modifications result in:

- 2 levels of scheduling (a root scheduler and a local scheduler);
- m vCPUs for each VM;
- reservation-based scheduling of the vCPUs (reservation-based root scheduler): each vCPU π_k is assigned a CPU reservation (Q_k, P_k);
- local scheduler based on fixed priorities.

This kind of scheduling hierarchies has already been widely studied in real-time literature; hence, there is no need to develop new analysis techniques, but previous work can be re-used. In particular, Section 4.1.3 will show that the presented implementation provides experimental results that are consistent with the Multiprocessor Periodic Resource (MPR) model analysis [78].

According to the MPR model, each VM is assigned a total runtime Θ every period Φ, to be allocated over at most m virtual CPUs; hence, the VM is associated with a multi-processor reservation (Θ, Φ, m) to be implemented as m CPU reservations. Using the presented implementation, this corresponds to using a reservation period $P_k = \Phi$ and a runtime $Q_k = \lceil \Theta/m \rceil$ for each virtual CPU π_k. Note that the original MPR paper only analyzed EDF-based local schedulers; however, the paper mentioned that it is possible to extend the analysis to fixed priorities. Such an extension is already implemented in the CARTS tool [157], that has been used to validate the experimental results.

Application-dependent Analysis

As discussed, existing CSF analysis can be applied to container-based hierarchical scheduler presented in this chapter, and Section 4.1.3 will show that the scheduler provides results consistent with the theory. However, this kind of analysis is often pessimistic, because it has to consider the worst-case arrival pattern for the tasks executing in the container. If more information about the structure of the containerized applications is known, then a less pessimistic application-dependent schedulability analysis can be used.

For example, consider a simple scenario where a pipeline of n audio processing tasks $\tau_1, \ldots, \tau_n$ is activated periodically to compute the audio buffer to be played back on the next period. The activation period of the pipeline also constitutes the end-to-end deadline D for the whole pipeline computations. This use-case is recurrent for example when using common component-based frameworks for audio processing, such as the JACK[5] low-latency audio infrastructure for Linux, where multiple audio filter or synthesis applications can be launched as separate processes, but their real-time audio processing threads are combined into a single arbitrarily complex direct acyclic graph of computations. For the sake of simplicity, in the following has been considered a simple sequential topology where the tasks $\tau_1, \ldots, \tau_n$ have to be activated one by one in the processing workflow, typical of having multiple cascaded filters, and the focus is on the simple single-processor case.

The traditional way to handle the timing constraints of the audio tasks in this scenario is the one to deploy the whole JACK processing workflow at real-time priority. However, in order to let the audio processing pipeline co-exist with other real-time components, one possibility is to use the SCHED_DEADLINE policy, that allows to associate a runtime Q_i and a period P_i with each thread of the pipeline: each task τ_i is scheduled with a CPU reservation (Q_i, P_i).

Therefore, one would apply a standard end-to-end deadline splitting technique, for example setting each intermediate deadline (and period) P_i proportional to the worst-case execution time C_i of task τ_i in the pipeline, while keeping the intermediate reservation budget Q_i equal to (or slightly larger than) C_i:

$$\begin{cases} Q_i &= C_i \\ P_i &= \frac{C_i}{\sum_{j=1}^{n} C_j} D. \end{cases} \tag{4.2}$$

This would ensure that each task τ_i in the pipeline gets C_i time units on the CPU within P_i, where all of the intermediate deadlines sum up to the end-to-end value D. The computational bandwidth U_{DL} needed for the whole pipeline is:

$$U_{DL} \quad = \quad \sum_{i=1}^{n} \frac{Q_i}{P_i} = \sum_{i=1}^{n} \frac{C_i}{\frac{C_i}{\sum_{j=1}^{n} C_j} D} = n \frac{\sum_{j=1}^{n} C_j}{D}. \tag{4.3}$$

Using the container-based hierarchical scheduler, instead, it is possible to configure the system with the same ease in terms of schedulability guarantees, but in a much less conservative way. Indeed, the pipeline tasks can simply be all attached to the same group reservation (scheduling the threads with real-time priorities and inserting them in a dedicated cgroup) with runtime equal to the

[5]More information available at: `http://www.jackaudio.org`.

overall pipeline processing time $Q = \sum_{j=1}^{n} C_j$, and a period equal to the end-to-end deadline constraint $P = D$, achieving an overall computational bandwidth U_{HCBS} of simply:

$$U_{HCBS} = \frac{Q}{P} = \frac{\sum_{j=1}^{n} C_j}{D} = \frac{U_{DL}}{n}, \tag{4.4}$$

namely n times smaller than needed when associating a CPU reservation for each task of the pipeline.

However, in the most straightforward set-up where the tasks in the pipeline execute strictly one after the other[6], it is clear that, even under SCHED_DEADLINE, the whole pipeline can execute with a much lower real-time bandwidth occupation. Indeed, once a task is finished, it blocks after unblocking the next task in the pipeline. As a consequence, only one of the real-time reservations is needed at each time, throughout each activation of the whole pipeline. Therefore, it is sufficient to have a single reservation occupying a bandwidth sufficient for hosting the computationally heaviest task (Q_i/P_i turns out to be a constant anyway, following the above deadline splitting technique in Equation (4.2)), and resetting its runtime and deadline parameters forward, following the $(Q_1, P_1), \ldots (Q_n, P_n)$ sequence, as the processing across tasks moves on. However, albeit convenient, such a "reservation hand-over" feature is not available in SCHED_DEADLINE, and, as shown below, its effect is actually achieved equivalently by the hierarchical scheduler here proposed.

4.1.3 Experimental Results

The hierarchical scheduler for Linux presented in this chapter has been evaluated through a set of experiments. The patchset used for these experiments is freely available online[7]. The experiments have been performed on different machines, with different kinds of CPUs including an Intel(R) Core(TM) i5-5200U and an Intel(R) Xeon(R) CPU E5-2640, achieving consistent results, with CPU frequency switching inhibited and Intel Turbo Boost disabled.

Real-time Schedulability Analysis

A first experiment has been designed to show one of the advantages of the proposed approach, with respect to traditional full hardware virtualization. To achieve this result, a container-based VM has been started using LXC. The VM has been configured with 4 virtual CPUs, with runtime $10ms$ and period $100ms$. When running a CPU-consuming task implemented as an empty `while()` loop in the VM, it is possible to notice that it consumes 10% of the CPU time *on each virtual CPU.* This happens because when the task consumes the whole runtime on a virtual CPU, the corresponding deadline entity is throttled, and the task is migrated to a different virtual CPU, where it is able to consume other $10ms$ of runtime.

[6] JACK can be configured to execute tasks in a real "pipelined" fashion, where multiple tasks are activated at the same time on different pieces of the overall audio buffer being processed.

[7] `https://github.com/lucabe72/LinuxPatches/tree/Hierarchical_CBS-patches`, applies to the `master` branch of the Linux `tip` repository (`git://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git`), as checked out at end of June 2018 (commit `f3a7e2346711`).

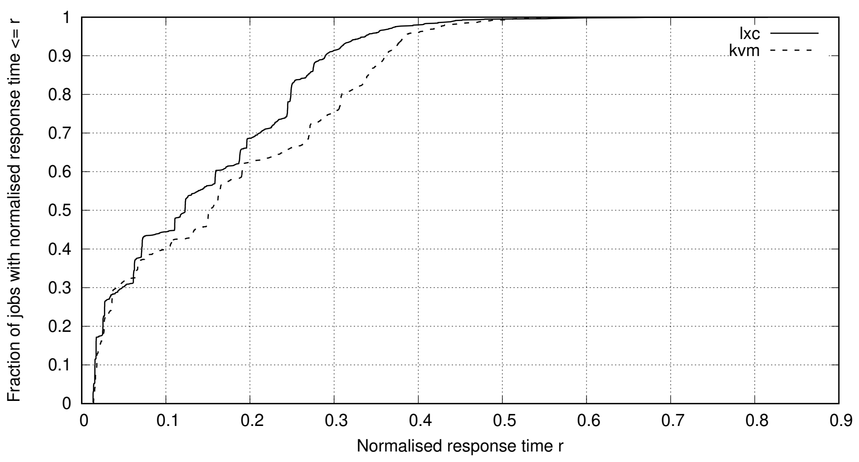


Figure 4.1: CDF of the normalised response times obtained using LXC and kvm. While the worst case response time is the same, LXC provides better average response times (the LXC CDF is above the kvm CDF).

This experiment shows that using the presented hierarchical scheduler the guest's real-time tasks can effectively consume all the runtime allocated to all the virtual CPUs of the VM. When using full hardware virtualization, instead, this is not possible. For verification, the same experiment has been repeated in a kvm-based VM[8] scheduling the 4 virtual CPU threads with SCHED_DEADLINE (runtime $10ms$ and period $100ms$ for each thread), verifying that the CPU-consuming task is able to consume only 10% of the CPU time *of one single virtual CPU*. This happens because when a virtual CPU thread consumes all the runtime, it is throttled, but the guest scheduler does not migrate the thread because the guest scheduler has no way to be notified when the virtual CPUs are throttled.

In the second experiment, the presented hierarchical scheduler has been verified to correctly implement CSF. Different components $\mathcal{C}$ with different utilizations $\sum_i \frac{C_i}{T_i}$ have been generated offline using the CARTS tool[9] to derive some (Θ, Φ, m) parameters that allow for scheduling the component tasks without missing deadlines. Then, the component has been executed in an LXC container (using a minimal Linux-based OS designed to allow reproducible experiments in virtualized environments), verifying that no deadline has been missed.

For example, the task set $\Gamma = \{(2, 32), (2.88, 40), (11.6, 46), (9.125, 48), (0.7555), (17.362), (14.5, 90), (2.6, 103), (4.5, 261), (67, 270), (34.3, 275), (4.45, 283), (8.55, 311), (47.4423), (97.4, 490)\}$ (where (C_i, T_i) represents a periodic task with WCET C_i and period T_i, in ms) is schedulable on 4 virtual CPUs with MPR parameters $(\Theta = 30ms, \Phi = 10ms, m = 4)$. When executing such a task set in an LXC container with 4 CPUs, runtime $7.5ms$ and period $10ms$, no deadline has been missed indeed.

The experimental Cumulative Distribution Function (CDF) of the normalized response times (where the normalized response time $r_{i,j}/T_i$ of a job is

[8]Other virtualization solutions use different kinds of hypervisors - for example, Xen uses a bare-metal hypervisor - but the results do not change.

[9]More information available at: `https://rtg.cis.upenn.edu/carts/`.

JACK clients	C_i	$290\mu s$
$(i = 1, 2)$	P_i	$1319.32\mu s$
`jackd`	C_3	$58.05\mu s$
	P_3	$263.86\mu s$
`Interference`	$C_4 = Q_4$	$6667\mu s$
	P_4	$16667\mu s$

Table 4.1: JACK experiment parameters

defined as the job's response time divided by the task period) is represented in Figure 4.1. For comparison, the figure also contains the experimental CDF obtained when running the same task set Γ in a kvm-based VM (with each one of the 4 virtual CPU threads scheduled by SCHED_DEADLINE, with runtime $7.5ms$ and period $10ms$). From the figure, it is possible to appreciate 3 different things. First of all, both the VMs are able to respect all the deadlines: this can be seen by verifying that both the curves arrive at probability 1 for values of the normalized response time ≤ 1 (if the normalized response time is smaller than 1, the deadline is not missed). Second, the worst-case experienced normalized response time is similar for the two VMs: the two curves arrive at probability 1 for similar values of the worst-case response time. This means that the worst-case behavior of the two VMs is similar, as expected. The last thing to be noticed is that the LXC curve is generally above the kvm one. This means that in the average case the response times of the container-based hierarchical scheduler are smaller than the one of the kvm-based one, because of the para-virtualized nature of the container-based scheduler (as previously mentioned): when a virtual CPU is throttled, the scheduler can migrate its tasks to a different virtual CPU that still has some runtime to be exploited.

Audio Pipeline

The third experiment shows the advantages of using the container-based hierarchical scheduler for the management of a real-time JACK audio processing workflow, with respect to the approach of isolating each activity in a separate CPU reservation by scheduling all the JACK threads as SCHED_DEADLINE. In this experiment, 2 JACK clients are sequentially chained. The first one takes as input stream the input data of the audio device, performs some computations and forwards it to the next client, until the last one. Actual audio I/O through ALSA is performed by `jackd`, the JACK audio server, which constitutes an implicit 3^{rd} element in the sample audio processing pipeline. JACK has been configured with a buffer size of 128 audio frames with a sample rate of 44100 frames/s, resulting in a total audio latency of $1000/44100 * 128 = 2.9025ms$ (corresponding to the period of the audio pipeline). The JACK clients performed a synthetic audio processing activity with nearly constant execution time. The overall JACK real-time workload added up to a 22% CPU load on the CPU. Also, an additional *interference load* has been added, as an additional 4^{th} real-time activity using a 40% CPU reservation. All real-time tasks have been pinned down on the same CPU, in order to reproduce the assumptions behind the calculations in Section 4.1.2. All the parameters for all real-time tasks and associated reservations have been summarised in Table 4.1.

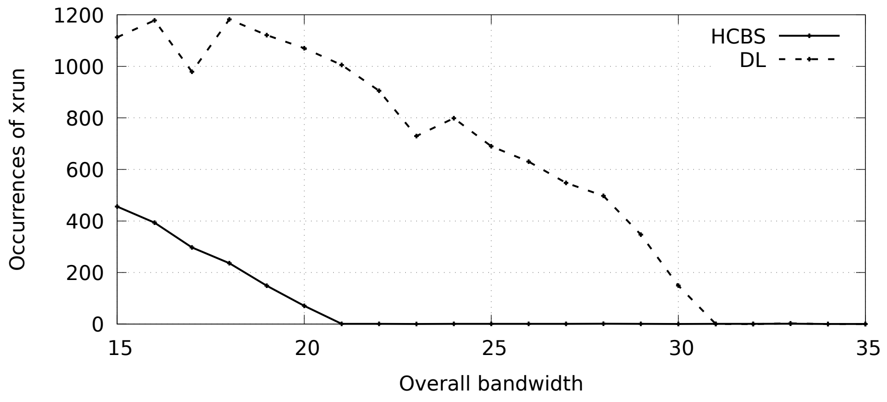


Figure 4.2: Number of xruns reported by JACK throughout all the 10 repetitions of each configuration. The proposed control group scheduler (HCBS) requires a smaller percentage of CPU time than standard SCHED_DEADLINE (DL) to avoid xruns.

The performance of JACK in terms of experienced xruns have been measured when the JACK threads are scheduled under the mainline SCHED_DEADLINE policy (indicated as "DL" in the figures) and when the hierarchical control group scheduler (indicated as "HCBS" in the figures) is used. First, the 3 real-time threads belonging to the JACK audio processing pipeline from the 2 JACK clients and `jackd` itself, have been scheduled with SCHED_DEADLINE using the parameters obtained from the deadline splitting approach in from Equation (4.2). As expected, using these parameters no xruns were experienced. Since Equation (4.2) can end up in an over-allocation of CPU time, the test has been repeated reducing the SCHED_DEADLINE runtimes. In practice, the Q_i parameters have been rescaled proportionally, so that the fraction of CPU time reserved to the three threads ranged from 15% to 35%. The DL line in Figure 4.2 reports the results, highlighting that the system starts behaving correctly when the percentage of CPU time reserved to the three JACK threads is about 31%. This is well below the conservative theoretical bound in Equation (4.3), as expected from the arguments at the end of Section 4.1.2.

After using the standard SCHED_DEADLINE policy to schedule the JACK threads, the test was repeated running JACK within an LXC container, so the whole set of JACK real-time threads have been scheduled within a hierarchical reservation. The period of the reservation was set to $2.9025ms$, and runtime varied around the ideal value $Q = \sum_{i=1}^{3} C_i = 638.05\mu s$ (21.98% of real-time bandwidth utilization), so as to match a real-time utilization from 15% to 35%. As visible from the HCBS line in Figure 4.2, there are no xruns for a real-time utilization of 21%, closely matching theoretical expectations from Equation (4.4). This value is significantly below the experimental threshold found above when using the original SCHED_DEADLINE.

Therefore, experimental results confirm the increased ease of use and efficiency in the use of the real-time HCBS computational bandwidth when using the proposed control group scheduler, compared to the original SCHED_DEADLINE scheduler.

4.1.4 Conclusions and Open Challenges

This chapter presented a new hierarchical scheduler implementation for the Linux kernel, designed to support container-based virtualization so that it fits in the CSF, that can be conveniently used with LXC containers having multiple virtual CPUs. The presented scheduler is implemented by modifying the real-time control groups mechanism so that the SCHED_DEADLINE policy is used underneath to schedule the real-time runqueues of each cgroup.

Experimental results show that a real-time application scheduled within an LXC container under the new scheduler behaves as predicted by existing theoretical CSF analysis. Also, in a realistic use-case involving the use of reservation-based CPU scheduling for guaranteeing CPU time to an audio processing workflow, the proposed control groups scheduler proved to be easier to configure and achieved better results, with the potential of occupying a lower real-time computational bandwidth within the system, for preventing the occurrence of xruns.

A challenge that still remains open is the investigation and analysis of the proposed scheduler in the context of parallel real-time activities deployed in multi-CPU containers.

4.2 Virtual Network Functions as Real-time Containers in Private Clouds

Information and communication technologies have undergone a relentless evolution in recent years, with a tremendous push towards distributed computing. The uprise of cloud computing, coupled with the widespread diffusion of broadband Internet connections, caused a paradigm shift towards more and more services provisioned through cloud computing infrastructures, in an on-demand, elastic, fashion, with needs of consumers evolving fast towards high-reliability, high-availability, and high-performance.

On the side of infrastructure providers, be it public cloud providers or private ones, there is an increasing interest in the efficient management of the hosted services. This brought to a number of innovations over the last years, ranging from features made available by hardware and CPU manufacturers in form of hardware-assisted virtualization mechanisms to reduce overheads due to machine virtualization, to software solutions based on tweaking internals of hosted operating systems and software stacks, pushing towards para-virtualization or very different solutions based on library operating systems or unikernels [101, 129].

Among others, network operators are looking with a growing interest in adopting (and adapting) the flexibility of (private) cloud computing to the provisioning of network functions as needed throughout their infrastructure. This is witnessed by the growing interest by, e.g., access network operators, in Network Function Virtualization (NFV) [142], a paradigm shift from traditional physical network appliances to network functions deployed as software components throughout a set of data centers. Thanks to the convergence towards IP-based networking, these functions can be managed with the flexibility and dynamicity of a private cloud, becoming effectively Virtual Network Functions (VNFs). These have a strong demand for locality, as these functions stay in the critical per-packet processing path and possess critical latency requirements, calling for solutions that have great efficiency in processing, among others. This is bringing an increasing interest in operating system (OS)-level virtualization techniques [20], i.e. Linux containers, as provided for example by LXC[10], LXD[11] or Docker[12], which exhibit basically no performance loss when compared to bare-metal deployments, still retaining the advantages in isolation, software stack organization and dependencies management typical of deployments using machine virtualization.

Deploying virtual machines (VMs) or containers in a shared infrastructure raises well-known problems of temporal interference among co-located components, particularly in presence of over-subscription of the CPUs, i.e., multiple virtual CPUs (vCPUs) mapped to the same physical CPU (pCPU) of a host, but also in presence of bottlenecks due to other resources, such as disks or networks, for data-intensive workloads. Focusing on the processing performance, controlling the interferences among co-located containers is typically done in nowadays clouds via a 1-to-1 vCPU-to-pCPU allocation, a.k.a., no CPU over-subscription, or dedicating entire physical machines to individual services.

However, this implies a minimum granularity in service allocation equal to the one of a single CPU computing power, with the consequence of a potentially high under-utilization of the infrastructure, in presence of deployed services with relatively small workloads. Therefore, an infrastructure provider is left with the choice between: a) deployments with some degree of over-subscription with potentially highly unstable performance of the individual service as due to the time-varying workload of others sharing the same pCPU(s), because there is little to no control of the interferences among them; b) deployments with no over-subscription, leading to stable performance of individual services at the cost of high under-utilization of the underlying physical infrastructure. Both options have additional drawbacks for a variety of application scenarios, including the considered NFV one (i.e., for the nodes at the edge of the network): a) over-subscription impacts on the QoS during traffic peak hours; b) 1-to-1 allocation unacceptably increases power consumption, which is already playing a dominant role in the overall energy budget of access networks.

This work presents the vision for tackling the problem mentioned above, based on a fine-grain allocation of pCPU(s): a real-time CPU scheduler in the OS kernel guarantees allocation of precise shares of a pCPU time to individual containers with a per-container time granularity, resulting in a stable and predictable performance of the hosted services.

The approach is validated by using a patched Linux kernel extending the mainline SCHED_DEADLINE CPU scheduler [112] with hierarchical scheduling capabilities, applied to scheduling containers hosting the later described synthetic application.

4.2.1 Related Work

Recent years have seen the growing success of the cloud paradigm outside of its original employment scenarios due to all its well-known advantages. However, those new application fields are characterized by additional constraints that

[10]More information available at: `https://linuxcontainers.org/lxc/introduction/`.
[11]More information available at: `https://linuxcontainers.org/lxd/introduction/`.
[12]More information available at: `https://docs.docker.com/`.

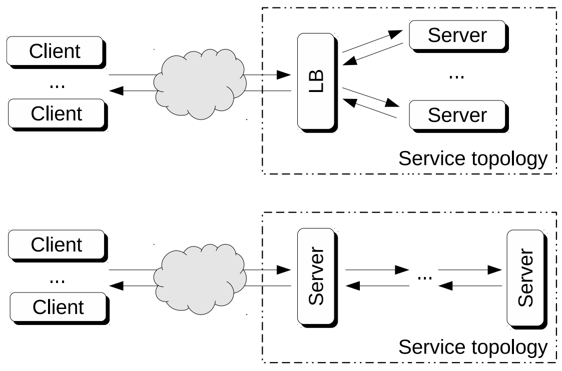


Figure 4.3: Reference service topologies.

cannot be handled without improving the modeling and the management of the underlying infrastructure, to exploit it more significantly [60].

Several works exist on controlling performance of distributed cloud services via elasticity and auto-scaling mechanisms [10, 166], intelligent placement strategies [106, 107], possibly including network-awareness [11] and SDN-based approaches [56]. To mitigate interferences of co-located services, real-time scheduling applied to hypervisors has been proposed, e.g., for the kvm [54] and Xen [110] hypervisors. In these works the hypervisor is extended to apply hierarchical real-time scheduling theory [175] and to allocate precise slices of the physical CPU execution time to each VM running on the platform. Some extensions to OpenStack [193] have also been proposed to deal with the experimental features introduced at the hypervisor level. The same Xen extensions have also been used for realizing a real-time NFV solution [115]. Note that, besides deployment of NFV with predictable processing latency, the presented technique can similarly be used for achieving predictable execution of a number of cloud workloads that need more and more end-to-end latency control in their execution, including real-time multimedia [55], cloud robotics [183], or cloud-based functions in automotive [108].

Some works addressed the problem of accelerating cloud infrastructures with support for heterogeneous hardware platforms, including GP-GPUs [200] and FPGAs [85]. This is orthogonal to the problem of limiting temporal interferences among co-located services, dealt with herein. To this end, a wide range of performance-related features is nowadays available in operating systems and hypervisors, that need to be exposed to higher cloud orchestration layers, that are currently active research projects [60].

4.2.2 Proposed Approach

The here used general reference system, illustrated in Figure 4.3, is composed of a number of clients submitting requests to a service topology including a number of servers either in charge of picking up a fraction of the incoming service traffic as due to the action of a load-balancer (shown in the top half of the figure), or being part of a group of replicated services, or to be traversed sequentially after one another (service chain, shown in the bottom half of the figure) or a combination of these elements.

In these cases, traditional performance control approaches in cloud computing prescribe the use of elastic scaling (typically horizontal, or, less frequently, vertical) coupled with load-balancing within cloud orchestration layers for controlling the overall service QoS. This has two major drawbacks: 1) the approach is viable only with services spanning across multiple instances; 2) if services are co-located on the same physical servers and physical CPUs, to make an efficient use of the infrastructure, then CPU contention causes the performance of each individual server to be highly unstable as due to changes in the workload of co-located services. Furthermore, elastic control loops tend to recover possible performance shortcomings a-posteriori, once the problem becomes evident through monitoring at the orchestrator sensing level, likely once clients have already been impacted by the performance degradation.

In the proposed approach, it is thus of paramount importance to have low-level mechanisms to keep the performance of individually hosted servers as stable as possible, even in cases of co-located functions on the same CPUs/cores. As it will become clear later, this is possible by recurring to special real-time scheduling of the CPU at the OS/kernel (or hypervisor) layer, allowing for temporal isolation among co-located containers. This mechanism can be nicely integrated with standard QoS control mechanisms for cloud services, making it easier to perform said control actions, thanks to the better stability of the performance of individual elements deployed within an elastically provisioned service. Furthermore, in the proposed approach it is possible to dynamically change the scheduling parameters, introducing an additional knob that can be used by an orchestration layer to fine-tune the CPU allocation to individual containers (vertical scalability), while achieving its control goals.

Therefore, in the following, the focus of this chapter narrows down to the problem of isolating the performance of individual co-located containers within a cloud platform, with particular reference to CPU scheduling, thus CPU-intensive services[13], as illustrated in Figure 4.4. The meaning of the per-container scheduling parameters (Q_i, P_i) will be clarified just below.

In the general context just highlighted, the presented on-going research is looking, among others, at the specifics of the NFV use-case, where a set of Virtual Network Functions (VNFs) are deployed as containers hosting packet processing servers (characterized by heterogeneous timing requirements, as due to different classes of handled traffic) across a number of possibly heterogeneous computing nodes.

Hierarchical Real-time Scheduling of Linux Containers

In what follows, the proposed modifications to the Linux real-time scheduler are described, with reference to how the technique has been applied to isolate execution of Linux containers. This way, it is possible to choose the configuration parameters for the server (Q, P) as a function of the desired QoS. That allows a resources controller to figure out the feasibility of a requested throughput based on the underlying resources and the already allocated services.

Linux containers, as created via the `lxc` tool, are associated with a *control group* (`cgroup`) allowing for the specification of *limits* on the amount of resources each container can use, including memory, physical CPUs, as well as limit to the

[13]For data-intensive services, the technique can be enriched by integrating additional QoS control mechanisms at the networking, disk or I/O layers.

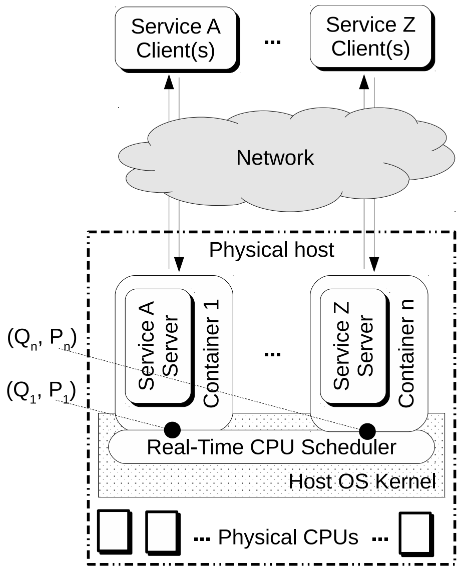


Figure 4.4: Proposed approach: n services are deployed as containers over a host with multiple physical CPUs. Container i has runtime Q_i and period P_i.

amount of time real-time tasks (`SCHED_FIFO` and `SCHED_RR`) within a container can be scheduled for. The Linux scheduler has been modified to allow building theoretically-sound scheduling hierarchies through `cgroups`[14].

The mainline Linux kernel has been recently added the `SCHED_DEADLINE` CPU scheduling class [112], a variant of the well-known EDF-based Constant Bandwidth Server (CBS) [6], allowing for attaching each task with a CPU reservation, expressed in terms of a `runtime` Q and a `period` P, with the meaning that Q time units are granted to the task on the CPU(s) every P time units.

The presented "Hierarchical CBS" (HCBS) scheduler[15] extends this mechanism with hierarchical scheduling concepts [175], realizing a mechanism where a CPU real-time reservation can be assigned to a control group as a whole, controlling the amount of time real-time tasks in each group are allowed to run on each CPU/core. This results in a 2-levels hierarchy of schedulers, where `SCHED_DEADLINE` selects the control group to be scheduled on each CPU, and the fixed priority real-time scheduler in the Linux kernel selects one of the tasks from the scheduled control group.

The resulting mechanism, conceptually similar to [49], supports partitioned scheduling in the host (each `SCHED_DEADLINE` entity used to schedule a control group is bound to a CPU/core) and generic affinities in the guest (fixed priority tasks in the control group can have generic affinities; hence both partitioned and global scheduling of real-time tasks are supported).

[14]The Linux kernel already provides hierarchical scheduling for real-time tasks, but its design aims only at acting as a limitation, not as a guarantee.

[15]The patch is available at: `https://github.com/lucabe72/LinuxPatches/tree/Hierarchical_CBS-patches`.

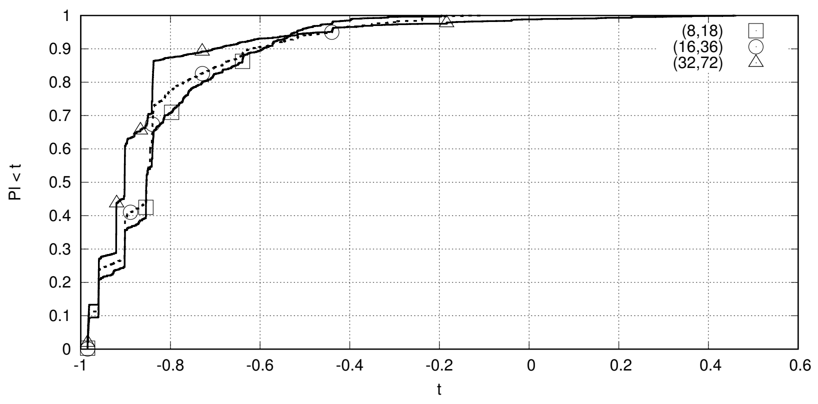


Figure 4.5: CDF for the normalized lateness of a task set scheduled in an `lxc` container with various values of runtime and period.

4.2.3 Experimental Results

This section presents some experiments with the approach presented in Section 4.2.2, using a Linux kernel v4.16.0-rc1, modified with the HCBS patch, and Linux containers through `lxc`, where were set per-container HCBS parameters (Q, P) as needed for each experiment.

Although the presented HCBS scheduler can support the stochastic analysis based on queueing theory (extending, for example, the analysis already performed for single tasks served by `SCHED_DEADLINE` [63]), it can also support hard real-time scheduling with guarantees provided through the Compositional Scheduling Framework (CSF) [110, 175].

To show this, the task set $\Gamma = \{(4879, 30000), (561, 36000), (10427, 104000), (4408, 109000), (20271, 250000)\}$ (where (C, T) indicates a periodic real-time task with Worst Case Execution Time C and period T; times are expressed in μs) has been scheduled by `SCHED_FIFO` and priorities assigned according to Rate Monotonic in an `lxc` container. The container has been assigned various combinations (Q, P) of runtime and period (according to CSF analysis, the task set is schedulable with $Q = 8ms$ and $P = 18ms$), and the resulting *normalized lateness* of the real-time tasks have been measured. The normalized lateness $l = \frac{r-T}{T}$ is defined as the difference between the *response time* r of a task and the task period T, divided by the task period (positive values indicate a missed deadline).

Figure 4.5 presents the Cumulative Distribution Function (CDF) of the normalized lateness measured for 3 combinations of scheduling parameters:

- $(Q = 8ms, P = 18ms)$: schedulable task set (no missed deadlines) according to CSF analysis. The CDF reaches 1 for values of the normalized lateness smaller than 0, so the theoretical results are confirmed
- $(Q = 16ms, P = 36ms)$: CSF analysis does not guarantee the schedulability of the task set, however the CDF reaches 1 for values of the normalized lateness smaller than 0 (no missed deadlines). This result does not contradict CSF analysis, that only provides a sufficient schedulability condition, and is quite pessimistic

- ($Q = 32ms, P = 72ms$): the scheduling system is stable according to queueing theory (the utilization of the task set $U = \sum \frac{C}{T} = 0.4$ is smaller than $32/72 = 0.44444$), but CSF analysis does not guarantee the schedulability of the task set. This is confirmed by the fact that the CDF reaches 1 for a normalized lateness equal to 0.46, so some deadlines are missed.

4.2.4 Conclusions and Open Challenges

This chapter introduced a feasible vision for deploying distributed cloud services with stable performance (with focus on NFV), based on containers and lightweight OS virtualization functionalities. Thanks to the used hierarchical real-time scheduler (which leverages existing theory in real-time literature), the proposed approach provides predictable QoS and can be used, for example, for QoS control in components in the context of 5G network function split. The mechanism ensures stable performance of deployed services, enabling the possibility to apply sound performance modeling, analysis and control techniques.

It would be interesting the challenging future activity of applying the presented architecture to real software components, for example, prototyping the mechanism within the OpenStack cloud management and orchestration framework, and use the OpenAirInterface[16] software as a case-study related to access network.

4.3 Self-suspending Tasks

Modern computation systems are characterized by a growing level of complexity due to an increasing number of cores and the availability of heterogeneous dedicated subsystems. To fully exploit this huge computation power, new programming models and paradigms that highly rely on parallel executions requiring synchronization among the different threads have been developed.

One of the most well known among these approaches is the fork-join one, that is used in a wide range of domains from library for multicore-enabled applications like OpenMP, to Map-Reduce applications that nowadays characterize cloud services. This and other synchronization issues need to be managed with proper protocols [40] that could introduce self-suspensions in the tasks execution. Another aspect is correlated with the presence of dedicate computation units (e.g., FPGA, DSP, GPU), where threads offload highly optimized elaborations to speedup the execution, like the use of a DSP to perform signal processing (e.g., filtering, FFT). Nowadays, applications strongly rely on the communication among nodes and with devices. These could range from sensors acquisitions in the embedded domain to highly interconnected systems in the automotive environment or disk-intensive tasks in the BigData application field. All these behaviors share the necessity for the task to *self-suspend waiting for some event.*

Also real-time applications cannot neglect any more the use of this features, but they need to be provided a way that does not jeopardize timing constraints. The interest regarding scheduling analysis for self-suspending tasks has grown in recent years. However, the currently available results are not comparable

[16]More information available at: `http://www.openairinterface.org/`.

with those provided for more classical task model due to the complexity of the problem [164].

Approaches to deal with the schedulability analysis of self-suspending tasks can be dived in two main branches: *suspension-oblivious* and *suspension-aware*. In the *suspension-oblivious* approach [123, 132] the maximum suspension time that each job of a task could endure is included in the worst-case execution time (WCET) of the task. Even if this approach presents the same pessimism in the analysis as the use of busy execution (i.e., the task actively waits for the suspension to finish), it presents advantages at runtime because the task leaves the processor that could be used to reduce response time of other tasks, serve aperiodic request and best-effort activities or be reclaimed to reduce energy consumption. Instead, *suspension-aware* analysis explicitly considers suspensions in the task model and in the related schedulability analysis.

The complexity of modern software solutions has driven the adoption of a modular approach. This is now an established common practice in terms of code design and implementation. In recent years, this view has been introduced also for runtime execution to better exploit the computational power of modern platforms while reducing the complexity of the analysis. Most of the proposed approaches are based on the concept of resource reservation, that assigns a fraction of the computation time provided by the platform to each activity enforcing that no more than such an amount is effectively given. The mechanism could be applied to a huge range of platforms, from small embedded systems to server farms.

The Constant Bandwidth Server (CBS) has been originally proposed by Abeni and Buttazzo [6] for multimedia applications. They proposed it as a scheduling methodology based on reserving a fraction of the processor bandwidth to each task, under the EDF scheduling algorithm. Marzario et al. identified that the CBS is not able to ensure hard reservation due to the deadline aging problem [136]. The Hard CBS [36] (HCBS) has been proposed extending the original CBS to implement *hard reservation* [162] (i.e., guaranteeing a minimum budget in any time interval). Bertogna et. al. [27, 37] presented the BROE algorithm to provide hard real-time guarantee to tasks with an approach light enough to be implemented even in small microcontrollers. BROE extends the HCBS to handle resource sharing in hard real-time Hierarchical Scheduling Frameworks. Recently, an implementation of the Hard CBS algorithm called SCHED_DEADLINE [111] has been included in the mainline Linux. To deal with the self-suspensions caused by locking due to shared resources, Faggioli et. al. [81] proposed the multiprocessor bandwidth inheritance protocol (M-BWI) that allows to schedule self-suspending tasks under CBS using busy executions. That resource reservation approach can be used for other resources like disk and network as proposed by Valente et. al., [187, 188].

Concerning self-suspending tasks, Richard [163] identified that *suspension-aware* schedulability analysis of periodic self-suspending tasks is NP-hard in the strong sense. Ridouard et al. studied the *suspension-aware* schedulability analysis of self-suspending tasks in uniprocessor systems presenting several negative, but very interesting, results [164, 165].

Abdeddaïm and Masson [2] presented a timed automata based model for self-suspending tasks and proposed a method to test the sustainability of a schedule with respect to the execution and self-suspension durations. Recently, Nelissen et al. [141] have invalidated existing results on suspension-aware analysis for uniprocessor systems. In this work they presented an exact analysis for self-suspending task with one self-suspension region and a sufficient test in the case of multiple self-suspension regions, both in case of fixed-priority scheduling.

Liu and Anderson proposed *suspension-aware* schedulability tests for self-suspending tasks in multiprocessor systems [120, 121], addressing both G-EDF and G-FP scheduling policies. Liu and Anderson have also derived a tardiness bound [122] for self-suspending tasks in the context of soft real-time multiprocessor systems under G-EDF and G-FIFO scheduling.

This work has three main contributions:

- the identification that the Hard CBS algorithm (and its current implementation in the mainline Linux) is not able to provide resource reservation for self-suspending tasks under *suspension-oblivious* analysis;
- a novel reservation algorithm called H-CBS-SO is proposed, extending the HCBS to support temporal isolation among self-suspending tasks;
- since the novel algorithm has been implemented in the Linux kernel, implementation details are presented and experimental results aiming at evaluating the performance of the implementation in terms of run-time overhead are reported.

4.3.1 Background and notation

This chapter considers a task set Γ composed of n real-time self-suspending tasks (SS-tasks) running upon a uniprocessor system. A SS-task alternates execution and self-suspending phases; no limitation is given to the number of interleaved phases. A SS-task τ_i is characterized by a worst-case execution time (WCET) C_i, a maximum self-suspension time S_i, a period (or minimum interarrival time) T_i and an implicit relative deadline $D_i = T_i$. Each SS-task must start and end with an execution phase (i.e., with the realistic assumption to not have self-suspensions at the beginning or at the end of the "body" of the SS-task).

Each SS-task τ_i executes upon a dedicated reservation server $\mathcal{S}_i$ characterized by a budget Q_i and a period P_i. This chapter will consider two different types of reservation server algorithms: the HCBS algorithm, briefly recalled in the previous sections and in Section 4.1.1, and a novel algorithm, the H-CBS-SO, proposed in Section 4.3.2. The reservation servers are assumed to be scheduled according to the EDF scheduling policy. An example of the system configuration considered in this chapter is illustrated in Figure 4.6; this scheduling scheme is denoted as Task Isolation Framework (TIF).

4.3.2 HCBS for Suspension-Oblivious Analysis

Concerning self-suspending tasks, no *suspension-aware* schedulability tests seem to have been proposed for SS-tasks executing upon the HCBS, and this section demonstrates that the HCBS algorithm is not suitable to directly support a suspension-oblivious analysis of SS-tasks running upon HCBS servers. When considering SS-tasks executing upon HCBS servers, one can try to extend the *suspension-oblivious* approach by using the schedulability results from Theorem 1 and 2. That is, given a set of n SS-tasks τ_i, each one executing upon a dedicated HCBS server $\mathcal{S}_i$, the server parameters are configured as $Q_i = C_i + S_i$

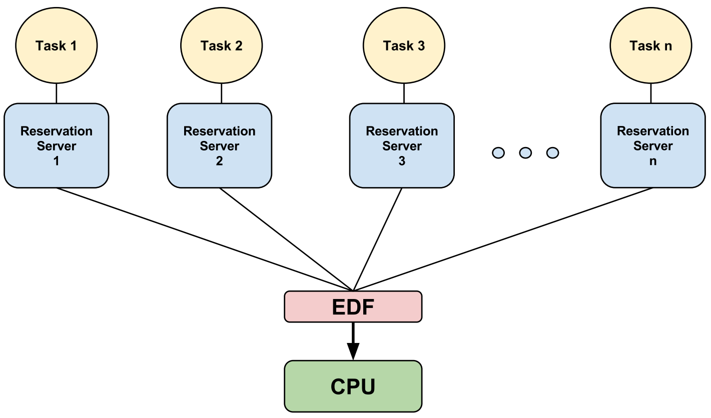


Figure 4.6: Example of a Task Isolation Framework.

and $P_i = T_i = D_i$ by accounting self-suspensions as execution times. Then, Theorem 1 and 2 are applied to verify whether the task set is schedulable. Unfortunately, the HCBS algorithm is not directly suitable to support such an approach to deal with the schedulability analysis of self-suspending tasks, as the following example illustrates.

Example 1 (HCBS and SS-task without busy execution)**.** *Consider a task τ with $C = 1$, $S = 3$ and $D = T = 8$. Suppose the task τ be executed upon a reservation server $\mathcal{S}$ having $Q = C+S = 4$ and $P = T = 8$. As Figure 4.7 shows, the task τ starts executing at $t = 4$, that is the latest time at which τ can start executing still guaranteeing $Q = 4$ budget units until the deadline $P = T = 8$. Suppose now that τ immediately self-suspends its execution: according to the classical HCBS formulation, since $\mathcal{S}$ has no more workload to be served, the server $\mathcal{S}$ becomes Idle. At time $t = 7$ the task can resume its execution. Hence, Rule 2 of the HCBS is applied. In this case $t_r = 8 - q(7)/\alpha = 6$: then, having $t > t_r$ (*bandwidth check*), the budget is immediately replenished to Q and the deadline shifted at $d = 15$. In this way, since the server (in the worst-case) cannot start executing before $d - Q = 11$, the task τ will miss its deadline. In other words, the reservation server was not able to guarantee C time units of computation in a period P, notwithstanding that the server budget was set at $Q = C + S$.* □

Since under suspension-oblivious analysis suspensions are treated as execution time, a second approach could be to replace the task self-suspensions with *busy executions*, modifying the actual task implementation. With this approach, when a task has to suspend (e.g., due to an I/O operation), it starts a busy execution wasting processor cycles. The busy execution ends when the task can be resumed from the self-suspension. This solution, illustrated by the following example, is clearly simple and has a strong practical effectiveness.

Example 2 (HCBS and SS-task with busy execution)**.** *Consider a SS-task τ and a HCBS server as in Example 1. The self-suspension of τ is replaced with a busy execution. As shown in Figure 4.8, when τ self-suspends at time $t = 4$, it continues executing wasting processor cycles until $t = 7$. Thanks to busy*

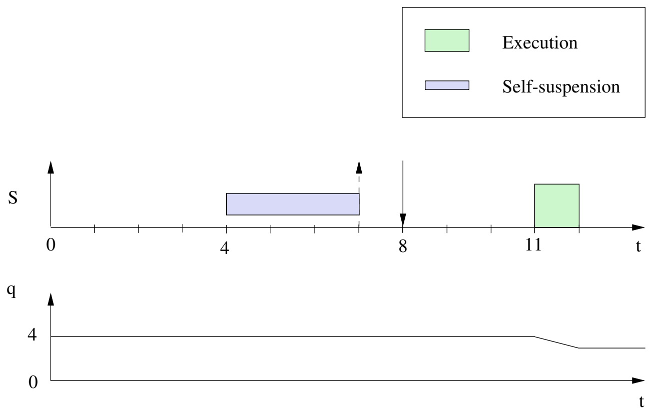


Figure 4.7: HCBS serving a SS-task: a counterexample using *suspension-oblivious* analysis.

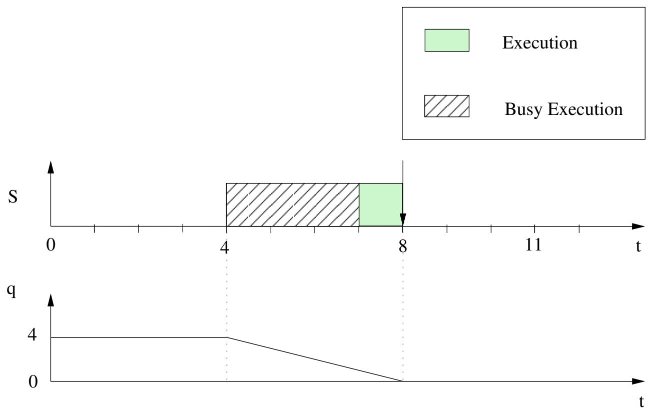


Figure 4.8: HCBS serving a SS-task where self-suspensions have been replaced with busy executions.

execution, the server is now able to guarantee $C = 1$ execution units over a period $P = 8$; this is because the server remains active without performing any bandwidth check. □

As illustrated in Example 2, after replacing self-suspension with busy execution Theorem 1 and 2 allow to check for the system schedulability of a SS-task running upon a HCBS server with $Q = C + S$. However, while replacing self-suspensions with busy executions does not provide any benefits when a *suspension-oblivious* analysis is used, it is clearly worse when other performance metrics are considered. For example, since the busy execution consists in wasting processor cycles, it is not possible to reclaim the idle-time generated from the self-suspensions, which could be used to serve non real-time workload or improve the average response-times of the tasks. In the same way, also when energy constrained systems are addressed, it is preferable to avoid busy executions still guaranteeing the schedulability of the system.

The key observation is that using busy execution, if a server with pending workload is the highest priority one (i.e., earliest absolute deadline) then it is executing; whilst with self-suspension of servers this is not still true. The highest priority server can be in idle state due to a self-suspension, while in practice this server should not be considered as an idling server since it has pending workload temporarily self-suspended.

Looking at Example 1, it is possible to notice that deadline miss is caused by the *bandwidth check* ($t > t_r$) that disallows the server to execute in the time interval [7,8]. Unfortunately, as the following example shows, this problem cannot be solved by simply removing the *bandwidth check* when a server is resumed from a self-suspension.

Example 3 (HCBS without *bandwidth check* and SS-tasks)**.** *Consider a task set composed of two self-suspending tasks, τ_1 having $C_1 = 2, S_1 = 0, T_1 = D_1 = 4$, and τ_2 having $C_2 = 2, S_2 = 1, T_2 = D_2 = 7$. Using* suspension-oblivious *analysis this task set results schedulable under EDF, since*

$$\sum_{i=1}^{2} \frac{C_i + S_i}{T_i} = \frac{2}{4} + \frac{2+1}{7} \leq 1. \tag{4.5}$$

Suppose that both τ_1 and τ_2 are associated to two HCBS servers $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with $Q_1 = C_1 + S_1 = 2$ and $Q_2 = C_2 + S_2 = 3$. The period of the servers is the same of the served task. As Figure 4.9 shows, both the servers are released at the same time; then, according to the EDF policy, $\mathcal{S}_1$ starts executing. At time $t = 2$ the server $\mathcal{S}_2$ can start to execute and immediately self-suspends its execution. Suppose now that τ_2 violates its maximum self-suspension time, self-suspending its execution for 2 time units. Then, at time $t = 4$, τ_2 resumes its execution without performing the bandwidth check *and executes for 3 time units, making 1 time unit of overrun. Since the budget of the server $\mathcal{S}_2$ was set to $Q_2 = 3$, the task τ_2 is able to overrun without any budget exhaustion mechanism is triggered. Finally, at time $t = 7$, τ_1 can start executing missing its deadline at time $t = 8$. Unfortunately, since a greater self-suspension time and an overrun of τ_2 compromise the schedulability of τ_1, this example shows that the task isolation property (Proposition 1) of the HCBS could be broken.* □

To address the problems discussed in this section, this chapter proposes an extension of the HCBS algorithm which "takes into account" task self-suspensions exactly as if the task was busy executing. In this way, the reservation server algorithm can be analyzed with a *suspension-oblivious* analysis.

The H-CBS-SO Algorithm

In this section is defined the H-CBS-SO algorithm, an extension of the HCBS algorithm to support self-suspending tasks analyzable with a *suspension-oblivious* analysis, and its basic properties can be derived.

The H-CBS-SO server introduces a new state (with respect to HCBS) denoted Self-Suspended. As Figure 4.10 illustrates, this new state can be reached by transitions from the Ready and the Suspended states; the Self-Suspended state allows transitions to the Ready and the Suspended states.

The H-CBS-SO extends the HCBS algorithm reported in Definition 1 by adding the following rules and the SS-QUEUE data structure:

Definition 2 (H-CBS-SO Algorithm)**.**

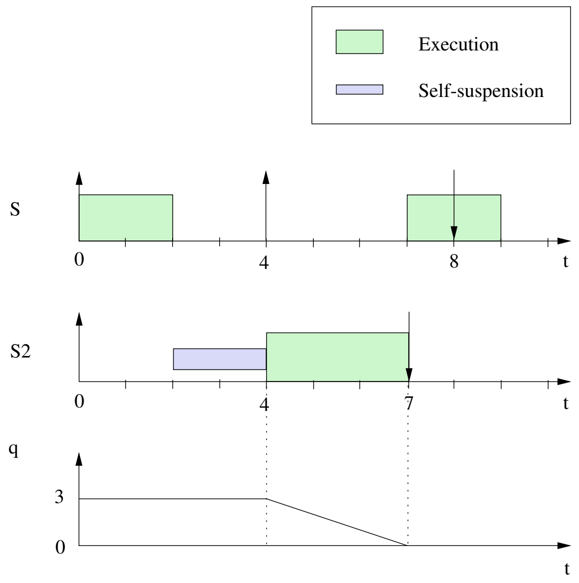


Figure 4.9: HCBS with SS-tasks: the *bandwidth check* of the HCBS is disabled.

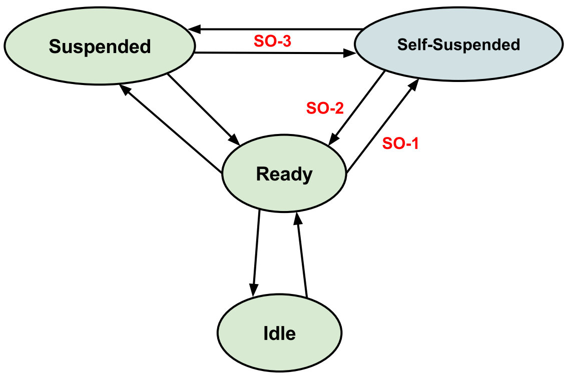


Figure 4.10: State transition diagram for the H-CBS-SO algorithm.

SS-QUEUE. *A queue containing all the server in the Self-Suspended state is introduced; the server in that queue are ordered by increasing absolute deadline.* S_{SS} *denotes the server on top of SS-QUEUE,* d_{SS} *and* q_{SS} *its absolute deadline and budget respectively. If SS-QUEUE is empty,* $d_{SS} = q_{SS} = \infty$ *is set.*

- ***Rule SO-1***. *When a task* τ *self-suspends, the server associated to* τ *becomes Self-Suspended and is inserted in the SS-QUEUE.*
- ***Rule SO-2***. *When a task* τ *resumes form self-suspension, the server associated to* τ *becomes Ready and is removed from the SS-QUEUE.*
- ***Rule SO-3***. *When* $q_{SS} = 0$*, the server* $\mathcal{S}_{SS}$ *is removed from the SS-QUEUE, becomes Suspended and is suspended until time* d_{SS}*.* $\mathcal{S}_{SS}$ *is renamed by* $\mathcal{S}_j$*. At time* d_j*, the budget* q_j *is replenished to* Q_j *and the deadline* d_j *is postponed to* $d_j + P_j$*; finally,* $\mathcal{S}_j$ *becomes Self-Suspended and is inserted in the SS-QUEUE.*
- ***Rule SO-4***. *(Budget accounting rule) - If a server* S_i *is executing and* $d_i \geq d_{SS}$ *holds, then* both q_{SS} *and* q_i *are decreased. On the other hand, if* $d_i < d_{SS}$ *only* q_i *is decreased. If there is no server in the Ready state (idle-time), then* q_{SS} *is decremented.*

Notice that Rules SO-1 and SO-2 do not modify the deadline of the server. Also, notice that, when the server is moved from the Self-Suspended state to the Ready state, Rule SO-2 does not perform any *bandwidth check* (Rule 2 of HCBS).

Example 4 (H-CBS-SO and SS-tasks, following Example 1). *Consider the same SS-task* τ *of Example 1 executing upon a server* $\mathcal{S}$ *implemented with the H-CBS-SO algorithm. Suppose also that* τ *starts executing at* $t = 4$ *as in the previous example and immediately self-suspends its execution. According to the H-CBS-SO algorithm, the server* $\mathcal{S}$ *becomes self-suspended and is inserted in SS-QUEUE. Then, Rule SO-3 is applied and the budget* $q(t)$ *is decremented until* τ *resumes its execution (time* $t = 7$*). At this time, Rule SO-2 is applied and* $\mathcal{S}$ *becomes Ready with budget* $q = 1$*. Since Rule SO-2 does not provide any bandwidth check, the server* $\mathcal{S}$ *can execute until time* $t = 8$ *avoiding* τ *to miss its deadline.* □

Example 5 (H-CBS-SO and SS-tasks, following Example 3). *Consider the same task set of Example 3 where the servers* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are implemented with the H-CBS-SO algorithm. Both the servers are configured with the same parameters of the previous example. At time* $t = 2$ *the server* $\mathcal{S}_2$ *can start to execute and immediately self-suspends its execution; then, according to Rule SO-1,* $\mathcal{S}_2$ *becomes Self-Suspended and is inserted in SS-QUEUE. Suppose now that* τ_2 *violates its maximum self-suspension time, self-suspending its execution for 2 time units. During the self-suspension, Rule SO-3 is applied and the budget* q_2 *is decremented by 2 time units. Then, at time* $t = 4$*,* τ_2 *resumes its execution with budget of* $\mathcal{S}_2$ *equal to* $q_2 = 3 - 2 = 1$*.*

Likewise Example 3, τ *is supposed to have an execution of* 3 *time units, making* 1 *time unit of overrun. In this case, having used the H-CBS-SO algorithm,* τ *is able to execute only in the time interval* $[4, 5]$ *due to a budget exhaustion at time* $t = 5$*.*

Finally, at time $t = 5$, τ_1 can start executing without missing its deadline at time $t = 8$. Unlike Example 3, in this case a greater self-suspension time and an overrun of τ_2 do not compromise the schedulability of τ_1, guaranteeing the task isolation property. $\square$

The rules of the H-CBS-SO have been designed to imitate the parameters updating of a server in the HCBS with busy execution algorithm, but removing the wasting of processor cycles typical of the busy execution. This property is related to the H-CBS-SO server having earlier deadline, since it is the one that would have executed when self-suspensions are replaced with busy executions. This is expressed in the following proposition, which is the main property of the H-CBS-SO algorithm.

Proposition 2. *With H-CBS-SO, self-suspensions of the server having the earlier deadline are accounted as with HCBS with busy executions.*

Proof. First, consider the state-transitions from Ready to Self-Suspended and from Self-Suspended to Ready (Rules SO-1 and SO-2, respectively). Consider a server $\mathcal{S}_{SS}$ in the Ready state and executing (d_{SS} is the earliest deadline) serving an active task τ_{SS} that self-suspends. Then Rule SO-1 is applied and the server becomes Self-Suspended. Two cases can be distinguished: (i) a server $\mathcal{S}_i$ executes while $\mathcal{S}_{SS}$ is Self-Suspended (ii) there is no such server $\mathcal{S}_i$ while $\mathcal{S}_{SS}$ is Self-Suspended (idle-time). In case (i), first suppose $d_i < d_{SS}$. Then with HCBS with busy execution $\mathcal{S}_{SS}$ is preempted by $\mathcal{S}_i$ and q_{SS} is not decremented. Similarly, Rule SO-4 implies that q_{SS} is not decremented while $\mathcal{S}_{SS}$ is in the Self-Suspended state. Let us instead suppose $d_i \geq d_{SS}$. Then with HCBS with busy exection $\mathcal{S}_{SS}$ continues executing (i.e., $\mathcal{S}_i$ can not preempt $\mathcal{S}_{SS}$) and the budget q_{SS} is decremented accordingly. Similarly, Rule SO-4 implies that q_{SS} is decremented. In case (ii), with HCBS with busy execution $\mathcal{S}_{SS}$ continues (busy) executing (there is no idle-time) and the budget q_{SS} is decremented accordingly. Similarly, Rule SO-4 implies that q_{SS} is decremented. Finally, consider a server $\mathcal{S}_{SS}$ in the Self-Suspended state and resuming its execution. Then Rule SO-2 is applied and the server becomes Ready without performing any budget check. As with HCBS with busy execution, $\mathcal{S}_{SS}$ continues executing with (unchanged) deadline d_{SS} and budget q_{SS}.

Now consider the remaining state-transitions: from Self-Suspended to Suspended and from Suspended to Self-Suspended (Rule SO-3). Consider a server $\mathcal{S}_{SS}$ in the Self-Suspended state and exhausting its budget q_{SS} ($q_{SS} = 0$, after applying Rule SO-4). By Rule SO-3, $\mathcal{S}_{SS}$ is removed from the SS-QUEUE, becomes Suspended and is suspended until d_{SS}; the budget q_{SS} is then no more decremented by applying Rule SO-4. A similar result holds with HCBS with busy execution by applying Rule 3. Consider a server $\mathcal{S}_j$ in the Suspended state serving a self-suspended SS-task τ_j at time d_j. By Rule SO-3, the budget q_j is replenished to Q_j and the deadline is postponed to $d_j + P_j$. A similar result holds with HCBS with busy execution by applying Rule 3. Finally, Rule SO-3 inserts $\mathcal{S}_j$ into the SS-QUEUE, thus leading back to the cases described in the first part of the proof. $\square$

When self-suspensions are replaced by busy executions, it is clearly not possible to have nested self-suspensions, because the processor would be occupied by the busy execution in place of the self-suspension of the server having the earliest deadline. When nested self-suspensions occur, the H-CBS-SO server does not imitate the parameters updating of the HCBS with busy executions. However, the following proposition addresses this point showing that nested self-suspensions do not affect the system schedulability.

Proposition 3. *If a deadline is missed by a H-CBS-SO server, then it would be missed also with HCBS with busy executions.*

Proof. Since H-CBS-SO extends HCBS by adding the Self-Suspended state and rules describing transition to and from this state, it is necessary to consider only the behavior of the H-CBS-SO server triggered by self-suspensions. It is possible to distinguish two cases: non-nested self-suspensions and nested self-suspensions. In the first case, since there is only a single self-suspension, Proposition 2 guarantees that the H-CBS-SO has the same behavior of HCBS with busy executions. Consider now the second case: suppose $\mathcal{S}_j$ be a H-CBS-SO server that is self-suspended when at least another server is self-suspended (*i.e.,* nested self-suspension). Let SS be the set of self-suspended servers having deadlines $d_i \leq d_j$, $\forall \mathcal{S}_i \in SS$. If the set SS is empty, then $\mathcal{S}_j$ is the self-suspended server having earlier deadline, and the considerations of Proposition 2 can be applied to replicate the behavior of the HCBS with busy execution. From now on suppose that the set SS is not empty, hence considering the case in which $\mathcal{S}_j$ is in the SS-QUEUE and it is not the first server in SS-QUEUE. Let t be the earlier time instant at which $\mathcal{S}_j$ is resumed from its self-suspension or the set SS becomes empty, which is the time instant at which the server $\mathcal{S}_j$ stops to have a budget update different from the case of HCBS with busy execution (i.e., also the time at which $\mathcal{S}_j$ is removed from the SS-QUEUE or it becomes the first server in SS-QUEUE).

If the deadline d_j is missed, then $\mathcal{S}_j$ was not able to execute $q_j(t)$ before time d_j, i.e., $q_j(t) + I > (d_j - t)$, where I is the interference caused by servers having earlier deadline than $\mathcal{S}_j$ from time t on.

In case of using the HCBS with busy execution, due to the busy execution of self-suspended servers in the set SS, the budget $q_j^{'}(t)$ of $\mathcal{S}_j$ at time t is greater or equal than the one with H-CBS-SO, i.e., $q_j^{'}(t) \geq q_j(t)$. This is because server $\mathcal{S}_j$ could have executed during self-suspensions of the servers in the set SS, while it would be prevented from executing in case of self-suspensions replaced by busy executions.

Hence, if $q_j(t) + I > (d_j - t)$ then also $q_j^{'}(t) + I > (d_j - t)$ holds, thus concluding the proof.

□

Proposition 3 allows to use the schedulability test from Theorem 2 for SS-tasks after inflating their WCETs with the worst-case self-suspension times.

Theorem 3 (Suspension-Oblivious Analysis)**.** *Given a set of n SS-tasks τ_i with implicit deadline, each SS-task is associated with a H-CBS-SO server $\mathcal{S}_i$. For each H-CBS-SO server $\mathcal{S}_i$, $Q_i = C_i + S_i$ and $P_i = T_i = D_i$ are defined. Then, this set of tasks, each one executing upon a dedicated H-CBS-SO, is schedulable with EDF if Equation (4.1) holds.*

Notice that the schedulability condition from Theorem 3 is sufficient but, in general, not necessary for the schedulability of the task set. This is illustrated in the following example.

Example 6. *Consider a task set composed of two self-suspending tasks, τ_1 having $C_1 = 1, S_1 = 2, T_1 = D_1 = 4$, and τ_2 having the same parameters. Assuming that both τ_1 and τ_2 are associated to two H-CBS-SO servers $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with $Q_i = C_i + S_i = 3$ for $i = 1, 2$. The period of the servers is the same of the served task. Using* suspension-oblivious *analysis this task set results non-schedulable under EDF, since*

$$\sum_{i=1}^{2} \frac{C_i + S_i}{T_i} = \frac{3}{4} + \frac{3}{4} > 1. \tag{4.6}$$

However it is easy to see, by checking all possible schedules over the hyperperiod of 4, that this task set is EDF schedulable. This is because the self-suspension times of a given job can be used to execute other pending workload. □

Notice that multiple servers can be in the Self-Suspended state (i.e., enqueued in the SS-QUEUE) at the same time-instant, as illustrated in the following example. This justifies the presence of the queue.

Example 7. *Consider a task set composed of three self-suspending tasks, τ_1 having $C_1 = S_1 = 1$, $T_1 = D_1 = 4$, τ_2 having $C_2 = S_2 = 1$, $T_2 = D_2 = 8$ and τ_3 having $C_3 = S_3 = 1$, $T_3 = D_3 = 10$ executing within H-CBS-SO servers $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, respectively, with $Q_1 = 2$, $P_1 = 4$, $Q_2 = 2$, $P_2 = 8$ and $Q_3 = 2$, $P_3 = 10$. Suppose that at time $t = 0$ server $\mathcal{S}_1$ starts and immediately self-suspends (i.e., it is inserted in the SS-QUEUE) and that, subsequently, server $\mathcal{S}_2$ starts and self-suspends. In this situation $\mathcal{S}_1$ is the head of the queue (i.e., $\mathcal{S}_{SS} = \mathcal{S}_1$), having earlier deadline than $\mathcal{S}_2$. Suppose then that server $\mathcal{S}_3$ starts executing for C_3 time-units; according to Rule SO-4, $q_{SS} = q_1$ is decremented until its exhaustion. This behavior reflects Proposition 2: with HCBS with busy execution, server $\mathcal{S}_1$ would have continued to (busy) execute disallowing $\mathcal{S}_2$ and $\mathcal{S}_3$ to execute; this also could explain the ordering by increasing deadlines of the SS-QUEUE.* □

The H-CBS-SO is able to guarantee temporal isolation for SS-tasks, similarly to the isolation property of the HCBS reported in Proposition 1. This is expressed by the following proposition:

Proposition 4 (Isolation property of the H-CBS-SO). *Given a set of n SS-tasks, each one running upon a dedicated H-CBS-SO server as in Theorem 3, such that condition 4.1 holds, if a task exceeds its WCET C or/and its maximum self-suspension time S then the schedulability of the other SS-tasks will not be affected.*

Proof. Since H-CBS-SO extends HCBS by adding the Self-Suspended state and rules describing transition to and from this state, this similarly follows from Proposition 2 and Proposition 3, after recalling Proposition 1. □

Idle-time reclaiming. The H-CBS-SO algorithm, contrary to the HCBS algorithm with busy execution, is able to guarantee, together with the schedulability of the task set (Theorem 3), a higher lower-bound on the idle-time (i.e., time-instant where no server has pending workload).

Consider a task set composed of n SS-tasks each one executing over a dedicated H-CBS-SO server with $Q_i = C_i + S_i$ and $P_i = T_i$ such that condition expressed in Equation (4.1) holds. $H := \text{lcm}\{P_1, \ldots, P_n\}$ denotes the hyperperiod of the servers.

Suppose that all the servers are released simultaneously at time $t = 0$. At time $t = H$ the length of the idle-time intervals in $[0, H)$ is lower-bounded by the following quantity:

$$\mathcal{I}_{SS} \geq H \left(1 - \sum_{i=1}^{n} \frac{C_i}{P_i}\right) \tag{4.7}$$

Since during self-suspension times the SS-task does not require to execute, at time $t = H$ the workload executed by the processor is at most $\mathcal{C} = \sum_{i=1}^{n} C_i \frac{H}{T_i}$. Then the idle-time on the interval $[0, H)$ is at least the difference between the elapsed time H and the time $\mathcal{C}$ in which the processor has executed.

Notice that, with HCBS with busy execution, since self-suspensions are replaced by busy executions, then the idle-time in $[0, H]$ is lower-bounded by

$$\mathcal{I}_{BE} \geq H \left(1 - \sum_{i=1}^{n} \frac{C_i + S_i}{P_i}\right) \leq \mathcal{I}_{SS}. \tag{4.8}$$

It is important to remark that, with HCBS without busy execution, even if we can guarantee the same lower-bound $\mathcal{I}_{SS}$ for idle-time, it is not possible to guarantee the schedulability of SS-tasks as shown in Example 1.

The idle-time guaranteed by the H-CBS-SO can be exploited by integrating such a reservation algorithm with reclaiming mechanisms like IRIS [136] or H-GRUB [8]. Both these algorithms have been designed to work with HCBS and are able to reclaim the idle-time guaranteeing that the spare bandwidth is fairly distributed among the needing servers. IRIS is based on an update of the server parameters when an idle-time occurs while H-GRUB provides a budget accounting rule that takes into account the spare bandwidth. The idle-time could be also exploited for energy-management purposes through the utilization of power-aware scheduling algorithms able to use such a wasted intervals of time to reduce the energy consumption applying techniques of Dynamic Frequency and Voltage Scaling (DVFS) and Device Power Management (DPM) like those proposed by Aydin et. al. [15] and Marinoni et. al. [134].

Generalization for Hierarchical Scheduling

The *hierarchical* or *component-based* design has been widely accepted as a methodology to enable modularity and simplify the analysis of large and complex systems (e.g., [175, 176]). This section extends the H-CBS-SO algorithm to the case where multiple tasks can be run upon the same server, after extending the system model from Section 4.3.1.

Consider a two-level hierarchical system. The system is composed of n H-CBS-SO servers $\{(Q_i, P_i)\}$, with server $\mathcal{S}_i := (Q_i, P_i)$ serving a set Γ_i of implicit-deadline SS-tasks. The task model is as in Section 4.3.1; The notation $\tau := (C_\tau, S_\tau, T_\tau)$ denotes a SS-task. The *global scheduler* is based on the H-CBS-SO algorithm. Each subsystem uses a *local scheduler* to select the running task; EDF is used as local scheduling policy.

H-CBS-SO rules are used as in Definition 2 but replacing Rules SO-1 and SO-2 with, respectively:

- **Rule SO-1-hier**. If there is no workload to execute on the server $\mathcal{S}_i$ *and* at least one task in Γ_i is self-suspended, then $\mathcal{S}_i$ becomes Self-Suspended and is inserted in the SS-QUEUE (if not already in the SS-QUEUE).
- **Rule SO-2-hier**. If there is workload to execute on the server $\mathcal{S}_i$ *or* there is no task in Γ_i which is self-suspended, then $\mathcal{S}_i$ becomes Ready and is removed from the SS-QUEUE (if in the SS-QUEUE).

With this modifications, it is possible to follow the proof of Proposition 2 thus concluding that self-suspensions *of the server* are accounted with H-CBS-SO as busy executions are accounted with HCBS with busy executions. As a direct consequence of this observation, the Isolation Property (or *global schedulability analysis*) of Proposition 4 extends to the hierarchical context.

The local (suspension-oblivious) schedulability analysis of a H-CBS-SO server can be performed using the test proposed in [175]. According to this test, the task set Γ_i is schedulable by EDF on the server $\mathcal{S}_i$ if

$$\forall t > 0 \qquad \mathrm{dbf}(\Gamma_i, t) \leq \mathrm{sbf}(\mathcal{S}_i, t), \tag{4.9}$$

where

$$\mathrm{dbf}(\Gamma_i, t) := \sum_{\tau \in \Gamma_i} \left\lfloor \frac{t}{T_\tau} \right\rfloor \cdot (C_\tau + S_\tau) \tag{4.10}$$

is the *demand bound function* of the task set Γ_i (i.e., the maximum computational demand of Γ_i in any interval of length $t > 0$) and $\mathrm{sbf}(\mathcal{S}_i, t)$ is the *supply bound function* of the server $\mathcal{S}_i$ (i.e., the minimum amount of service time provided by the server in any interval of length $t > 0$). An expression for $\mathrm{sbf}(\mathcal{S}_i, t)$ (the same of the HCBS) can be found in [119]. Techniques to solve 4.9 can be found in [22].

4.3.3 Linux Implementation

This section describes how the H-CBS-SO has been implemented in the Linux kernel and shows the overhead introduced by this extension.

The Linux kernel 3.14 introduces a new scheduling class called SCHED_DEADLINE that implements the HCBS algorithm. This infrastructure can be extended to implement the novel H-CBS-SO algorithm and verify its performance. In particular, the implementation has been made by modifying version 3.19 of the vanilla Linux kernel.

SCHED_DEADLINE is characterized by a strong relationship between tasks and servers: each server must have one and only one associated task. This leads to the lack of a clear distinction in the data structures. For this reason, in SCHED_DEADLINE, the terms “task” and “server” represent the same entity and can be used be used both interchangeably.

Modifications

This section presents the data structures and the functions that have been added or modified to implement the H-CBS-SO algorithm.

Data Structures

The SS-QUEUE (see Definition 2) is implemented through a red-black tree, which has a complexity of $O(\log(n))$ for insertion, removal and search operations, where n represents the number of elements in the tree. The SS-QUEUE is arranged by absolute deadlines, thus the server S_{SS} (*i.e.*, the one having earlier deadline) is represented by the leftmost element of the tree. Because the most frequent operations (e.g., removal and budget update) are performed on S_{SS}, a pointer to the leftmost element is added to speed up these common operations, reducing the complexity to $O(1)$.

An additional flag must be added for each task in order to keep trace of its self-suspension status, because the system must determine if that task was self-suspended when a budget replenishment is performed on it. This is due to the fact that it needs to be decided if the task must be reinserted in the SS-QUEUE following the SO-3 rule or can be set as ready.

Functions

A performing solution to catch the transitions to and from the self-suspension state can be obtained intercepting the insertion or removal of the tasks in the SCHED_DEADLINE runqueue. The reasons leading to the removal of a task from the runqueue are (i) server budget exhaustion; (ii) task termination; (iii) scheduling class modification and (iv) self-suspension, hence, the self-suspension is detected when a task is removed from the runqueue and this removal is not caused by one of the first three cases.

Below is presented how the H-CBS-SO rules are implemented in SCHED_DEADLINE.

SO-1. When a SCHED_DEADLINE task leaves the runqueue, it is checked if the cause of this transition is a self-suspension, and if this is the case, it is inserted in the SS-QUEUE.

SO-2. If a self-suspended SCHED_DEADLINE task enters the runqueue, then it can switch to the ready state, thus be removed from the SS-QUEUE.

SO-3. When the S_{SS} exhausts its budget, then its self-suspension flag is set, it becomes suspended and a timer is activated for its replenishment. If the self-suspension flag is active when the budget is replenished, then the task is inserted in the SS-QUEUE, otherwise it is inserted in the runqueue.

SO-4. SCHED_DEADLINE tasks budgets are periodically updated. In the new implementation, the budget of the S_{SS} is also updated when needed by the server rule. To minimize the overhead, no budget updates are performed on the SS-QUEUE when there are no ready SCHED_DEADLINE tasks. This idle time is measured with a timer and is used to bring back the system to a consistent state by scaling it from the budgets of the self-suspended tasks.

Performance Evaluation

Some tests are performed to evaluate the overhead of the newly introduced SCHED_DEADLINE functions by the H-CBS-SO implementation. The tests are performed running several periodic tasks and measuring the execution times of the two implementations.

Setup

The tests are performed on a machine equipped with an Intel Core 2 Duo running at 3 GHz. Measurements are obtained through the Ftrace tool, that is the internal Linux kernel tracer and is used as function profiler in this experiments.

Task Sets

The task sets used for the performance evaluation have a number of tasks equal to 2^k, where $k = \{1, \dots, 10\}$. The total utilization factor of each task set is chosen equal to $U = 0.8$. The utilization factor U_i of each task is randomly generated such that the minimum value is $U_{lb} = \frac{U}{n+1}$ and $\sum_i U_i = U$. The utilization factor is defined according the ***suspension-oblivious*** analysis, *i.e.,* it accounts for self-suspensions times as execution times. The period of each job is chosen as a random value between 0.1 ms and 10 ms. Each task releases 10000 jobs that (i) busy waits to simulate execution; (ii) self-suspends; (iii) busy waits again and (iv) waits for next activation.

The last step is implemented with the use of the *sched_yield()* system call, which in SCHED_DEADLINE zeros the budget and triggers the suspension until the next activation.

For each job, the busy wait to simulate the task execution is equal to $\frac{1}{6}$ of the total execution time, while the remaining $\frac{2}{3}$ is related to the self-suspension.

Execution Times

The following figures compare the performance of the SCHED_DEADLINE scheduling class before and after the H-CBS-SO extension, showing the execution times of the modified kernel functions. The overhead before the extension is indicated with the label HCBS, since it is the name of the native server algorithm in SCHED_DEADLINE.

Each plot displays the execution times values as a function of the number of tasks in the task set of each modified function. The results are expressed in microseconds in a logarithmic scale. The circles and the squares represent the mean values of the two implementation, while the vertical bars delimit the minimum and the maximum values.

The main observation is that the additional overhead required for implementing the H-CBS-SO has been observed to be considerably small in all the tested scenarios.

Figure 4.11 shows the overhead introduced in the *update_curr_dl()* function, which updates the budgets of the running task and the S_{SS} (see rule SO-4). In addition to the budget update, if S_{SS} exhausts its budget, it is removed from the SS-QUEUE (see rule SO-3). This removal operation is the reason of the execution time growth of this function when the number of tasks increases.

Figure 4.12 shows the overhead introduced in the *enqueue_task_dl()* function, which puts a task into the SCHED_DEADLINE runqueue, and so, may resume a self-suspended task from the SS-QUEUE. Also in this case, the SS-QUEUE removal operation is the cause of the general larger execution time of this function. If the function is executed for a task which was in Suspended state due to the SO-3 rule, then it simply updates a flag and returns. This flag is required in order to decide if, after the budget replenishment, the task must turn back to the Ready state or follow the SO-3 rule, and so, be pushed in the

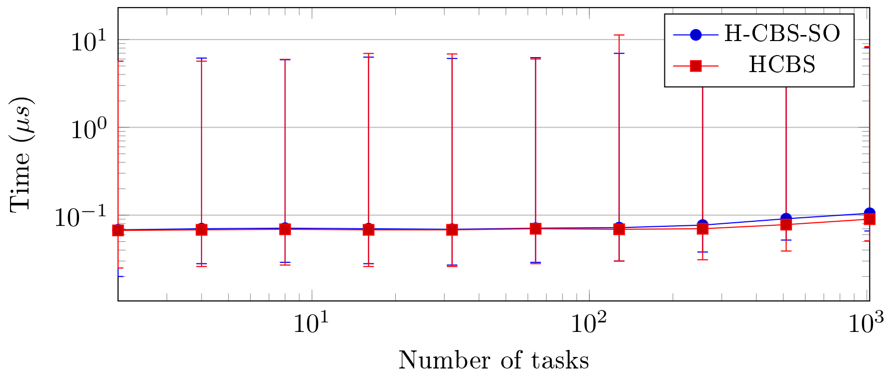


Figure 4.11: *update_curr_dl()* execution times.

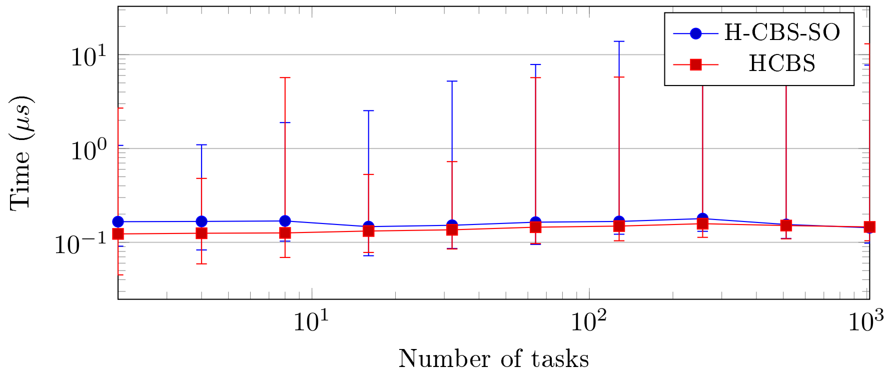


Figure 4.12: *enqueue_task_dl()* execution times.

SS-QUEUE. The flag update is a simple and quick operation and the probability of performing this operation increases with the number of self-suspending tasks. This is the reason of the convergence of the two execution time of the function before and after the H-CBS-SO extension.

The Figure 4.13 shows the overhead introduced in the *dequeue_task_dl()* function, which removes a task from the SCHED_DEADLINE runqueue, and so, may be caused by a self-suspension. This operation justifies the increased overhead shown in the picture, but the execution time grows logarithmically with the number of tasks, because of the rb-tree insert operation. The number of tasks used for this experiment is too small to show this behavior.

The Figure 4.14 shows the overhead introduced in the *dl-_task-_timer()* function, which replenishes the budget of a suspended task. If the suspended task was self-suspended, then it must be pushed back in the SS-QUEUE, otherwise, it is pushed in the runqueue. Performance are apparently improved because, by splitting SS-QUEUE and runqueue, the nodes of the two trees are less, resulting in a lower access time to their elements.

Overall, the figures show that the computational cost of the new functions is comparable to the original SCHED_DEADLINE implementation. As a result, the modifications introduced to implement the H-CBS-SO algorithm present a low impact on the system load.

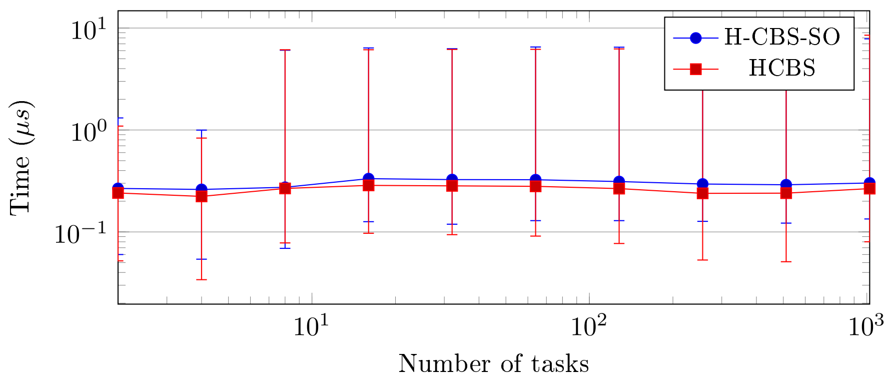


Figure 4.13: *dequeue_task_dl()* execution times.

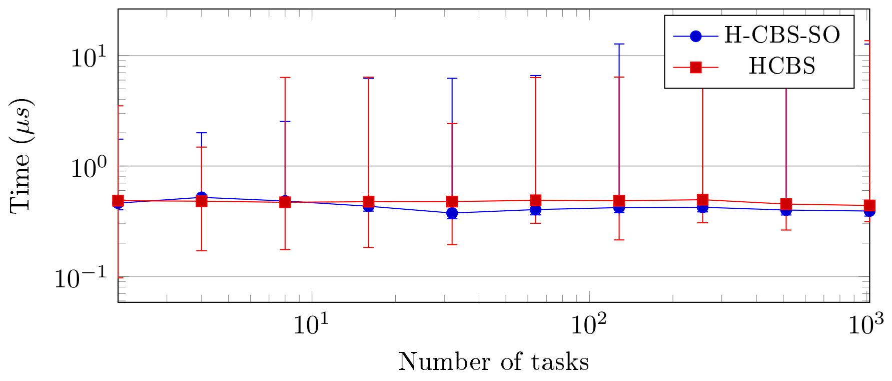


Figure 4.14: *dl_task_timer()* execution times.

4.3.4 Conclusions and Open Challenges

Modern platforms are composed of multiple heterogeneous computation units and are managed with software infrastructures providing temporal isolation among concurrent applications. To guarantee timing constraints of real-time applications executing in this kind of environments, new task models and scheduling algorithms must be introduced.

In this chapter is presented the novel H-CBS-SO scheduling algorithm that provides resource reservation for real-time self-suspending tasks. It has been shown that it is able to guarantee a *suspension-oblivious* schedulability analysis for self-suspending tasks running upon H-CBS-SO servers, while avoiding to waste processor cycles, as happens when using busy executions in place of self-suspensions.

The proposed algorithm has been implemented in the Linux kernel, as an extension of the SCHED_DEADLINE scheduling class, today part of the mainline of Linux. The implementation has been described and evaluated in terms of run-time overhead.

An open challenge is to address is the issue of suspension-aware analysis of self-suspending tasks scheduled using HCBS based reservation servers, as well as integrating existing idle-time reclaiming mechanisms with the H-CBS-SO algorithm. In addition, synchronization protocols for resource reservation scheduling should be extended to cope with self-suspensions related with such protocols.

Chapter 5

Heterogeneous Architectures

Computing platforms are evolving towards heterogeneous architectures including processors of different types, and hardware accelerators, like graphics processing unit (GPU), neural networks accelerator, cryptographic accelerator, signal processing accelerator, or other custom functions.

The advantages of using heterogeneous architectures are not limited to the computational performance boosting, but also on the energy efficiency that these ad-hoc devices provide.

An example of widespread heterogeneous architectures is represented by the Arm big.LITTLE family of microprocessors and the upcoming DynamIQ. The big.LITTLE architecture embeds two different kinds of CPU architectures sharing the same Instruction Set Architecture (ISA), but one architecture is more complex and performing, the other is simpler and more energy efficient. The DynamIQ architecture is an evolution of the big.LITTLE, which may include more than two different CPU architectures, sharing the same instruction set, but with different computational speed and energy consumption characteristics. These microprocessors find a wide use in mobile devices, for which the energy consumption is a mandatory constraint, but at the same time may require high computing power for some specific workloads, e.g., video games and multimedia.

Counting more than two billion devices, Android is nowadays one of the most popular open-source general-purpose operating systems, based on Linux. Because of the diversity of applications that can be installed, it manages a number of different workloads, some of them requiring performance/QoS guarantees. When running audio processing applications, the user would like an uninterrupted, glitch-free, output stream that reacts to the user input, typically with a delay not bigger than $4 - 10$ ms, while keeping the energy consumption of the mobile device as low as possible. Section 5.1 focuses on improvements to the real-time audio processing performance on Android. Such improvements were achieved by using the previously mentioned deadline based scheduler SCHED_DEADLINE and an adaptive scheduling strategy that dynamically modulates the allocated runtime.

The proposed strategy is evaluated through an extensive set of experiments, showing that 1) compared to the existing way to ensure low-latency audio processing, the proposed mechanism provides an energy saving of almost 40%, and 2) compared to the existing way to achieve a good balance between power consumption and latency in a glitch-free audio processing experience, the proposed solution reduces audio latency from 26.67 ms to 2.67 ms, at the expense of a limited power consumption increase of 6.25%.

The just mentioned work highlighted the need for the development of proper power-consumption models for heterogeneous multicore architectures that capture the variability of energy consumption based on processing workload type, in addition to the classical variables considered in the literature, like type and frequency of the CPU. As shown in Section 5.2, this problem is motivated presenting experimental results gathered on a Odroid-XU3 board equipped with an Arm big.LITTLE SoC, showing that power consumption has a non-negligible dependency on the workload type. The section also presents a model to define the execution time of the tasks, which depends on both the workload, and the CPU frequency and architecture. The validation of these models is performed through modifications to the open-source RTSIM real-time scheduling simulator to extend its CPU power consumption and execution time duration models, integrating results taken from the real platform. The developed tool constitutes a useful base for future research in energy-aware real-time scheduling on heterogeneous platforms, being a useful solution to provide preliminary results of different energy-aware scheduling policies.

One of the most flexible solutions to implement hardware accelerators for speeding up specific functions is with the use of field programmable gate arrays (FPGAs), that can be erased and reprogrammed with the requested functionalities. More specifically, heterogeneous platforms equipped with processors and field programmable gate arrays (FPGA) can be exploited to accelerate specific functions triggered by software activities.

The increasing capacity and performance of modern FPGAs, have made them attractive in several application domains, including space applications [1, 145]. With satellite lifetimes increased far beyond 10 years, re-programmability in flight becomes a stringent requirement. Moreover, in space environments, where radiation can cause bit flips in memory elements and ionisation failure in semiconductors, the use of reconfigurable hardware allows modifying on-board functions by replacing faulty/outdated designs at different stages of a mission.

Thanks to the Dynamic Partial Reconfiguration (DPR) capabilities of modern FPGAs, such functions can be programmed at run-time, allowing for the virtualization of the available area to support several hardware modules in time sharing, hence making them even more attractive.

Section 5.3 reports some preliminary experimental studies conducted to evaluate the feasibility of the proposed approach, profile the temporal parameters involved in such systems (e.g., reconfiguration and execution times) and identify possible bottlenecks. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to fully static approaches.

This chapter also proposes a framework for supporting the development of safety-critical real-time systems that exploit hardware accelerators developed through FPGAs with DPR capabilities. With the aim of investigating this direction, here is presented a prototype implementation of a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time recurrent tasks. A model is presented and then used to derive a response-time analysis to verify the schedulability of a real-time task set under given constraints and assumptions.

Although the analysis is based on a generic model, the proposed framework has been conceived to account for several real-world constraints present on today's platforms and has been practically validated on the Zynq platform, showing that it can actually be supported by state-of-the-art technologies. Additionally, a number of experiments are reported to evaluate the worst-case performance of the proposed approach on synthetic workload.

Finally, to properly exploit the DPR feature, novel operating system supports are needed. This chapter concludes presenting an implementation of the FRED framework for the Linux operating system over the Zynq-7000 platform produced by Xilinx. Design solutions for managing hardware accelerators are discussed, and a software architecture for Linux is presented, which comprises (i) support for shared-memory communication with hardware accelerators, (ii) an improved driver to handle the FPGA reconfiguration, and (iii) a scheduler for the management of requests of hardware acceleration. The proposed solution allows exploiting the enormous number of software systems available for Linux (such as drivers, libraries, communication stacks, etc.), and the typical programming flexibility of software, while relying on predictable hardware acceleration of heavy computations.

5.1 Energy-efficient Low-latency Audio on Android

A significant limitation of the Android[1] operating system (OS) has always been the difficulty in providing[2] low-latency audio features, which are often required in a significant number of interactive multimedia applications, like professional grade multimedia. One of the obstacles in achieving this is the heterogeneity of devices running Android, having different hardware capabilities, as well as the lack of proper interaction models between applications and kernel, thus requiring several software abstraction layers and big audio buffers, both resulting in the capability of generating a smooth audio output, at the price of an audio latency increase.

Multimedia applications, as shown in Figure 5.1, can be realized on Android with many different APIs. From the Java language, a number of APIs in the `android.media` package can be conveniently used for playing or recording audio locally, managing audio files, or streaming audio from a remote source. However, this option forces the uses of large buffers, causing non-negligible audio latencies. For interactive, low-latency scenarios, applications need to possess a C/C++ component that uses the available low-latency C/C++ APIs, e.g., `OpenSL ES` or `AAudio` (more details will be provided later).

When using the low-latency audio APIs, a real-time thread using the SCHED_FIFO scheduling discipline is used, which, along with SCHED_RR, is part of the SCHED_RT scheduling class of the Linux kernel. Because of the energy requirements of the target mobile devices, SCHED_RT has been integrated with

[1] This work refers to the *master* branch of the Android Open Source Project (AOSP) synchronized on the 21st of May 2018 at 4:36pm CET, an under-development version of Android Q.

[2] Audio latency estimations can be measured with the Superpowered Mobile Audio Latency Test App, and the results are collected and shown at the following address: `https://superpowered.com/latency`.

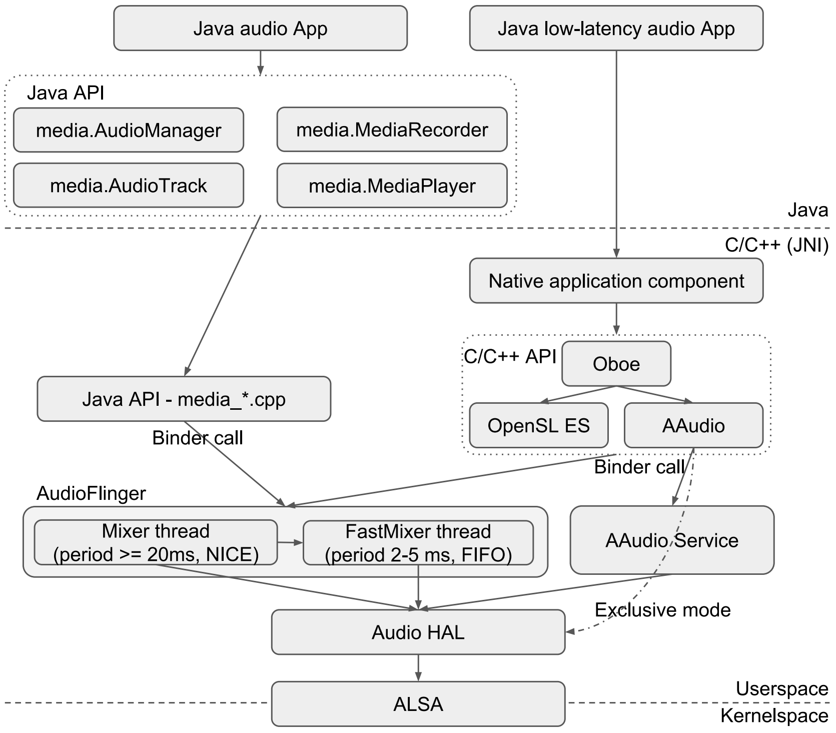


Figure 5.1: Overview of the Android audio architecture.

the `schedutil` policy of the `CPUFreq`[3] framework that, according to internal heuristics, estimates the expected overall system workload and sets the CPU frequency.

Unfortunately, this approach slowly reacts to dynamic workload changes, such as sudden spikes up of CPU demand due to, e.g., a sudden increase in the number of voices being synthesized in software. Indeed, by the time the `schedutil` heuristics realizes that the CPU workload increased, and a frequency increase is needed, it might already be too late, and the audio pipeline exhibits an audible glitch. This problem can be mitigated by increasing the audio buffer size, resulting in a larger latency of the audio processing pipeline, so that the system has more time to adapt the CPU frequency before an audio glitch happens, which ultimately makes the current solution not suitable for professional audio applications. Alternatively, low latency can be achieved by locking the CPU to the maximum frequency when a SCHED_RT task is present, as done in the mainline Linux kernel (more details in Section 5.1.2), leading to an unacceptable increased power consumption for mobile devices.

This work proposes a novel, exploratory solution to tackle this problem, based on: 1) adapting the `AAudio` framework so as to switch to a deadline-based programming paradigm which uses the SCHED_DEADLINE scheduling class recently added to the Linux kernel [112]; this, differently from SCHED_RT, allows `schedutil` to adapt the frequency coherently with the real-time workload as known to SCHED_DEADLINE, preserving the timing constraints of

[3]More information available at: `https://www.kernel.org/doc/Documentation/cpu-freq/`.

the application; and 2) by extending the `AAudio` API by introducing a new mechanism to notify the system about workload demand changes, which are used for a proactive update of the computational bandwidth, thus of the CPU frequency. Experimental results conducted over an Android board demonstrate the advantages of the proposed approach in both application responsiveness and energy efficiency.

5.1.1 Related Work

Several prior efforts exist addressing the problem of ensuring predictable execution and performance stability of multimedia applications and soft real-time workloads on general-purpose OS. These have been incorporating for decades a variety of heuristics for handling multimedia applications in the context of desktop systems, specifically for letting multimedia applications coexist with CPU-intensive processing workloads while keeping a smooth and glitch-free playback. These heuristics have been based on the concept of detecting *interactive* workloads, which are usually characterized by alternating sequences of short-lived processing and sleeping time frames, from *batch* workloads, typically having a much longer duration of CPU-intensive activities. Then, the detected interactive processes are transparently boosted in their priorities with respect to batch ones, without requiring the developers of multimedia applications to make any specific adjustment.

For example, the Linux SCHED_OTHER policy has been using a heuristic of this kind [184], tracking per-task sleep vs. ready-to-run time windows, and boosting the dynamic priority of a task after wake-up, while de-boosting it while running continuously.

Nevertheless, these mechanisms are known to result in quite an unstable allocation of the CPU to multimedia applications. So, these have been traditionally designed with large pre-computed buffers, or with the ability to adapt themselves to the resources available as typically done in video applications that skip frames as needed. However, in the case of low-latency requirements, it is common to resort to APIs for either increasing the `nice` level into the general-purpose scheduler or switching to a predictable real-time scheduling policy, typically available as the POSIX SCHED_FIFO or SCHED_RR disciplines [179].

The research literature on real-time systems proposes many approaches focused on CPU scheduling and the application of reservation-based scheduling techniques [162], that also apply to soft-real time architectures [103], based on adaptive approaches, by which the OS kernel guarantees timely allocation of the CPU to competing applications according to their requirements, including their specific time granularity. These constitute a basis for stable task execution on top of which sound mathematical models can be built.

Several authors also have considered energy-aware adaptive scheduling of soft real-time and multimedia applications. In [126, 180] and [4], authors propose to use theoretical arguments from control theory to the problem of adaptively scheduling real-time tasks, based on a closed-loop model of the system rooted in linear systems theory, which is made possible by adopting an underlying reservation-based scheduling discipline. Further research along this line considered the adaptation of per-task resource reservations recurring to non-linear controllers [58], probabilistic real-time guarantees [57], and pipelines of tasks with end-to-end deadline guarantees [149].

In GraceOS [201] the Linux kernel is enhanced with the ability to monitor cycle demand distributions of applications, and use them to adaptively configure scheduling parameters and frequency switching of the platform to meet the application quality of service requirements, as well as the system overall energy consumption constraints. Similar in objectives was the work in [64], where an adaptive multi-layer control architecture for energy-aware soft real-time was proposed. This included adaptiveness: 1) at the application level, where multi-mode applications had the capability to dynamically switch among a set of discrete modes of operation with different resource and timing requirements; 2) at the middleware level, where a per-application feedback-scheduling controller was attached to each application to control its needed scheduling parameters; and 3) at the OS/kernel level, where a supervisor was performing overload controls, and a centralized QoS/power manager was dynamically re-optimizing the whole system configuration in a distributed real-time environment.

More recently, the `powernowd` daemon for automatic CPU frequency control on Linux has been modified [61] to prevent switching to power modes that would break schedulability of a system that has already accepted real-time reservations, on the single-processor scheduler [150] mentioned above.

Concerning audio applications on Linux, experimentation with deadline-based real-time scheduling has been conducted [62] using the JACK[4] low-latency audio infrastructure. JACK was modified to use AQuoSA [150], an old single-processor implementation of a Constant Bandwidth Server [6] (CBS) based on EDF scheduling for Linux; however power management and multicore scheduling were not considered in that study, and only history-based runtime adaptation was investigated.

A different view on power management is offered by the Q-RAM framework [86, 161], which allows to model resource allocation in real-time systems as an optimization problem, allocating various kinds of resources (memory, CPU, network bandwidth) to tasks so that some cost functions are minimized (or utility functions are maximized). Using Q-RAM, power management can be modeled by considering the consumed power as a cost.

Focusing on Android, prior research literature also exists, addressing real-time issues, priority inversion and other performance-oriented aspects of the OS. A form of priority inversion mitigation [113] has been integrated for a long time within the Binder IPC framework, extensively used throughout Android applications and services, to preserve the `nice` level of the calling thread and, more recently, of its real-time priority for real-time tasks [100], across synchronous remote procedure calls (RPCs)[5].

Further proposed modifications for enhanced real-time support in Android include [99, 196, 198], adopting a real-time Java Virtual Machine run-time platform to control interferences due to the garbage collector, enhancing the memory allocator, and improving the accessibility of scheduling services from unprivileged applications, without the need for a Binder call to a privileged process. Besides these, other works [197] have been proposed to extend the Android interfaces for the development of soft real-time applications, by introducing statically specified memory bounds and priority awareness.

[4]More information available at: `http://jackaudio.org`.

[5] For details, refer to commit history of `binder.c` as available at: `https://android.googlesource.com/kernel/common/+/android-4.9/drivers/android/binder.c`.

Compared to the existing prior literature on adaptive real-time, power-aware scheduling for multimedia and soft real-time applications, the present work is the first one focusing on the Android operating system, combining the new SCHED_DEADLINE real-time scheduler recently made available within the Linux kernel with the `CPUFreq` power management subsystem and its `schedutil` governor, considering an underlying heterogeneous processing architecture with CPUs having different processing capacities (e.g., Arm `big.LITTLE`), and introducing a proactive approach that lets applications declare in advance their expected workload changes, so that scheduling parameters adaptations can be anticipated, resulting in an effective capability to keep a low-latency glitch-free playback in the presence of heavy fluctuations of the processing demand, while at the same limiting power consumption to the minimum.

5.1.2 Background

This section provides background information on the Android audio pipeline architecture used to enable low-latency features for audio applications, how Android tries to achieve energy efficiency while scheduling latency-sensitive tasks, and deadline-based scheduling features in the Linux kernel, that are at the foundation of the presented solution.

Android Audio Architecture

Audio applications can be developed in Android by using, as shown in Figure 5.1, the high-level Java APIs available through the set of `android.media.*` classes, such as: `media.MediaPlayer`, to control playback of audio/video files and streams towards the local devices; `media.MediaRecorder`, to record audio/video from the local devices; `media.AudioManager`, to control volume and other playback tunables; `media.AudioTrack` to handle buffering of audio samples for playback; `MediaCodec`, to handle a plethora of available codecs for media playback and streaming. These Java classes interact via the Java Native Interface (JNI) with their C/C++ counterparts, which access the available audio services by interacting with the `AudioFlinger` server through Binder. Within `AudioFlinger`, up to 32 different audio streams coming from different applications are mixed by the `Mixer` thread, normally activating at a period greater than 20 ms, and having the ability to perform complex adaptations such as sample rate conversions. To avoid possible audio glitches, this thread runs with a boosted priority (using a negative `nice` value) within the SCHED_OTHER scheduling class. The `Mixer` thread hands over the audio samples to the underlying userspace Android audio Hardware Abstraction Layer (HAL), which ultimately sends the data to the device drivers through the `ALSA` sub-system within the Linux kernel.

Developers willing to realize low-latency, interactive audio applications, typically have to make the extra step of implementing a JNI component in their application, and use directly the available low-latency C/C++ audio APIs, either the traditional `OpenSL ES` (available since Android 4.1), or the recently added `AAudio` (available since Android 8.1), or the further `Oboe` over-arching API that is capable of using either the former or the latter API. These APIs interact with a particular component of the `AudioFlinger` server, called `FastMixer`, that has a much shorter activation period, normally 2–5 ms, which in turn hands over

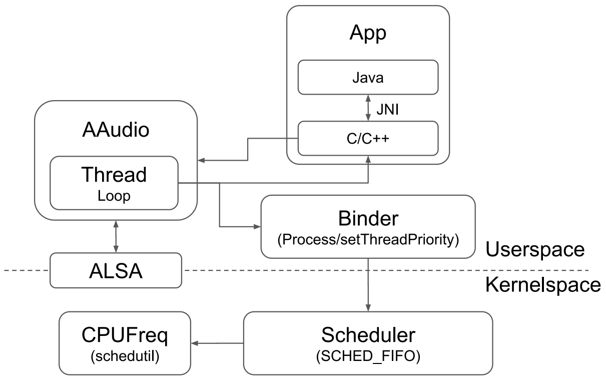


Figure 5.2: Logical blocks involved in a low-latency Android `AAudio` playback.

audio data to the audio `HAL` and ultimately to the kernel. In order to ensure a glitch-free playback under such conditions, `FastMixer` keeps its functionality at the bare minimum (i.e., up to only 8 tracks can be mixed, one of which is the audio coming from the regular non-low latency path, and no resampling is supported), and it has a real-time thread scheduled using the SCHED_FIFO policy. In particularly demanding cases, it is possible to bypass the `FastMixer` entirely, so avoid its overheads, by requesting exclusive access to the audio device, when initializing `AAudio`.

In what follows, the focus of this chapter is on building low-latency audio applications making use of the `AAudio` framework, particularly for the specific case of exclusive access to the audio device[6].

Low-latency Audio Pipeline in Android

The low-latency audio pipeline in Android has the structure exemplified in Figure 5.2. A typical Java application that wants to use the low-latency audio pipeline must use JNI to define the callback which generates the audio stream, and must export the callback to the audio stream framework through the provided API. When the audio stream starts, `AAudio` generates a new thread and sets its scheduling class to SCHED_FIFO through a Binder call. This thread loops forever executing an application-supplied callback, which produces the audio frames for playback through the `ALSA` subsystem. An audio frame contains a sample for each available channel (e.g., for stereo playback, a frame has two samples, for the left and right channels). Instead of using a blocking operation on the device (i.e., `select`, `poll` or `epoll`), the looping thread sleeps for an amount of time determined by a timing model that wakes up the process when there is sufficient room in the audio buffer to be refilled.

The application callback writes audio frames in chunks called *bursts* of size b, that are queued into the playback ring-buffer of the audio device, which has a bigger size B set as a multiple of the *burst* size: $B = k \cdot b$, $k \in \mathbb{N}^+$.

To ensure a smooth and glitch-free playback, the playback ring-buffer is kept as full as possible, as exemplified in the sample scenario depicted in Figure 5.3. The application has typically a ramp-up phase at the beginning of the playback

[6]A design suitable for the general case that includes an arbitrary processing graph of computations producing the audio stream is still an open challenge.

Symbol	Value (default)	Value (low-latency)	Description
S	48	48	Sampling rate (kHz)
b	480[7]	64	Audio burst size (samples)
B	7680[7]	128–192	Audio buffer size (samples)
r_t	160	2.67–4	Audio latency (ms)

Table 5.1: Typical audio parameters.

(the first 4 bursts in the figure), during which the full B-sized buffer is filled up, followed by a nearly periodic activation, with additional b audio frames provided by the callback at each activation (as visible for bursts 4, 7, 8, 9 in the figure). With a configured sampling rate of S frames per second for the audio device, this results in one activation of the application callback every period of length b/S. However, B is the parameter which directly affects the audio latency, because the residency r_t of an audio sample in the device ring-buffer is: $r_t = B/S$ (as visually highlighted in the figure for burst 4).

When the real-time audio thread wakes-up, it may be scheduled immediately (i.e., at time $6b/S$ in the figure), or its execution may be postponed by the scheduler due to other activities on the system, like non-preemptible kernel sections, serving interrupts or scheduling higher priority tasks. As a consequence, the real-time thread may fail to compute its next burst within its next activation time. In this case, the in-kernel buffer fill-level goes lower than usual; however, this does not immediately result in an audio glitch thanks to the buffered additional bursts (highlighted in the figure for burst number 5). An audio glitch only occurs if the audio processing is delayed further, up to the hard deadline of B/b periods.

Typical values of these parameters are reported in Table 5.1 for non-interactive playback scenarios, as well as low-latency audio applications.

Power Management on Linux/Android

Since Android 8, the default `CPUFreq` frequency-scaling governor is `schedutil`, which adjusts the CPU frequency according to the utilization statistics computed by the Window-Assisted Load Tracking[8] (`WALT`) or, in some experimentations, the Per-Entity Load Tracking[9] (`PELT`) algorithm. These utilization metrics are either not present (`WALT`), or not used (`PELT`) from the mainline `schedutil` governor when it comes to choose the frequency to run a SCHED_RT task. For these tasks, `schedutil` always sets the maximum available frequency. The mainline behavior is particularly inefficient for system-on-a-chip (SoC) architectures that organize processors in clusters, where cores of the same cluster share the same frequency: even a single SCHED_RT task on a single core forces the whole cluster at the maximum frequency.

As different devices have different capabilities in CPU frequency scaling and power management (e.g., big.LITTLE[10], DynamIQ[11]), the EAS (Energy Aware

[7]Device-specific values.
[8]https://lwn.net/Articles/704903/.
[9]https://lwn.net/Articles/531853/.
[10]More information available at: `https://developer.arm.com/technologies/big-little`.
[11]More information available at: `https://developer.arm.com/technologies/dynamiq`.

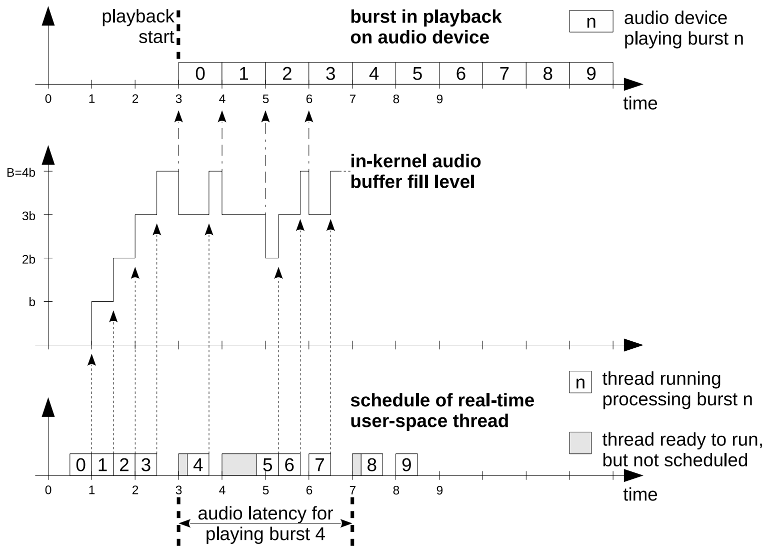

playback start
burst in playback on audio device
n audio device playing burst n
time
in-kernel audio buffer fill level
B=4b
3b
2b
b
schedule of real-time user-space thread
n thread running processing burst n
thread ready to run, but not scheduled
audio latency for playing burst 4

Figure 5.3: Exemplification of audio processing pipeline, showing the schedule of the real-time application thread processing each burst (on bottom), the fill level of the audio ring-buffer within the kernel (central) and the audio burst under playback at each time instant (on top). Time on the x axis is expressed as multiples of b/S.

Scheduling) framework has been recently added to the Linux kernel to provide a unified view of these capabilities. EAS manages metadata about the frequency clusters topology, along with information about power consumption and processing performance (called *capacity*) for each Operating Performance Point (OPP) of each CPU. EAS also includes mechanisms for exploiting this information at its best regarding task placement and OPP selection, based on the current workload.

Since the CPU frequency can only be chosen among a set of predefined OPPs imposed by the hardware and, sometimes, further reduced by software constraints[12], the process of translating the utilization to the CPU frequency is performed by selecting the smallest OPP capable of satisfying the computational demand.

Deadline-based Scheduling in Linux

The SCHED_DEADLINE scheduling class has been introduced in the Linux kernel version 3.14, and implements a deadline-based scheduling algorithm, tailored for the management of real-time tasks requiring temporal guarantees on their execution times. SCHED_DEADLINE also provides temporal isolation among tasks by implementing the CBS algorithm.

More precisely, SCHED_DEADLINE realizes a reservation-based approach for CPU scheduling of real-time workloads, where a SCHED_DEADLINE task τ_i is associated with three scheduling parameters: a `runtime` Q_i, a `deadline`

[12]The vendors often provide access to a subset of the hardware OPPs depending, for example, on thermal dissipation capabilities, or energy availability of the final product.

Symbol	Description
m	Number of CPUs in the system
n	Number of real-time tasks in the system
Γ	Set of considered real-time tasks
Γ_j	Set of real-time tasks on CPU j
Q_i	SCHED_DEADLINE runtime for task i
D_i	SCHED_DEADLINE relative deadline for task i
P_i	SCHED_DEADLINE period for task i
C_i	Worst-case per-activation execution-time of task i
T_i	Minimum inter-arrival period of task i

Table 5.2: Parameters characterizing real-time periodic tasks and SCHED_DEADLINE reservations.

D_i and a `period` P_i. This results in the Linux kernel granting the task Q_i time units on the processor every time window of duration P_i, where in each period the Q_i time units of execution are granted within the relative deadline D_i.

Focusing on the standard case of periodic real-time tasks with relative deadline equal to the period, the typical use of SCHED_DEADLINE is the one of a task with known minimum inter-arrival period T_i and worst-case per-activation execution time C_i, where one would set its CBS scheduling runtime as $Q_i = C_i$ and the scheduling period and deadline as $P_i = D_i = T_i$.

On multiprocessor systems, SCHED_DEADLINE implements a *global* EDF-based scheduling policy, but it can also be configured as a *partitioned* scheduler by proper use of the `cpuset`[13] CGroup controller. On SMP systems with CPU frequency locked, the global configuration of SCHED_DEADLINE guarantees each task in a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ to complete with a well-known worst-case tardiness beyond its relative deadline [69, 189], as long as the system capacity is not violated:

$$\sum_{i\in\Gamma} \frac{Q_i}{P_i} \equiv \sum_{i\in\Gamma} \frac{C_i}{T_i} \leq m, \tag{5.1}$$

with m being the number of CPUs. On the other hand, with the partitioned configuration, SCHED_DEADLINE guarantees each task to respect its relative deadline, as long as the capacity of each CPU $j \in \{1, \ldots, m\}$ is not violated:

$$\forall j \in \{1, \ldots, m\}, \sum_{i\in\Gamma_j} \frac{Q_i}{P_i} \equiv \sum_{i\in\Gamma_j} \frac{C_i}{T_i} \leq 1, \tag{5.2}$$

with $\Gamma_j \subseteq \Gamma$ denoting the tasks on CPU j.

The just introduced notation, summarized in Table 5.2, will be extensively used throughout the chapter.

Furthermore, since version 4.16, SCHED_DEADLINE has been integrated with `schedutil` [171] by using the power-aware variant of the Greedy Reclamation of Unused Bandwidth algorithm [170] (GRUB-PA). As a result, the scheduler is now able to scale the CPU frequency according to the bandwidth requirements of the SCHED_DEADLINE tasks, limiting the power consumption while preserving their timing constraints.

[13]More information available at: `https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt`.

In more details, the GRUB scheduling algorithm [117] can estimate the *active utilization* of the SCHED_DEADLINE tasks running in the system (informally speaking, this is the fraction of CPU time used by such tasks), and this information can be used to scale the CPU frequency so that the SCHED_DEADLINE tasks use exactly the specified fraction of CPU time. The original GRUB and GRUB-PA algorithms were uni-processor only, but the reclaiming and bandwidth accounting mechanisms have been extended to support multiple CPUs/-cores [7].

5.1.3 Adaptive Reservations on Android

This details the presented proposal for engineering low-latency audio applications on the Android platform, which makes use of the SCHED_DEADLINE real-time scheduler, with `GRUB-PA` extensions, along with the `schedutil` governor. Second, dynamic audio workloads are tackled by proposing an extension to the `AAudio` API with a method that allows applications to declare their time-varying workload demand. The approach is implemented and validated with the use of `SynthMark`[14], a recently available open-source audio pipeline emulator for Android.

Deadline-based Scheduling for Low-Latency Audio

This work focuses on low-latency audio applications making use of a single, sequential, non-suspending callback, and the real-time thread spawned by `AAudio` that executes the callback, then blocks until there is room in the audio buffer. The application may have other non real-time threads for its API, but since they are not part of the playback latency sensitive path, they are ignored in this work. This application structure represents a typical model for the majority of low-latency, interactive audio applications that are nowadays developed.

As explained in Section 5.1.2, the callback is activated every time the audio buffer has enough space to allocate at least a burst. This, coupled with the attempt to keep the buffer as full as possible, results in the activation of the real-time audio thread at every period of duration $T = b/S$. Each such activation needs to complete ideally before the next activation period: when a burst is put in playback onto the device, another burst is requested to be produced. Therefore, each activation of the real-time thread has a relative soft deadline of $D = b/S$, where the hard deadline is B/S instead.

It is thus possible to apply the implicit deadline task model, as extensively studied in the real-time literature, and specifically, the CBS algorithm setting the scheduling period and deadline equal to $P = D = b/S$, and setting the scheduling runtime equal to the task worst-case execution time $Q = C$. The execution time C, needed for the processing of each callback activation, is a value much harder to characterize. It depends on: 1) the computational capabilities of the hardware architecture where the application is running; 2) the maximum capacity of the CPU the real-time thread is scheduled on, whenever CPUs can have asymmetric performances like in the case of Arm `big.LITTLE` architectures widely adopted by mobile Android devices; and 3) the CPU frequency on architectures supporting DVFS (Dynamic Voltage and Frequency Scaling), as dynamically adapted by the scheduler and `CPUFreq` subsystem.

[14]https://github.com/google/synthmark/.

The intrinsic dynamics of the application workload has also to be considered. Even on a CPU running at a constant frequency, an application can have very different computational demands. This may depend, for instance, on the number of musical notes played by the user for a virtual instrument, or the number and complexity of active audio filters/effects for an audio processing application. Taking as a reference a virtual instrument synthesizer, such as a virtual piano, because of the use of a sustain pedal or because of the notes decay, the number of notes that are simultaneously played can dynamically vary in a typical range between 0 to a few hundreds, resulting in a callback workload that undergoes quick variations spanning across multiple orders of magnitude.

The computational requirement C is a fundamental parameter, that can be used to: 1) analyze in advance and guarantee the schedulability of the tasks set according to the real-time theory, 2) allocate the proper amount of resources to the process, and 3) determine the most efficient CPU frequency and, for heterogeneous architectures, the most efficient processor the task shall run on. The next section outlines how all these challenges are tackled.

Adaptive Scheduling

To schedule the audio callback, Android currently uses SCHED_FIFO along with either `PELT` or `WALT` as signals to drive the OPP selection via `schedutil`.

One of the main limitations to using this approach for low-latency audio applications is related to the reactive OPP selection policy these signals enforce. Indeed, when either `PELT` or `WALT` are in use, to trigger a frequency change when the audio workload suddenly increases, it takes some time for the signal to detect the new CPU bandwidth requirement correctly. In the worst case, it could take between 50 ms and 100 ms for `PELT` (depending on its configuration) to detect a 90% increase of the CPU bandwidth demand. For a low-latency audio scenario where $B = 256$ and $b = 64$ at $S = 48\ kHz$, the `PELT` detection latency is too high when compared to the buffered audio of only 5.33 ms. Even when `WALT` is in use, a 90% CPU utilization demand increase cannot be detected in less than 10 or 20 ms, depending on its configuration. Thus, independently from the utilization signal source in use, with just kernel space driven OPP selection, it is hard to grant a glitch-free playback meeting the requirements of a low-latency audio application.

The reactive nature of CPU frequency driving from kernel space cannot be easily fixed without explicit hints from userspace. This is why a possible solution has been so far to control the frequencies from userspace by explicitly setting constraints on the minimum frequency which can be selected by the kernel. Such approaches, despite being effective, are however still missing a specific API to define bandwidth requirements in a platform-independent way. Platform independence is of paramount importance when it comes down to build portable applications which can still satisfy tight temporal constraints independently from the specific platform on which they will be executed.

To tackle all these problems, this chapter proposes to extend the `AAudio` API to allow the application to notify the underlying OS about a workload change. This is done by dynamically supplying a `workUnits` parameter, whose variations correlate with the expected workload changes of the real-time thread callback. The `schedutil` governor can then immediately trigger an up-frequency switch, even before starting the heavier computations, resulting in the needed CPU frequency increase being completed as fast as possible to efficiently and adequately support the increasing workload demand. This mechanism allows audio applications to inform the OS about expected workload changes so that the system can adjust the OPP to a value that satisfies the QoS requirements of the application, preserving the critical objective of minimizing the energy consumption. Since the same audio application can be installed in a multitude of different devices, the `workUnits` hint must be considered as an indication on the status of the application, independent from the computing architecture and the system behavior.

The translation from `workUnits` to the actual OPP can be designed by preserving the SCHED_FIFO scheduling policy, but forcing the `CPUFreq` governor to use a userspace defined utilization for the given task, that would directly be translated to a minimum CPU frequency value. The Linux kernel does not support this `schedutil` extension yet, but it is currently under discussion on LKML[15].

In this chapter, an alternative approach is undertaken, based on the use of the SCHED_DEADLINE scheduling class, as shown in Figure 5.4. The `AudioTask` notifies the currently requested `workUnits` to the `HostCPUManager` which, extending the concepts from [3, 61], internally uses a `Predictor` to estimate the upcoming computing time of the callback according to statistics of the past observed processing times, and by modulating the obtained value through the supplied `workUnits` values.

To produce valid heuristics, the `Predictor` requires reliable and precise measurements on the callback runtime. Measuring this time by executing the `clock_gettime()` syscall before and after running the callback, and finally subtracting the two results is not a reliable approach. In fact, the final value can be affected by both the CPU frequency changes happened during the callback execution, and the capacity of the CPUs where the task executing the callback has been executed, e.g., a `big` or `LITTLE` CPU. Both these problems have been solved by extending the `sched_getattr()` syscall to return the next absolute deadline d and the remaining CBS server runtime q, where the scheduler already scales this last value according to the CPU frequency and capacity. These values can be sampled before and after the callback execution, thus, by also knowing the actual CBS computing time Q that the task can use every period T, it is possible to compute the *normalized* execution time C_m of the callback as:

$$C_m = q_{\text{start}} - q_{\text{end}} + Q \left\lfloor \frac{d_{\text{end}} - d_{\text{start}}}{T} \right\rfloor . \tag{5.3}$$

Here, the rightmost term is almost always zero: whenever the callback completes within its relative deadline $D = b/S$, according to the CBS rules the absolute deadline is not updated, so $d_{\text{end}} = d_{\text{start}}$. For those rare cases in which the callback completes beyond its (soft) deadline (e.g., burst 5 in Figure 5.3), at the end of the callback, the CBS algorithm will have recharged the budget of a quantity equal to Q for each deadline postponement of the scheduling period $P = T$.

After each task activation callback completes, its normalized runtime C_m is measured, and the `Predictor` updates its stored runtime statistics. The estimated runtime for the next activation $k+1$, $C_e\left[k+1\right]$, is computed by

[15]https://lwn.net/Articles/751361/.

implementing an exponentially weighted moving average:

$$C_e[k+1] = \alpha C_m + (1-\alpha) C_e[k], \tag{5.4}$$

with asymmetric smoothing constants α: if the C_m value is bigger than $C_e[k]$, $\alpha = 0.95$ is used, otherwise $\alpha = 0.1$ is used. This helps to react to workload underestimations quickly and slows down the decreasing of workload estimation. The predictor stores an independent moving average for every observed `workUnits` value within a hashmap, where at each callback completion the stored runtime value associated to the current `workUnits` is updated according to Equation (5.4). The advantages of using a hashmap are that it does not require prior knowledge of the minimum or maximum value of `workUnits` (as an array would need), and guarantees efficient access and modify operations. When the `Predictor` is asked to estimate the next activation computing time, the following cases may happen:

- the hashmap is empty: the maximum computing time, corresponding to a configurable constant bandwidth is returned. For example, 0.94 has been used in these experiments.
- `workUnits` found in the hashmap: the value stored in the hashmap is returned.
- `workUnits` smaller than any other value in the hashmap: to be conservative, the duration for the smallest `workUnits` value in the hashmap is used.
- `workUnits` bigger than any other value in the hashmap: the returned computing time is computed by intersecting the linear regression among the available elements in the hashmap. In absence of any other information on the processing workload, it is made the assumption that the callback computing time is likely a linear function in the number of `workUnits`, for simplicity.

The pessimism of this algorithm helps to prevent audio glitches and the bad energy performance is limited to the application start time, when the workload statistics are not available to the `Predictor` yet.

The `BandwidthAllocator` receives the latest estimated C_e value, which is used to update the SCHED_DEADLINE runtime Q. Since the predicted C_e derives from a slightly modified average value of the previously measured runtimes, some margins are added to the value finally used to update the scheduler:

$$Q = m_m C_e + m_o, \tag{5.5}$$

where $m_m \geq 1$ is a configurable proportional margin and $m_o \geq 0$ is an offset. This operation is performed every time the application updates the `workUnits` number, and after a given number of written bursts, e.g., 30 in these experiments. Thanks to the use of the GRUB-PA policy, the new SCHED_DEADLINE workload is immediately communicated by the scheduler to `schedutil`, which takes care of updating the CPU frequency accordingly.

Relying only on the callback computing time C_e and heuristics provided by a Predictor to estimate the computational time depending on the `workUnits`, introduces the need for choosing a proper trade-off between stability and reactiveness of the adaptation.

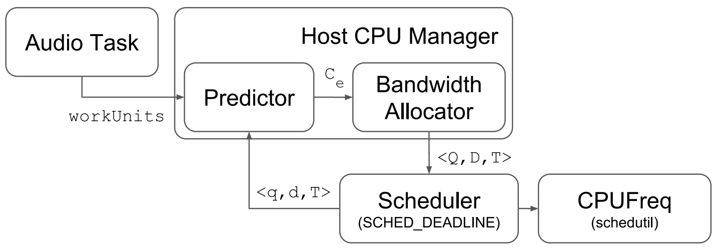


Figure 5.4: Dynamic bandwidth allocation: logical blocks.

SynthMark

SynthMark is a benchmarking tool that contains a real audio synthesizer generating an audio stream that, instead of being sent to the real Android audio pipeline, is sent to a virtual audio pipeline and consumed by a virtual audio sink, both embedded in the tool. The internals of SynthMark have been designed for realistic emulation of the behavior of an Android audio application using the real low-latency audio pipeline features. Using an emulator like SynthMark provides more flexibility for Android OS developers when customizing the system regarding configurations, programming models, and scheduling algorithms, and allows for obtaining detailed and comprehensive performance statistics so that it becomes easier to identify the weaknesses of the approaches under development. Being SynthMark platform independent, it also allows for measuring the effects of the OS on latency, independently of the Android audio framework.

As shown in Figure 5.5, what in Android would be the audio application, in SynthMark is implemented by the Synth module that is, in fact, a real polyphonic audio synthesizer that produces real audio samples generated with a chain of oscillators, filters and ADSR (Attack, Decay, Sustain, Release), and whose computational effort depends on the number of requested notes. The Synth module exports a callback that is quasi-periodically executed by the VirtualAudioSink module, which implements a thread whose scheduling parameters can be modified and, by default, on Linux kernel based systems, uses SCHED_FIFO. After the application callback is executed, VirtualAudioSink performs a write operation on the (virtual) audio sink to notify the availability of new audio samples, resulting on a blocking operation that waits until the buffer has enough space to contain the newly produced data. VirtualAudioSink also counts underruns occurring whenever the audio buffer gets empty.

After every callback execution, the VirtualAudioSink measures its duration and forwards this value, together with the current number of notes played, to the underlying HostTools component, to keep track of workload changes. HostTools takes care of storing callback statistics and adjusting the bandwidth through internal heuristics. It will be described and evaluated in the next section.

5.1.4 Experimental Results

To evaluate the proposed solution and compare it with the current Android approach on both energy efficiency and audio latency, it has been performed a

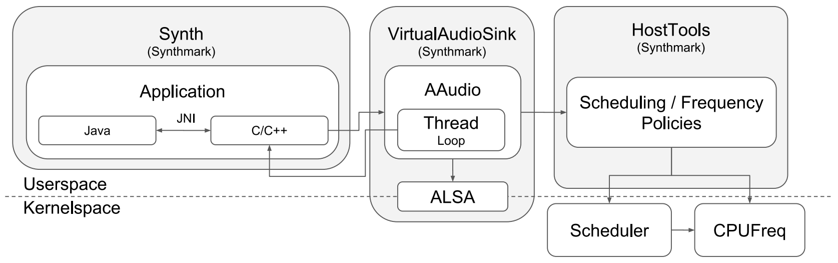


Figure 5.5: `SynthMark` logical blocks in relationship with the Android low-latency audio pipeline.

set of experiments running the extended `SynthMark`[16] tool on an HiKey 960[17] target board running the presented extended Android Linux kernels[18] connected to an ACME CAPE[19] energy meter. On the software side, the `SynthMark` tool has been used by pinning it to a `big` CPU. This was done to ensure fair comparisons by ruling out possible side-effects due to tasks migrations and CPU capacities variations among different cores.

In a preliminary calibration phase, the `VoiceMark` benchmark of `SynthMark` has been used to evaluate the total CPU utilization for a given number of active audio voices. It results that 210 voices were able to saturate the `big` CPU when running at the maximum frequency.

The following experiments used the `LatencyMark` benchmark of `SynthMark`, with a burst size of $b = 64$ and a number of `workUnits` that dynamically varied every 3 seconds.

A first experiment was aimed at evaluating the total audio latency required when using the current Android low-latency approach based on SCHED_FIFO and `WALT`, with its default window size of 20 ms. This experiment consisted of alternating the number of voices from 5 to 185, corresponding to utilizations of the `big` CPU ranging from 2.38% to 88%. As shown in Figure 5.6a, the `WALT` utilization followed the real demand as due to the increase in the number of voices, reaching the final steady state with a delay of approximately 80 ms because of the `WALT` window size, which also postponed the frequency update. In this example, the system was able to reach the proper OPP after about 60 ms after the workload increase, about 20 ms before the steady state condition, since the CPU utilization sampled at that time already corresponded to the final OPP. Because of this frequency adaptation delay, when the application load increases, the system is not capable of instantly satisfying its computational requirements, thus generates audio glitches with relatively small buffer sizes.

With the previously described configuration, a buffer size of 20 bursts was required for smooth audio playback, that on the other hand introduced an audio latency of approximatively 26.67 ms. When the workload decreases instead, the system kept an OPP higher than required for a small amount of time, resulting in a modest waste of energy.

[16]`https://github.com/balsini/synthmark/tree/JSS-2018`.

[17]More information available at: `https://www.96boards.org/product/hikey960/`.

[18]`https://github.com/balsini/linux/tree/JSS-2018-android-hikey-linaro-4.9`, and `https://github.com/balsini/linux/tree/JSS-2018-android-hikey-linaro-4.9-dl-integration`.

[19]More information available at: `https://baylibre.com/acme/`.

The next experiment was run with `WALT` disabled and shows how, by dynamically adapting the task bandwidth according to the number of `workUnits` and the callback durations, SCHED_DEADLINE can automatically adjust the OPP. This experiment has the same configuration previously shown, with a periodic `workUnits` variation between 5 and 185. The safe margins of the `BandwidthAllocator` have been tuned for the device and are set as $m_m = 1.005$ and $m_o = 0.041$. Figure 5.6b presents the experimental results for this scenario. In this case, since `WALT` was not active, `schedutil` evaluated the required CPU bandwidth as a sum of the `PELT` utilization reported by the SCHED_NORMAL (CFS), SCHED_FIFO and SCHED_RR (RT), and SCHED_DEADLINE (DL) scheduling classes. In these experiments, `SynthMark` was the only application running on the system; thus, the sampled `PELT` utilizations had almost no interference from other tasks, except interrupt handlers or background activities. This plot shows that the new deadline-based approach with dynamic bandwidth reservation was able to raise the system utilization in less than 0.01 ms and the frequency adjustment is requested in approximatively 0.054 ms after the workload change. By summing the time required by the CPU to switch the OPP on the HiKey 960 platform, this approach requires about 0.45 ms to complete the frequency adaptation after the workload increased.

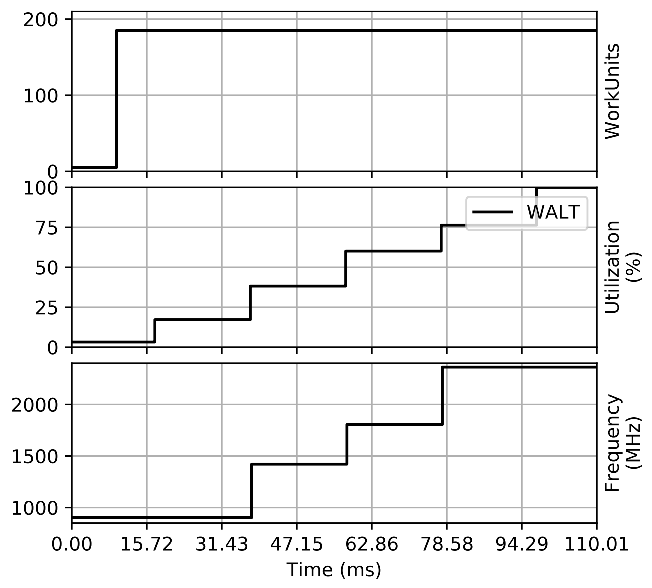
Since the OPP update is almost immediately performed before the workload change, the callback runs at a CPU frequency that satisfies its timing requirements. The profitability of this behavior for the reactiveness of the audio task was demonstrated by `LatencyMark`, which returned that the system is stable with a buffer size of 2 bursts, corresponding to 2.67 ms.

In order to evaluate the quality of this approach in more generic and realistic workload scenarios, the following experiments were run to evaluate SCHED_FIFO with `WALT` and the adaptive bandwidth allocation with SCHED_DEADLINE approaches by running `LatencyMark` with a random number of `workUnits` at every step, still in the range between 5 and 185, for a total duration of 60 seconds. The seed of the random number generator has been fixed to allow the reproducibility of the experiment and to provide the same `workUnits` random sequence when comparing the two approaches.

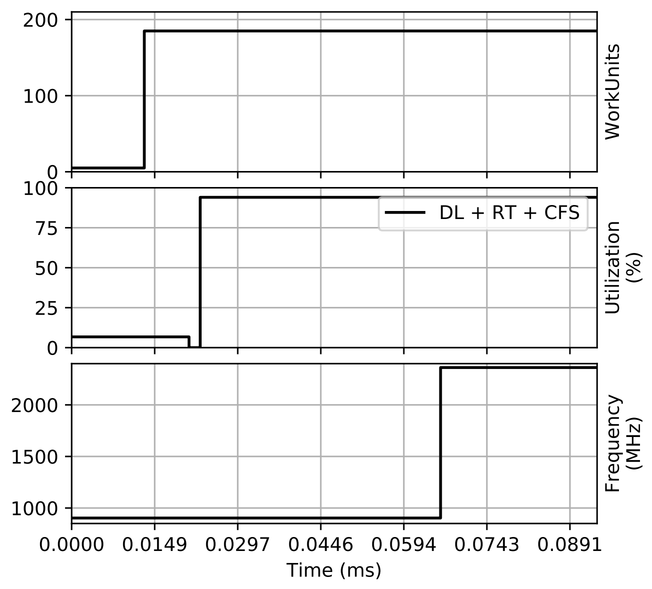
Figures 5.7a and 5.7b show the tight relationship between the CPU frequency and the number of `workUnits` in both the SCHED_FIFO with `WALT` and the adaptive bandwidth allocation with SCHED_DEADLINE. If the adaptive bandwidth allocation approach can keep the audio latency at 2.67 ms, on the other hand, the estimation of the utilization is more pessimistic than the one performed by `WALT`.

For the sake of completeness, it has also been tested the current solution implemented by many low-latency audio application developers, that maintains a high CPU frequency on top of SCHED_FIFO with `WALT` by fixing the number of `workUnits` to a static value. Concerning the audio latency, this solution achieved the same results of the adaptive bandwidth allocation approach, reducing the delay to 2.67 ms, but forces the CPU to run continuously and at a fixed OPP, also when not required.

The pessimism of the adaptive bandwidth allocation with SCHED_DEADLINE approach reflects on the CPU frequency selection that, as shown in Figure 5.8, tends to prefer higher CPU frequencies compared to SCHED_FIFO using `WALT` with dynamic `workUnits`, while the fixed workload approach always uses the maximum OPP. These numbers directly affect the energy efficiency, which can

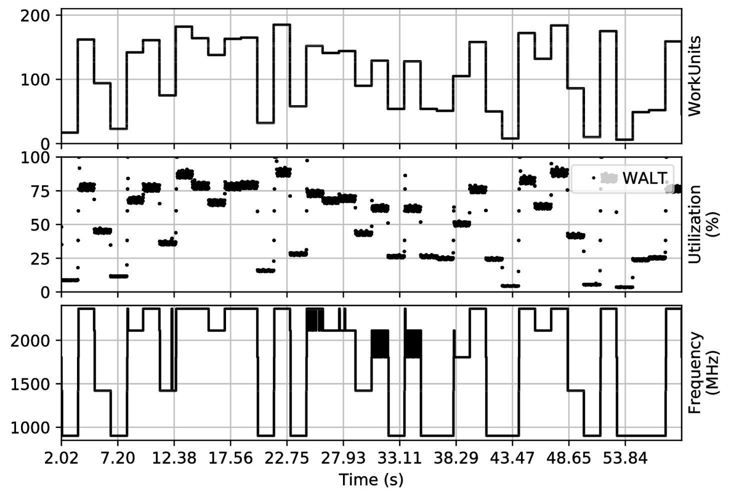


(a)

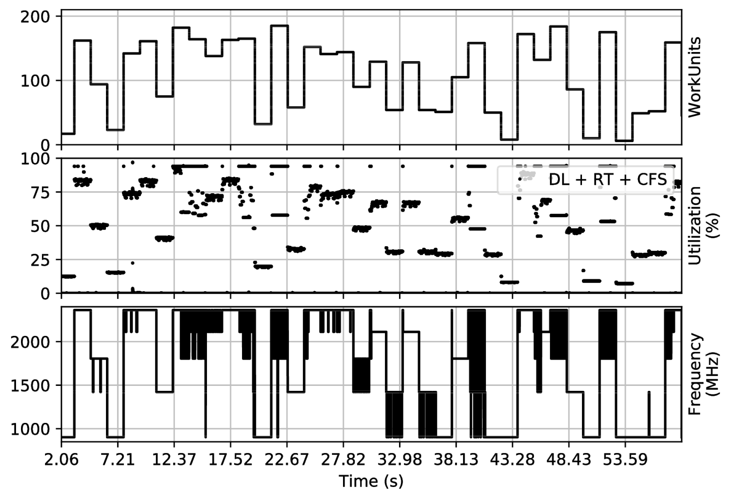


(b)

Figure 5.6: `SynthMark` utilization perceived by `schedutil` using SCHED_FIFO with `WALT` (5.6a) and by the `Predictor` (5.6b), and frequency adjustment at the varying the number of `workUnits`.



(a)



(b)

Figure 5.7: `SynthMark` behavior using SCHED_FIFO with `WALT` (5.7a) and with adaptive bandwidth allocation (5.7b), at the varying of a random number of `workUnits`.

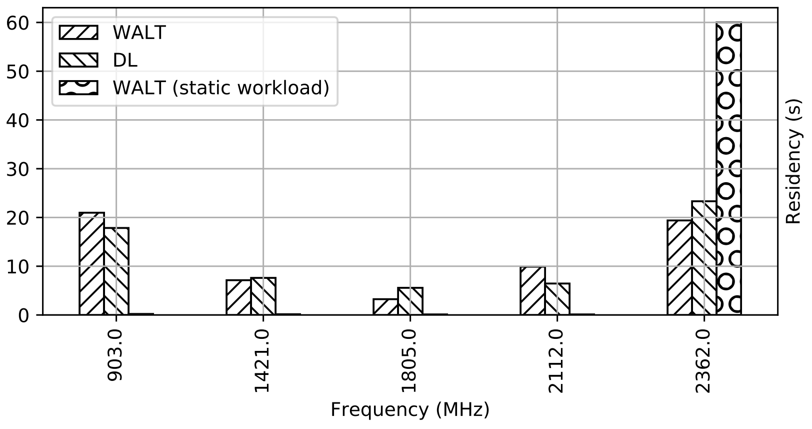


Figure 5.8: Frequency residency comparison between the `WALT` and the adaptive bandwidth allocation approaches.

be evaluated at the single core level by using the official, normalized energy consumption metrics[20], that are also used by the kernel to find the most efficient task placement among the CPUs. With these normalized metrics, it turned out that when using SCHED_FIFO with `WALT`, the energy consumption of the experiment was approximatively 96.16, while fixing the `workUnits`, the energy consumption reached about 171.01. Using the adaptive bandwidth approach using SCHED_DEADLINE had an energy consumption of 102.17 instead.

Finally, it has also been measured the total energy consumption of the HiKey 960 board with the ACME CAPE energy meter directly connected to the board power supply, to compare the efficiency of the different approaches. It turned out that when using SCHED_FIFO with `WALT`, the total board energy consumption of the experiment was 206.35 Ws (Watt $\cdot$ second) while, setting a static number of `workUnits`, the total energy consumption reached about 256.93 Ws. With the adaptive bandwidth approach using SCHED_DEADLINE, the board had a total energy consumption of 210.22 Ws instead.

As summarized in Table 5.3, the proposed adaptive bandwidth allocation with SCHED_DEADLINE approach outperformed what was achieved under SCHED_FIFO using `WALT`, with an audio latency ten times smaller with a CPU energy consumption about 6.25% higher in the case of dynamic workload, and outperformed the static workload solution, with a 40% reduction in the CPU energy consumption for the same latency.

5.1.5 Conclusions and Open Challenges

This chapter presents an approach to reduce the audio latency for professional grade multimedia applications. This is achieved by extending the Android API with 1) a mechanism to provide hints on the audio application workload change,

[20]The energy consumption metrics for HiHey boards can be found at the following link: `https://android.googlesource.com/kernel/hikey-linaro/+/android-hikey-linaro-4.9/arch/arm64/boot/dts/hisilicon/hi3660-sched-energy.dtsi`. Please, refer to `CPU_COST_A72` field for the energy consumption of an HiKey 960 active `big` CPU.

	SCHED_FIFO with WALT (dynamic)	SCHED_FIFO with WALT (static)	Adaptive bandwidth	Improvement (%)
Audio Latency	**26.67 *ms***	2.67 *ms*	**2.67 *ms***	**90%**
CPU energy Cons.	96.16	**171.01**	**102.17**	**40%**
Total energy Cons.	206.35 *Ws*	256.93 *Ws*	210.22 *Ws*	14.5%

Table 5.3: Summary of the audio latencies and energy consumptions obtained under the SCHED_FIFO with WALT both with dynamic and static workload, and the adaptive bandwidth allocation approaches.

2) a subsystem that forecasts the application computing requirements through heuristics, and 3) an extension to SCHED_DEADLINE to return execution measurements for the audio task.

In the performed experimentation, the presented solution outperforms the traditional energy efficient approach by reducing the audio latency of ten times, at the cost of an acceptable energy consumption increase of approximatively the 6.25%, and also provides a considerable energy consumption reduction of almost the 40%, achieving the same audio latency, with respect to the traditional low-latency approach.

Since some newly developed platforms have good CPU idle energy consumption performance, letting tasks complete as fast as possible at the maximum frequency is claimed to have competitive energy consumptions. An open challenge can be identified in the comparison between a frequency scaling solution and a race-to-idle scheduling policy, trying to exploit the full potential of both the approaches.

Another critical aspect that will be considered is that, when in Android there are multiple coexisting low-latency audio applications, their samples are sent to the `FastMixer`, which performs the signal mixing and outputs the obtained signal to the audio device. Having multiple interdependent tasks is something that also happens within the same audio application when using a parallel programming paradigm to produce the audio stream. Moreover, either if the platform used for the experiments of this section is a heterogeneous big.LITTLE, all the experiments have been performed forcing SynthMark to run on a single CPU of the big cluster. This solution does not exploit the full potential of the hardware architecture, but is the only viable approach due to the missing scheduler support for evaluating the CPU capacities in the migrations of the tasks, with the risk of migrating a task on a LITTLE CPU unable to manage the requested workload. The CPU capacity awareness enabling the task to migrate among the available CPUs preserving the real-time constraints is an open challenge already under investigation, which leads to the need of evaluating the migration latency impact on the performance of low-latency applications.

These mentioned scenarios require scheduling features that are not supported by SCHED_DEADLINE yet and represent an interesting open challenge.

5.2 Power Consumption and Computing Time Simulator for Heterogeneous Multicore Architectures

Mobile computing is flourishing as an essential tool to support our daily activities, with relatively powerful battery-operated and interconnected devices, capable of hosting an operating system (OS) and a plethora of interactive applications. These devices have evolved in hardware capabilities over the last few years, with a growing number of connectivity options, an unimaginable growth rate for their volatile and persistent storage sizes and increasing computational power, in the form of multicore architectures. In this context, energy management has been receiving a lot of attention from both research [96, 97] and industrial communities, as energy efficiency is now one of the top concerns when designing new devices, functionality, applications and services. These motivations led to the development of new platforms that are focused on heterogeneous processors with a shared ISA, such as the Arm big.LITTLE$^{(RM)}$. These, departing from traditional symmetric multi-processing (SMP) architectures, possess both low-complexity, low-power cores specializing in device bookkeeping activities, and high-power cores for CPU-intensive activities, in addition to classical frequency switching capabilities, where tasks can be migrated among all of the cores as needed.

This is causing the urgent need for engineering new functionality at the OS level, and specifically regarding joint CPU scheduling, task placement and energy management, where proper and novel trade-offs among energy consumption and interactivity of devices and applications with the outside world have to be sought. This is witnessed, for example, by the recent volume of activities around the Energy-Aware Scheduling (EAS) framework in the Linux kernel[21], engineered around the support for Arm-based CPUs in Android.

To this end, it is essential to rely on accurate models of the underlying hardware behavior and the impact of the available energy-management tunables on the application performance. These models can be embedded within proper simulation tools that allow for estimating the expected impact of novel energy management and task scheduling features at the OS level on the final application performance and its capability to respect possible timing constraints.

Problem Presentation and Contributions

Aiming at reproducing the energy behavior of computational activities within embedded heterogeneous architectures, it is necessary to take into account the power consumption of the CPUs and the time necessary to complete the computations, i.e., the execution times. These metrics allow to calculate the energy involved in the computation process and investigate on interesting trade-offs between computation performance and energy efficiency.

This work proposes a support to tackle the problem of power-aware CPU scheduling in real-time systems based on heterogeneous, single-ISA architectures by adopting a *workload-dependent* power consumption model, coupled with a

[21] More information is available at `https://lwn.net/Articles/749738/` and `https://developer.arm.com/open-source/energy-aware-scheduling`.

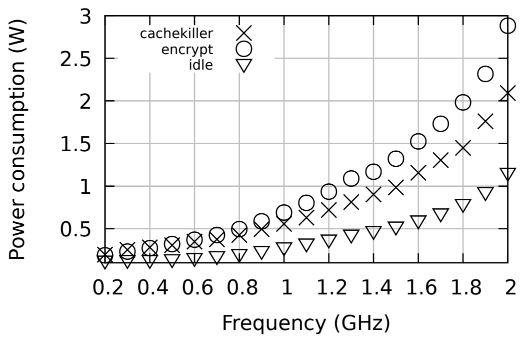


Figure 5.9: Measurements on the power consumption of the big CPU cluster, running different workloads at different frequencies.

workload-dependent execution-time scaling model, at the varying of CPU frequencies. This proposal is based on experimental results highlighting shortcomings of the commonly adopted simple scaling models at varying CPU frequencies, assuming power consumption as a quadratic function of the CPU frequency [203]. Indeed, Figure 5.9 reports the measured power consumption of two real, different workloads on the reference big.LITTLE board (Odroid-XU3), highlighting that the power consumption model also strongly depends on the workload type. The presented measurements show, for example, that at maximum frequency an application performing data encryption (`encrypt`) has a power consumption that is 38% higher than a memory-bound application generating a huge number of cache misses (`cachekiller`), forcing the CPU into continuous stalls waiting for completion of memory accesses. Real workloads pose themselves somewhere in the middle between these two extremes, depending on the instruction mix and data access pattern. Moreover, the same figure shows that the also the power consumption of the CPUs in a clock-gating idle state depends on the frequency.

For non-continuously running activities, the energy consumption depends also on the duration of the computations, which is affected by the frequency of the CPU in a way that is workload-dependent as well. Classical models rely on a simple scaling of the execution time with the operating frequency of the CPU. However, the experimental results show that the workload type affects this relationship in a non-negligible way. These are compared in Figure 5.10, where the measured execution time variation for three distinct workload types is reported, normalized with respect to the execution time at the lowest frequency. As evident, some workload types have a significant deviation from the simple scaling model commonly adopted in literature (continuous line), corresponding to Equation (5.13) in Section 5.2.2. More details will follow in Section 5.2.4.

The contribution of this work is (i) the development of power consumption and execution time models derived starting from established solutions already available in literature, (ii) their implementation through the open-source *RTSIM* real-time systems simulator, and (iii) the validation of the simulation outcome with the measurements performed on a real platform. This tool constitutes then a valuable means for the preliminary evaluation and testing of novel energy-aware task scheduling algorithms.

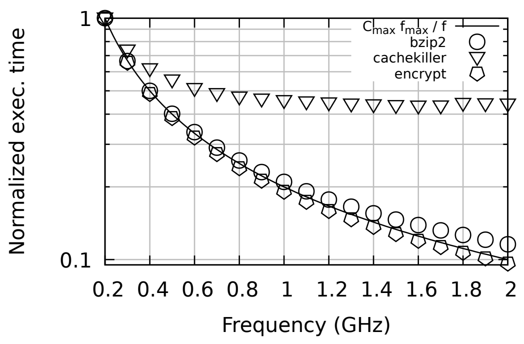


Figure 5.10: Normalized execution times of different workloads running on a big CPU at different frequencies.

5.2.1 Related Work

The research literature related to the present work falls mainly within the classes of power consumption models for voltage-scalable embedded and heterogeneous architectures and simulation of real-time systems including power-aware resource management logic.

Given the growing interest in energy efficient devices, the research communities have already developed several power consumption models for CPUs. The level of details and complexity is variable and depends on the specific application.

As long as the power consumption models are concerned, there are works based on a detailed description of the CPU architecture, such as the one by Möbius et. al. [140], focusing on a CPU model for on-line power estimation using *performance monitoring counter* of the physical CPU. Other detailed models exist, focusing on the electronics behind the CPU operation, like the one developed for the *Wattch* simulator by Brooks et. al. [43]. Indeed, in this case, the goal was the architecture-level power analysis, evaluating the power consumption at the instruction level.

These approaches are very accurate, at the expense of their usability: they require a detailed description of the hardware, and are often characterized by long computation times for the simulation. The here presented approach adopts a higher-level abstraction, which uses real data carried out over a set of micro-benchmarks. This allows for a realistic reproduction of the CPU power consumption pattern throughout an application execution, without the need for considering specific architectural details. This way, it is possible to achieve a trade-off between simulation efficiency and representativeness.

In that direction, high-level descriptions of the CPU power are preferred, like the ones leveraging on more coarse physical models. The interest in these models is justified by the fact that the power consumption of the CPU is modeled using physical quantities like frequency, currents and voltages, which are easier to access. This approach is proposed by several works [23, 46, 104, 177], describing the behavior of real platforms, relying on simplified models with a limited number of parameters and input variables. As demonstrated by Colin et. al. [52], the fitting of a non-linear model with only six parameters over the power behavior of a real board provides sufficient accuracy.

This work uses a more complete power consumption model, obtained taking the cue from the physical behavior of the CPU, close to the one used by Vogeleer et. al. [67]. The parameters of the model have been identified through real experiments, which constitute also a validation process for the final system.

In real-time systems research literature, significant investigations have been carried out in energy-consumption models and energy-aware task scheduling algorithms guaranteeing system schedulability or minimum levels of quality of service (QoS) while at the same time trying to realize energy-efficient policies on hardware supporting dynamic voltage and frequency scaling (DVFS). In this field, a reliable model for the execution time is of paramount importance, both for providing an accurate estimation of the power consumption, and for guaranteeing timing requirements for latency-sensitive applications. Often the computation time models are based on strict theoretical assumption and simplifications, which are far from the operating condition of a real system. Other works, on the contrary, go more into details and model the execution time or delays considering low level hardware information for the given platform. Works like the one by Palacharla et. al. [148] analyze the hardware details of the architecture to quantify the computational speed, but they are too complex and not practical. A more practical approach, which avoids considering hardware details, is shown in the work of Petrucci et. al. [156], where the computation time of a task depends on the accesses to the memory and number of instructions to be executed by the CPU. Despite the realism achieved thanks to the empirical identification of the model parameters, this work only considers CPUs with fixed frequency. Vogeleer et. al. [67] model instead the computation time of the task as a simple function of the frequency, thus allowing considering the effects of CPU frequency scaling.

Compared to the latter approach, ours is still a simple yet representative model, thanks to additional parameters. This achieved acceptable accuracy in a preliminary experimentation.

Other works related to power consumption modeling and scheduling optimization exist in the domain of high-performance computing and many-core systems, but their review is omitted.

Concerning the existence of complete frameworks able to reproduce the overall behavior of a full platform, several tools exist for modeling, simulation and schedulability analysis of embedded, real-time distributed systems, including *RTSIM* [21, 151], *MAST2* [93], *TIMES* [13], *SimTrOS* [39], *YARTISS* [48], *FORTAS* [53], *McSimA+* [9] and others. However, the main purpose of these is to tackle the task-set schedulability problem, not considering the energy impact.

The *SimDVS* [173] by Shin et. al., instead, addresses the problem, evaluating the impact of the scheduling decisions on the energy consumption, but is limited to single processor architectures.

Considering the above, a simulation framework supporting real-time scheduling and a more precise energy model for modern heterogeneous multicore architectures could be considered a useful tool for future research in the area.

5.2.2 Proposed Approach

Model of the Power Consumption

As widely known [46, 104], the power consumption of a CPU is determined by different phenomena, related to its transistor nature: switching activity and leakage effects.

Switching Activity The switching of transistors during the computation requires power to charge the gates capacitors; moreover, it gives birth to some power loss due to brief short-circuit conditions during toggling among logic levels. In the following, the former is called P_{cg}, the latter P_{sc}. As done by Vogeleer et. al. [67], the equations describing these quantities are the following:

$$P_{cg} = \alpha C V^2 f \quad \text{and} \tag{5.6}$$

$$P_{sc} = \eta P_{cg}, \tag{5.7}$$

where C is the capacitance of the transistor gates, f is the CPU switching frequency and V is the CPU voltage. P_{sc} has been considered to be proportional to P_{cg} by a factor η. This choice simplifies the model, without compromising the representativeness. The α factor represents the number of transistors involved in the switching, that is the activity level of the CPU. In the proposed model, α is the quantity that is mostly related to the workload type being run by the CPU, ultimately leading to different possible power consumption levels. Precisely, α is assumed to vary in a range $[\alpha_0, \alpha_{max}]$, where the lower bound α_0 is associated with the idle state of the CPU, whilst the upper bound α_{max} is associated with the CPU under an intensive workload. The overall power consumption due to the switching activity P_{sw} is given by the sum of the two, that is,

$$P_{sw} = P_{cg} + P_{sc}. \tag{5.8}$$

Leakage Effects Despite the technological improvements in the semiconductor device fabrication, there are always some leakage currents flowing between different parts of the transistor. This effect determines a power loss that will be referred to as P_{lk}. The definition of a complete model to represent this phenomena is complex and depends on several variables, including the temperature [116]. To overcome this complexity, some simulators like *Wattch* model this component as a fixed percentage of the dynamic power, others include a simple thermal modeling to improve the accuracy, like TEM^2P^2EST [70]. From the work of Skadron et al. [177], it turns out that there is a relationship between the leakage and the switching powers. More precisely,

$$P_{lk} = \Big(\frac{R_0}{V_0 T_0^2} e^{(\frac{B}{T_0} + \frac{B}{T})} T^2 V\Big) P_{sw}, \tag{5.9}$$

where T_0 is the ambient temperature, B is a constant, V_0 is the nominal voltage and R_0 the ratio between P_{lk} and P_{sw} when $T = T_0$. It is worth highlighting that the equation is temperature dependent. However, in the case of stable temperature, part of the expression can be approximated by a constant factor. This leads to $P_{lk} = \gamma V P_{sw}$, where

$$\gamma = \frac{R_0}{V_0 T_0^2} e^{(\frac{B}{T_0} + \frac{B}{T})} T^2. \tag{5.10}$$

Summarizing, the total power consumption of the CPU, when working at frequency f, voltage V and stable temperature, is:

$$P_{CPU} = P_{sw} + P_{lk} = (1+\eta)(1+\gamma V)\alpha C f V^2. \tag{5.11}$$

In this work, the constants of the model depend on the operating conditions, i.e., during the *idle* state of the CPU, those parameters are expected to be different from the ones of the model representing the CPU performing computational activities. Moreover, each kind of workload has a specific usage of the CPU, inducing a different power demand: these considerations are taken into account when identifying the parameters of the power consumption model.

Power Model Identification As stated above, the model used in this simulation framework is a trade-off between complexity and representativeness, leading to a "gray box" model. Therefore, the model parameters have been identified by fitting the outcome of experiments running a set of micro-benchmarks on a real platform.

The identification procedure has been accomplished using NeuroLab[22], a tool for data fitting based on genetic algorithms. Specifically, the function used for the fitting is

$$P_{CPU} = \delta + (1+\eta)(1+\gamma V) K f V^2, \tag{5.12}$$

where the αC of the Equation 5.11 has been substituted with a single parameter K, which, together with δ, η and γ, constitute the parameters of the fitting model. The additional parameter δ introduces a further degree of freedom in the function fitting. Given the dependence of the power consumption on the workload, the fitting has been accomplished for each workload type in a given set, which will be described in Section 5.2.4. The γ parameter has been considered constant, assuming the temperature as stable during each benchmark. Even though this could be a limitation, a proper modeling would require considering also the thermal model of the CPU, which goes beyond the purpose of this work.

Workload-dependent Execution Time Model

The execution time C of a task running on a CPU with DVFS capabilities is typically assumed as a simple function of the frequency:

$$C(f) = C_{max} \frac{f_{max}}{f}, \tag{5.13}$$

where C_{max} denotes the execution time at the maximum frequency f_{max} on the same CPU. This is based on the assumption that the time required by a task to complete only depends on the frequency. In reality, there are other factors not considered in this model that affect the task duration. For example, the memory access time does not scale with the frequency, thus represents a bottleneck partially mitigated by the use of caches.

An analytical model that has been found out to be able to reproduce the behavior of real workloads at the varying of the CPU frequency with a reasonable

[22]More information at: `https://github.com/balsini/NeuroLab`.

number of free parameters a, b, c, d is the following:

$$C(f) = a + \frac{b}{f} + ce^{-f/d}, \tag{5.14}$$

which includes, in order: (i) a fixed offset that models the presence of bottlenecks for which the speed does not depend on the CPU frequency, (ii) an hyperbolic component that models the ideal execution time scaling with the frequency, and (iii) an adjustment on the function slope.

To achieve the maximum accuracy, the execution time model should also consider, amongst all, the interference introduced by the other tasks running in the system and causing cache and bus contention, and hardware devices accessing the bus with DMA operations. This level of detail is beyond the purpose of this work, for which is provided an execution time model that applies for single tasks running in the system, which still represents a valid approximation for a number of applications.

5.2.3 Implementation Details

The power consumption and execution time models presented above have been implemented within the *RTSIM* simulator. This is a portable, extensible open-source package written in C++ for the simulation of real-time operating systems, supporting many real-time scheduling policies and typical real-time task models. *RTSIM* carries out a high-level simulation focusing on the timing and schedule of tasks in the system, without any functional-level simulation. Its typical use is to simulate worst-case scheduling scenarios for real-time task sets under a given scheduling policy, for the purpose of verifying whether any deadline miss happened or not. *RTSIM* includes a library for discrete event simulation (*METASIM*), and a set of libraries for real-time kernels simulation (*RTLib*). It also provides functionality to trace, store and visualize the events occurred in the simulated environment.

The simulation of a multicore computing system in *RTSIM* involves several modules, among which the most important are:

- `Task`: entity that executes for a given amount of time. A task can also be periodic, as common for real-time environments.
- `Scheduler`: the scheduling policy for the tasks running in the system, for example EDF, global EDF, fixed priority FIFO or Round-Robin, and others.
- `CPU`: modulates the duration of the task with the ideal linear model shown in Equation 5.13, and provides the basic power consumption estimation as $P(f) = V^2 f$.
- `Kernel`: a glue entity that connects the tasks with the scheduler and the CPUs, and manages possible virtual resources shared among tasks, like semaphores.
- `Trace`: module to store events and accumulate statistics.

Once the system is initialized, the simulation runs and evolves with the simulated entities generating, exchanging and executing simulation events.

The presented extensions to *RTSIM*[23] include improvements on the `CPU`, `Task` and `Trace` classes, and the implementation of the presented power consumption and execution time models with the `CPUModel` class. With these modifications, when a Task is put in execution, it can now declare the type of workload the task is going to execute, through an extension to the `fixed()` virtual instruction type of the task. This instruction now also specifies the workload type, in addition to the already available execution time of the computation, calibrated at the maximum frequency supported on the highest performance CPU in the system. *RTSIM* automatically adjusts the computation duration and the consumed power when scheduling that task executing each virtual instruction, according to the provided `CPUModel` parameters supplied at CPU instantiation time, the CPU capabilities, the frequency and the workload type.

The `TracePowerConsumption` class implements a power measurement probe, and it is possible to create one instance for each CPU, tracking the power consumption by querying the power model of the associated CPU.

5.2.4 Experimental Results

The simulator has been tested by comparing its simulated results with the ones obtained by experiments on the real platform.

All the experiments presented in this section refer to an Odroid-XU3 board, which embeds a Samsung Exynos 5422 SoC: an Arm big.LITTLE architecture with four Cortex-A7 and four Cortex-A15 CPUs, running the official Odroid Linux kernel 3.10. Similar results have been achieved with a Linux kernel 4.15. All the power measurements have been performed with the INA231 power meters already available within the board.

Experiments to Gather Model Fitting Data

In order to collect detailed data sufficient to fit the presented workload-dependent power consumption model, a number of different workload types have been experimented with: (i) `idle`: no task is running, so the system switches to the clock-gating idle state [152]. The used board supports two idle states, and the deeper can be accessed only when all the CPUs are idle, thus is never considered in these experiments, (ii) `bzip2`[24]: compression algorithm, with maximum compression level, (iii) `des3 encrypt/decrypt`[25]: Triple DES encryption algorithm, (iv) `sha256sum`[26]: checksum algorithm, and (v) `cachekiller`: application written with the purpose of generating a cache miss at every iteration, by accessing elements within an array bigger than the cache memory size, every access performed with a displacement bigger than the cache line.

All the data read or written by the aforementioned data intensive workloads is randomly generated, and stored in a *ramfs* mounted partition to avoid possible latencies or throughput limitations due disk or SD card devices.

To characterize the system behavior for the two different CPUs, each experiment is run by sequentially pinning the workload task on one of the big cores first, and on one of the LITTLE cores later.

[23]Freely available at: `https://github.com/balsini/rtlib2.0/tree/ewili-2018`.

[24]Bzip2 1.0.6, available at: `http://www.bzip.org`.

[25]OpenSSL 1.0.2g, available at: `https://www.openssl.org`.

[26]GNU coreutils 8.25, available at: `http://www.gnu.org/software/coreutils/sha256sum`.

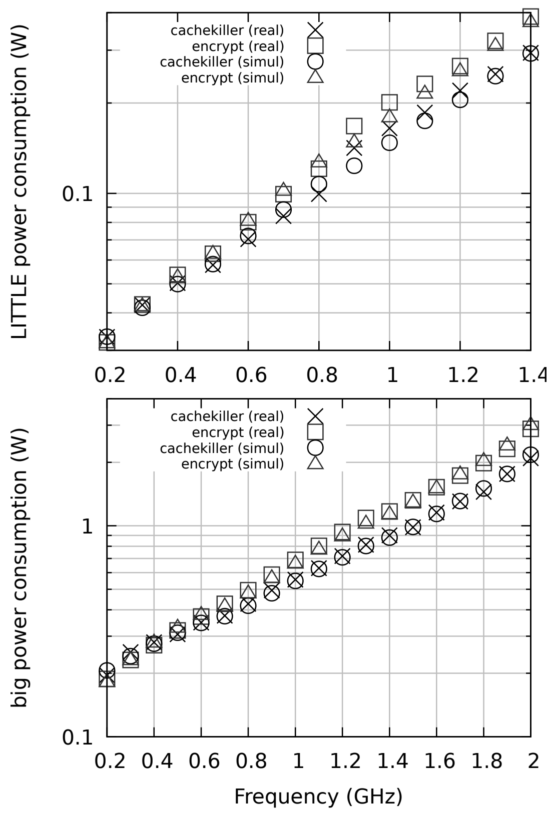


Figure 5.11: Comparison between the power consumption simulated with *RT-SIM* and the respective experimental results.

Each experiment on the real platform is repeated 50 times and each final data point is obtained as an average on the measured values, which have been found to have a small variability from run to run. The observed standard deviation of the measured values for each experiment, normalized with respect to the average among the values, has consistently been in the range between 4% and 12.1%.

Experiments With Modified RTSIM

In a first experiment, the power consumption model implemented in the simulator is evaluated. As shown in Figure 5.11, the simulated behavior successfully maps on the experimental results for the presented workloads. On the other hand, the LITTLE CPU presents a noticeable error in the mid-range frequencies. This effect is likely due to the CPU power supplier, which may be composed of multiple circuits [51] causing a discontinuous behavior.

For simplicity, the plot only shows a subset of the tested workloads. Among all the workloads, the highest relative least square error measured in the comparison between the experimental and simulated results is associated to the `cachekiller` running on a LITTLE CPU, and has a value of 16.1%.

The next experiment evaluates the execution time model for the example workloads running on different CPUs at different frequencies. In this case,

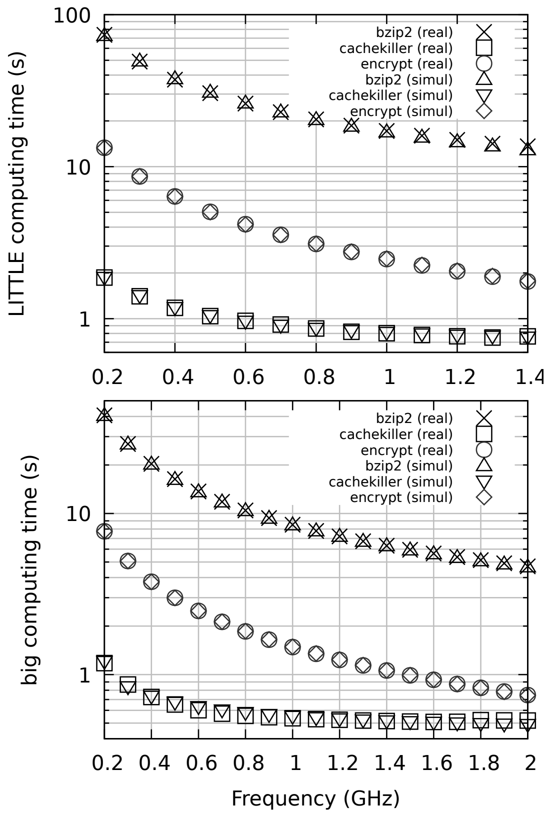


Figure 5.12: Validation of the execution times model of the simulator by comparing the experimental results.

as demonstrated in Figure 5.12, the model behavior is definitely close to the experimental data on the real platform.

As in the previous experiment, the highest relative least square error measured in the comparison between the experimental and simulated results is associated to the `cachekiller` running on a LITTLE CPU, and has a value of 3.57%.

5.2.5 Conclusions and Open Challenges

This chapter presented an effective modeling approach to reproduce the energy and timing behavior of a heterogeneous multicore architecture, running real-time tasks and under different workload conditions. The work dealt also with the implementation of the proposed models on the *RTSIM* simulator, extending its capabilities to obtain a comprehensive suite for simulation of energy-aware strategies.

The extended *RTSIM* real-time scheduling simulator has been tested through experiments, checking the output of the simulated scenarios with respect to the real cases. The accuracy of the results in terms of simulated execution times and energy consumption showed that this work represents a valuable tool for the evaluation and testing of novel energy-aware task scheduling algorithms.

An open challenge can be identified in the extension of the simulation environment to improve the modeling of the power consumption and execution time for complex workload patterns in terms of heterogeneity of the workload and degree of execution parallelism. On the other hand, it would be useful to investigate further modeling approaches for describing generic workloads, i.e., memory bound or CPU bound.

5.3 Real-time Dynamic Reconfiguration of FPGA Accelerators: the FRED Framework

Several embedded computing platforms are evolving towards heterogeneous architectures that integrate multiple processing elements of different nature, as classical central processing units (CPUs), general-purpose computing on graphics processing units (GPGPUs), and field programmable gate arrays (FPGAs). Such platforms allow balancing the flexibility of software systems with the advantages of a highly parallel custom hardware acceleration, thus achieving a consistent speed-up with a contained energy consumption.

FPGAs with DPR capabilities allow the user to reconfigure a portion of the FPGA at runtime, while the rest of the device continues to operate [88]. In particular, the reprogrammable and reconfigurable capabilities of FPGAs, their increasing capacity, and their suitability for signal processing have made them attractive in several application domains, as alternatives to application specific integrated circuits (ASICs) [91]. This is especially valuable in mission-critical systems that cannot be disrupted while some subsystems are being redefined [12].

Such a DPR feature opens a new scheduling dimension for systems running on such heterogeneous platforms, giving the possibility of virtualizing the FPGA, using timesharing techniques, so that it can be used to accelerate a number of hardware functions that is higher than that allowed by static partitioning, thus further improving the application performance.

Today, however, reconfiguration times are about three orders of magnitude higher than context switch times in multitasking, therefore FPGA virtualization can only be used for a limited set of applications. As shown in the next section, reconfiguration times significantly reduced in the recent years and are expected to further decrease in the near future. This enables the development of a new generation of operating systems that can manage the FPGA module, handling both software tasks (SW-tasks) and hardware tasks (HW-tasks) in a uniform fashion.

Trend of Partial Reconfiguration Performance

During a partial reconfiguration process, different hardware modules are involved, such as the memory, the bus, and the FPGA reconfiguration port. As a reconfiguration bitstream traverses such series of modules, the performance of the reconfiguration processes is limited by the slowest element, which represents the DPR bottleneck.

Since the DPR feature was introduced in FPGAs, all such elements were improved during the years. Liu et al. [124] designed a smart reconfiguration peripheral interface, based on the Xilinx ICAP port [154], that is able to approach

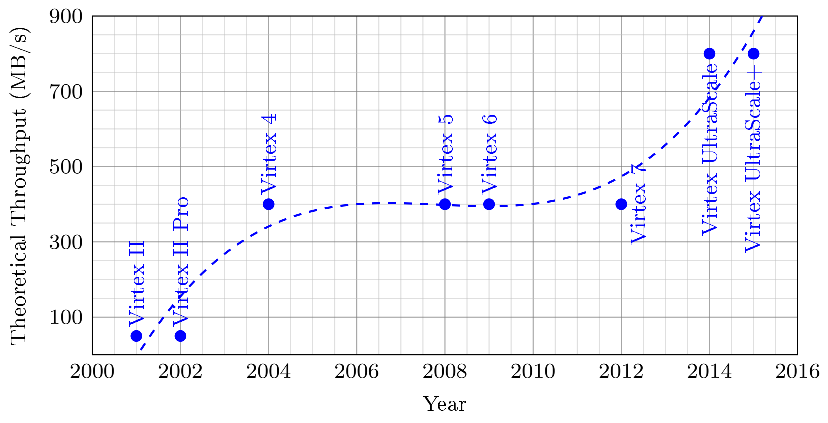


Figure 5.13: Trend of reconfiguration throughput.

a throughput of 400 MB/s. Also, Duhem et al. [77] designed a fast reconfiguration interface by overclocking the ICAP port up to 200 MHz, corresponding to a throughput of 800 MB/s. An overview of the trend of reconfiguration times (obtained by comparing the theoretical maximum throughput calculated from platforms' datasheets [147]) is shown in Figure 5.13. For this reason, it is plausible to expect that such a trend will continue in the upcoming years, thus making DPR a relevant direction to be explored.

Although reconfiguration times are not negligible, FPGAs allow hardware acceleration of a wide class of algorithms with a significant speedup factor [50, 172] over the corresponding sequential software implementation. For instance, in the case study analyzed in this chapter, a speedup factor up to 15x has been measured for an image processing filter implemented on the Zynq-7010 platform, which can reach a throughput of 145 MB/s for the DPR, allowing to reconfigure an FPGA area containing about 25% of the total resources in less than 3 milliseconds.

Nowadays, one of the top gamma products is represented by the Xilinx Zynq Ultrascale+, compatible with DDR4 memory and able to reach a maximum transfer rate of 2400 Mbps. It is connected with the ARM AMBA AXI4 and its logic elements are configured by an evolution of the SelectMAP reconfiguration port, called ICAP, running at a maximum frequency of 200 MHz with a data size of 32 bits.

In addition to the improvements achieved on the memory and the communication bus, a performance boost from the memory storage side has also been obtained through a bitstreams compression [181], moving the actual bottleneck to the reconfiguration interface.

Estimating the throughput of the reconfiguration process is not trivial, as it requires a precise ad-hoc orchestration of each hardware module involved in the process, and also requires the availability of all the hardware devices that are intended to be compared. Figure 5.13 shows the evolution of the FPGA reconfiguration performance during the last years, obtained by comparing the theoretical maximum throughput estimations calculated from the datasheets of the devices.

Since a higher throughput corresponds to smaller reconfiguration times (for a given bitstream size), the positive trend shown in Figure 5.13 enables a more dynamic management of the FPGA, allowing the implementation of virtualization mechanisms that can provide great advantages to real-time applications, with respect to fully static approaches.

Contributions

To investigate this issue, this chapter presents the prototype implementation for a timesharing mechanism that can be used to dynamically reconfigure predefined FPGA areas for accelerating different functions associated with real-time periodic tasks. The results achieved on such a prototype are encouraging and clearly show that, in spite of the relatively high reconfiguration times, a timesharing mechanism on the FPGA can significantly improve the performance of real-time applications with respect to a fully static approach.

When exploiting FPGAs with DPR in real-time embedded systems, a crucial issue is to provide worst-case response time bounds of computations consisting of software tasks and hardware accelerated functions. Although several works have been done to analyze the timing behavior of real-time applications using FPGAs, most of them did not consider DPR capabilities at a job level. To overcome this lack, this chapter also presents a new computing framework for enabling a timing analysis of real-time activities that make use of hardware accelerators developed through programmable FPGAs with DPR capabilities:

1. It presents FRED, a framework for supporting real-time applications on FPGAs with DPR feature. It relies on a *static* off-line partitioning of the FPGA fabric to limit worst-case scenarios arising when using DPR. Design issues related to scheduling and inter-task communication are also discussed for bounding worst-case delays.

2. It proposes a new task model to abstract a set of real-time activities running on the considered architecture.

3. It derives a response-time analysis to verify the schedulability of a set of real-time tasks consisting of both software parts and hardware accelerated functions.

The proposed framework has been conceived by considering several real-world constraints that are present on today's platforms. In fact, FRED has been also practically validated with a proof-of-concept implementation on the Zynq platform [73]. Such a practical validation highlighted that the proposed approach can be actually supported by state-of-the-art technologies with a limited run-time overhead. Moreover, to explore the worst-case performance of FRED, an empirical study[27] (based on synthetic workload) has been conducted to evaluate the proposed response-time analysis under different operating scenarios.

This chapter finally proposes the implementation of the FRED framework on the Linux operating system addressing several challenges, such as the architectural support for the accelerators, the reconfiguration and communication mechanisms, the implementation of the FRED scheduler, and the synchronization mechanisms between software and hardware tasks. In particular, it presents

[27]Artifact Evaluation (AE) instructions are available at: `http://retis.sssup.it/~a.biondi/ae/FRED/`. The artifact has been accepted by the RTSS AE committee.

a software architecture for Linux composed of (i) a kernel module for implementing shared-memory communication with hardware accelerators, (ii) a driver to handle the FPGA reconfiguration, and (iii) a user-space server process to schedule the requests for hardware acceleration.

5.3.1 Related Work

The reduction of reconfiguration times resulting from the FPGA technology evolution allowed exploiting the advantages of DPR for handling applications with a dynamic behavior. For example, a HW-task that could only be statically allocated in the earlier platforms, can now be reconfigured at runtime to implement mode changes in the application. More recently, some authors proposed methods for supporting a reconfiguration that can be periodically requested by SW-task at every job execution. This approach is referred to as *job-level reconfiguration*.

The solutions proposed in the literature to exploit FPGA acceleration are quite heterogeneous due to the evolution of such platforms and the wide range of applications that can take advantage of this technology. The intrinsic parallelism, the reduced interference among the running activities, and the reduced variability in the execution made such a technology appealing for real-time applications, ranging from network management [135] to scheduling of hard [71] and soft [89] tasks. However, these solutions are limited to static or slowly evolving scenarios. Before analyzing the related work on DPR for real-time task scheduling, a taxonomy is first introduced to classify the existing solutions and precisely position the proposed approach with respect to the literature.

Taxonomy The features considered to organize the taxonomy concern the reconfiguration approach, the allocation methods, the model of the *FPGA reconfiguration interface* (FRI), and the types of managed tasks.

Reconfiguration Approaches They can be distinguished between *static* and *dynamic*. In a static approach, the allocation of hardware tasks (HW-tasks) is performed at the initialization phase, while in a dynamic approach HW-tasks can be allocated at runtime upon specific events. Dynamic approaches can be used to support *mode-changes* in the application (allowing tasks to be added and removed from the task set) or trigger a reconfiguration every time a new job is scheduled (*job-level* reconfiguration). A static approach has no runtime reconfiguration overhead, but the maximum number of HW-tasks is limited by the physical size of the FPGA. Dynamic approaches trade extra reconfiguration overhead to increase the total number of HW-tasks that can be managed.

Allocation Methods They can be distinguished between *slotted* and *slotless*. In a slotted approach, the FPGA area is partitioned into slots of given size connected via buses provided on the static part of the FPGA. A HW-task can occupy one or more slots. In a slotless solution, HW-tasks can arbitrarily be positioned on the FPGA area and data are transferred through the reconfiguration interface inside the FPGA. Slotted approaches have the advantage of having the communication channels already in place, but the FPGA area may be partially wasted due to slot granularity. On the other hand, slotless solutions increase the utilization efficiency of the FPGA area, but are penalized by higher reconfiguration times due to the instantiation of communication channels and the increased traffic on the FRI due to the additional data transfer.

FRI Model The FRI plays a central role in FPGAs with DPR, thus, building a proper model of the FRI is crucial for estimating worst-case delays and enabling a real-time analysis. The easiest approach is to reduce complexity by considering reconfiguration delays negligible. This is a strong unrealistic assumption, considering that, in current FPGAs, reconfiguration delays can have the same order of magnitude of task execution times. A simple approximation can be obtained using a constant reconfiguration time. However, since the reconfiguration time is proportional to the number of elements to be reconfigured, and the FRI is a shared resource, providing a safe bound would introduce a huge pessimism in the analysis. Less pessimistic values can be obtained considering the reconfiguration time composed of two elements: one proportional to the number of elements to be reconfigured and one due to the time spent in waiting for the FRI. Most of the works focused on kernel mechanisms considered an FRI model tailored to real solutions, as the Xilinx ICAP port [154].

Task Model Modern heterogeneous platforms include FPGAs modules together with processors on the same chip [204]. On such platforms it is thus possible to execute both HW-tasks, running on the FGPA, and software tasks (SW-tasks), running on the processors.

Related Work Analysis The works considered in this section are related to the proposed approach in that they provide a timing analysis under reconfigurable FPGA architectures or propose a software support for HW-task management.

Di Natale and Bini [71] proposed an optimization method to partition the reconfigurable area of a homogeneous FPGA platform into slots to be allocated to HW-tasks and softcores running the remaining tasks. Given the high computational complexity of the method, this approach can only be used off-line to obtain a static task allocation, hence it does not exploit the advantages of the dynamic reconfiguration. Pellizzoni and Caccamo [155] addressed a similar problem in a more dynamic scenario, proposing an allocation scheme and an admission test to provide real-time guarantees of applications supporting mode changes, where tasks can either be executed in software on a CPU or in hardware on the FPGA.

Danne and Platzner [65] presented two algorithms (one EDF-based and one server-based) to schedule only preemptive HW-tasks, but the model adopted for the FPGA platform is quite simple and does not consider any reconfiguration time and allocation constraints. Saha et. al. [169] presented a new scheduling algorithm for preemptable HW-tasks, exploiting the higher speed and the improved capabilities of modern reconfiguration interfaces to dynamically change the allocation every time a task terminates. However, this approach assumes a homogeneous partition and a fixed reconfiguration time, which can lead to a huge waste of the area and a high pessimism in the analysis. In summary, in all the works cited above, the models used for the FPGA and the reconfiguration interface are too simple to describe the limitations of the available platforms, and

Paper	Reconfig.	Alloc.	FRI model	Tasks	RTA
Lübbers, 09	Static	Slotted	ICAP	HW/SW	No
Lübbers, 10	Job-level NP	Slotted	ICAP	HW/SW	No
Happe, 15	Job-level P	Slotted	ICAP	HW/SW	No
Iturbe, 15	Job-level NP	Slotless	ICAP	HW/SW	No
Di Natale, 07	Static	Slotless	*Not required*	HW/SW	Yes
Pellizzoni, 07	Mode-ch NP	Slotted	*Not addressed*	HW/SW	Yes
Danne, 05	Job-level P	Slotless	*Zero overhead*	HW	Yes
Saha, 15	Job-level P	Slotless	*Fixed overhead*	HW	Yes
Dittmann, 07	Job-level NP	Slotted	General (NP)	HW	Yes
This work	Job-level NP	Slotted	General (P/NP)	HW/SW	Yes

Table 5.4: Classification of the related work.

the corresponding approaches do not fully exploit reconfiguration capabilities under real-time constraints.

Dittmann and Frank [74] addressed the analysis of reconfiguration requests as a single core scheduling problem. The paper assumes a single set of homogeneous slots managed by a non-preemptable FRI and considers only HW-tasks (SW-tasks are not taken into account). Unfortunately, due to missing proofs, it is not clear how response-time bounds follow. In addition, the authors did not investigate sustainability issues and their analysis may be affected by later-discovered misconceptions concerning non-preemptive fixed-priority scheduling [66].

Classification Table 5.4 classifies the presented papers according to the proposed taxonomy, also highlighting the availability of a real-time analysis (RTA) to better emphasize the differences with respect to the proposed approach. Summarizing, different approaches have been proposed to exploit the advantages of DPR-enabled FPGAs, but none of them provided worst-case bounds for enabling a worst-case timing analysis of real-time sets of mixed HW-tasks and SW-tasks. In addition, most of the previous work did not consider heterogeneous FPGA slots. To overcome these limitations, the work proposed in this chapter presents a heterogeneous slotted-based framework designed to make reconfiguration times more predictable and derive a schedulability analysis for real-time applications exploiting DPR capabilities. FPGA reconfiguration is managed at the job level and the schedulability analysis takes into account the delays and the constraints coming from the FRI. Both preemptive and non-preemptive reconfiguration are analyzed.

Operating System Support A few approaches have been proposed to provide an operating system support for DPR in platforms including an FPGA. The common adopted solution for exchanging data between SW-task and HW-tasks is through proper software stubs interacting with the kernel scheduler and handling the HW-tasks using a dedicated library.

For instance, Lübbers and Platzner [128] proposed the ReconOS operating system, which extends the classic multi-threading programming model to hardware activities executed on an FPGA. HW-tasks interact with SW-tasks threads trough a custom developed POSIX-style API, using the same operating system mechanisms, like semaphores, condition variables, and message queues. Originally designed for fully-reconfigurable FPGAs, this solution has then been extended by the same authors to support partial reconfiguration [127], with a cooperative multitasking approach dealing with the contentions on a set of predefined reconfiguration slots. More recently, Happe et. al. [92] extended the ReconOS execution environment to provide HW-tasks preemptability. However, the focus of this work is on hardware enabling technologies, rather than kernel support mechanisms.

Iturbe et al. [98] presented the R3TOS operating system to support dynamic task allocation on an FPGA without relying on predefined slot partitioning and static communication channels. In their solution, scheduling and allocation of HW-tasks are performed by a module, called HWuK, which is also in charge of controlling the programming interface in an exclusive manner. The authors proposed a HW-task model, as well as algorithms for scheduling and allocation. However, a worst-case analysis is not provided and nothing is said on the schedulability of SW-tasks. Such a dynamic slot partitioning increases flexibility in the FPGA allocation at the cost of a higher complexity of the reconfiguration algorithms, reflecting in higher worst-case reconfiguration times.

The major problem in such kernel extensions is that they have been designed to improve the *average* system performance, without providing tight worst-case response times bounds. As a consequence, a model of the FPGA runtime behavior based on these methods leads to huge pessimism if used for a real-time scheduling analysis.

Also the Linux community has shown interest in the exploitation of the FPGAs features. However, the current mainline kernel only provides a simple support for the reconfiguration interface. So and Brodersen proposed BORPH [178], which extends the Linux kernel to allow co-scheduling of SW-tasks and HW-tasks. However, the project is discontinued and does not consider modern platforms.

In summary, none of the presented works addressed the problem of modeling the timing behavior of the reconfiguration interface and the interaction between SW-tasks and HW-tasks in such a way that they can be used for a tight real-time analysis. To address this issue, a prototype implementation of a job-level FPGA management has been developed to: (i) profile the timing behavior of the reconfiguration port with the purpose of deriving such a model, (ii) investigate the practical feasibility of the job-level approach for real-time applications, and (iii) identify possible bottlenecks. The final section of this chapter reports the results of some experimental studies conducted on such a prototype implementation.

The presented chapter shows how to overcome some limitations of the current state of the art providing: (i) the implementation of a framework designed to increase predictability of application exploiting FPGA acceleration, (ii) an implementation that does not limit HW-tasks to specific paradigms (e.g., stream processing, data flow), (iii) an efficient use of HW resources (i.e., improved reconfiguration interface driver, zero-copy data transfer mechanisms), (iv) a scalable implementation with respect to the number of HW-tasks in the system, and (v) a seamless integration in the Linux kernel.

5.3.2 Reconfiguration Times and Speedup Evaluation

This chapter presents a preliminary evaluation of the dynamic partial reconfiguration features of a real FPGA, both in terms of the speedups achieved with a hardware implementation of software functions, and reconfiguration speed.

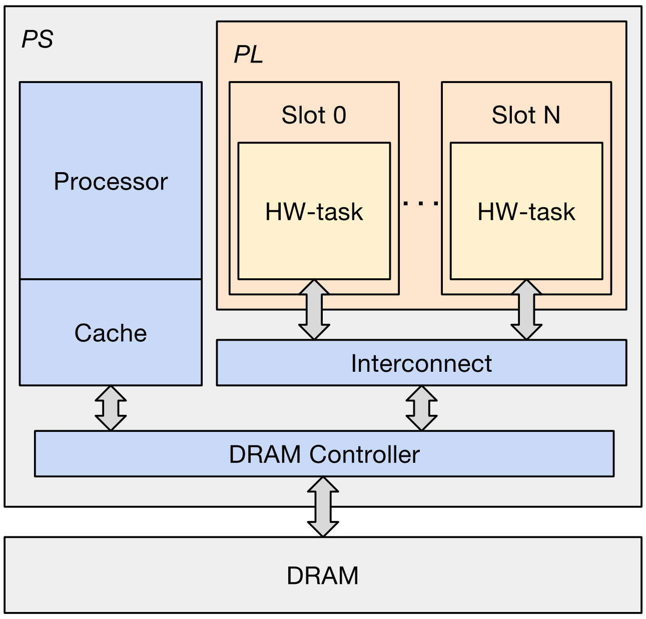


Figure 5.14: Block diagram of the considered system.

This preliminary evaluation on a real platform is fundamental for estimating if the proposed idea is feasible and provides computational advantages with respect to the classical CPU implementation, or the static FPGA accelerators configuration.

System Description

This work considers a heterogeneous computing system consisting of *one* processor and a DPR-enabled FPGA fabric, both sharing a common DRAM memory. A representative block diagram of the considered system is illustrated in Figure 5.14.

Possible representative platforms compatible with the considered system include the Zynq-7000 family by Xilinx, which provides ARM Cortex A9 processors and a FPGA fabric ranging from 28K up to 444K logic cells. Two types of computational activities can run on such a system:

- *software tasks* (SW-tasks): they are computational activities running on the processor; and
- *hardware tasks* (HW-tasks): they are functions implemented in programmable logic and executed on the FPGA fabric.

SW-tasks can speedup parts of their computation by requesting the execution of HW-tasks, which can be considered as *hardware accelerated functions*.

The area of the FPGA fabric is divided into a reconfigurable region and a static region. The reconfigurable region hosts the HW-tasks while the static region includes support modules for the HW-tasks, such as communication devices. The reconfigurable region is partitioned into *slots*, each including the same number of logic blocks. A HW-task can execute only if it has been programmed into a slot. Each slot can be reconfigured at run-time by means of a *FPGA reconfiguration interface* (FRI) and can accommodate at most one HW-task.

As typical for most real-world platforms (e.g., [105, 190]), the FRI

1. can reconfigure a slot without affecting the execution of the HW-tasks currently programmed in other slots;
2. is a peripheral device external to the processor (e.g., like a DMA [204]) and hence does not consume processor cycles to reconfigure slots; and
3. can program at most one slot at a time.

To program a given HW-task into a slot, the FRI has to program all the logic blocks of the slot. This is because unused logic blocks have to be disabled to "clean" possible previous configurations. The FRI is characterized by a *throughput* ρ, meaning that a time $r = b^S/\rho$ is needed to reconfigure a slot, where b^S is the number of logic blocks in each slot.

Each SW-task uses a set of HW-tasks by alternating execution phases with *suspension* phases where the SW-task is descheduled to wait for the completion of the requested HW-task. The same HW-task cannot be used by more than one SW-task. Each SW-task is periodically (or sporadically) released, thus generating an infinite sequence of execution instances (denoted as jobs). SW-tasks are also subject to timing constraints, meaning that each of its jobs must complete its execution within a *deadline* relative to its activation. Listing 5.1 reports the pseudo-code defining the implementation skeleton of a SW-task that calls a single HW-task.

```
void sample_software_task()
{
  // Task initialization (executed only once)
  << Initialization part >>

  // Define an instance of an HW-task
  Hw_Task hw_task = hw_task_init(sample_hw_task);

  // Task body
  while (true) {
    << Software elaborations chunk >>

    // Configure input and output data for the HW-task
    hw_task_set_args(hw_task, input_ptr, output_ptr);

    // Reconfigure and execute the HW-task
    rcfg_manager_execute_hw_task(hw_task);

    << Software elaborations chunk >>

    // Wait for the next job
    suspend_until(period);
  }
}
```

Listing 5.1: Pseudocode of a SW-task calling a HW-task.

The HW-task is initialized at line 7, where the label `sample_hw_task` is used to refer its implementation stored in memory. At line 14, the SW-task configures the HW-task by specifying two memory locations: (i) `input_ptr`, that contains the *input data* for the HW-task and (ii) `output_ptr`, prepared to contain the *output data* produced by the HW-task. Finally, at line 17, the SW-task executes a *blocking* call that triggers the reconfiguration and executes the HW-task. The SW-task correspondingly suspends its execution until the completion of the HW-task. The inter-task communication mechanism is discussed in the following section.

System Prototype

This section presents the implementation of a system prototype to handle HW-tasks under DPR on a real platform. The prototype has been used to conduct some preliminary experiments to evaluate the feasibility and the performance of the proposed approach.

Reference Platform The Zynq-7000 SoC family has been chosen as a reference platform for developing a working prototype of the system. It includes a dual-core ARM Cortex-A9 processor and a DPR-enabled FPGA fabric integrated on the same die.

The internal structure of a Zynq SoC comprises two main functional blocks referred to as processing system (PS) and programmable logic (PL) [204]. The PS block includes the ARM Cortex-A9 MPCore, the memory interfaces and the I/O peripherals, while the PL block includes the FPGA fabric. The subsystems in the PS are interconnected among themselves, and to the PL side, through an *ARM AMBA AXI Interconnect.*

The Interconnect can be accessed by custom logic modules (configured on the PL side) through a set of *master* and *slave* AXI interfaces exported by the PS to the PL side. In particular, the slave interfaces allow hardware modules hosted on the PL to access the global memory space where the physical RAM memory is mapped. This is achieved by implementing an AXI master interface inside the module logic. Such a master interface can be connected to the corresponding slave interfaces offered by the PS. In this way it is possible to implement a *shared-memory* infrastructure between the processor and the custom modules deployed on the PL.

The SoCs of the Zynq family supports dynamic partial reconfiguration under the control of the software running on the PS. The FPGA fabric included in the PL can be fully or partially reconfigured via the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams (i.e., images of custom modules to be configured onto the FPGA) from the main memory to the PL. This is achieved by means of the the processor configuration access port (PCAP).

Prototype Architecture In the system prototype, the area of the FPGA fabric included in the PL is divided into a static region and a reconfigurable region. The static region contains the static portion of the communication infrastructure (consisting in interconnection blocks similar to switches) and other support modules, while the reconfigurable region hosts the hardware modules that implement the HW-tasks and a common communication interface.

Such a common interface is similar to the one adopted by Sadri et al. [168] and includes (i) an *AXI master interface* for accessing the system memory, (ii) an *AXI slave interface* through which the HW-task can be controlled by the PS, and (iii) an *interrupt signal* to notify the PS when the computation has been completed. In the current setup, the AXI master interfaces included in the HW-tasks are attached to high-performance (HP) ports exported by the PS, while the AXI slave control interfaces are attached to the PS AXI master general purpose ports.

The reconfigurable region is partitioned into a fixed number of slots, each containing an equal number of logic resources. Each slot can accommodate a single HW-task. Since bitstreams relocation is not supported by the Xilinx's standard tools [105, 190] (i.e., the same bitstream cannot be used for multiple slots), each HW-task is synthesized as a *set of bitstreams*, one for each slot defined in the PL.

Software Support The software part of the system prototype has been developed as a user-level library for the FreeRTOS [125] operating system. The library facilitates the reconfiguration and the execution of HW-tasks by providing a simple API that enables the client programmer to exploit hardware acceleration.

From the client programmer perspective, the library models the concept of hardware acceleration with a set of HW-task objects and a software module named reconfiguration service. The interface of the reconfiguration service offers a single function to request the execution of a HW-task (as shown in Listing 5.1, line 17). Each HW-task object includes the following information: (i) a set of bistreams, one for each slot; (ii) the input parameters (memory pointers or data); (iii) two optional callbacks (linked to the start and the completion of the HW-task) that can be used to ensure memory coherence. The library has been build on top of the Xilinx software support library [144].

Before executing a HW-task, the presented implementation flushes the portion of cache containing the input data prepared by the SW-task, thus ensuring that the HW-task can access coherent data from the RAM memory.

Once the input data have been prepared, the SW-task checks for a vacant slot performing a *wait* operation on a FreeRTOS counting semaphore (initialized with the number of available slots). If all the slots are busy, the calling task is suspended until one of the slots will be released. When at least one slot is available, the function searches if any of the vacant slots already contains the requested HW-task. If none of the vacant slots contains the required HW-task, one of the vacant slots is reconfigured with the corresponding bitstream. The calling task is suspended until the reconfiguration has been completed.

As soon as the requested HW-task is configured, it starts executing. The calling SW-task suspends its execution until the completion of the HW-task. When the HW-task completes, the calling SW-task is resumed and performs a *signal* operation on the slots counting semaphore. The completion is notified to the PS with the interrupt signal predisposed in the common interface described in Section 5.3.2. Once the SW-task is resumed, the presented implementation invalidates the cache portion corresponding to the output data produced by the HW-task, thus ensuring that the processor can access coherent data.

Experimental Results

This section presents a preliminary case study implemented on the Zynq-7000 platform to evaluate the feasibility of the proposed approach, profile hardware acceleration speedup factors, and measure reconfiguration overheads. The considered platform includes a dual-core ARM Cortex-A9 processor and a 7-series FPGA integrated on the same chip. The internal structure of a Zynq SoC can be divided in two main functional blocks referred to as processing system (PS) and programmable logic (PL) [204]. The PS block comprises the ARM Cortex-A9 MPCore, the memory interfaces and the I/O peripherals, while the PL block includes the FPGA programmable fabric. The subsystems included in the PS are interconnected among themselves and to the PL through an ARM AMBA AXI (Advanced eXtensible Interface) interconnect.

The hardware modules configured on the PL can access the interconnect through a set of master and slave AXI interfaces exported by the PS side to the PL side. Slave interfaces allow modules to access the global memory space and share the DRAM memory with the processors. Dynamic partial reconfiguration is supported under the PS control. PL fabric can be fully or partially (re)configured by the PS through the device configuration interface (DevC) subsystem. The DevC includes a DMA engine that can be programmed to transfer bitstreams from the main memory to the PL configuration memory through the processor configuration access port (PCAP).

To test the system, four standard algorithms have been implemented as both HW-tasks and equivalent software procedures. The test set includes tree simple implementations of image convolution filters (*Sobel*, *Sharp* and *Blur*) and an integer matrix multiplier (referred to as *Mult*). The HW-tasks have been designed with the Vivado high-level synthesis tool, while the software versions have been implemented in the C language.

The Blur and the Sharp filters have been configured to process images of size 800×600 pixels, while the Sobel filter has been configured to process images of size 640×480 pixels. All the three filters process images with 24-bit color depth. The matrix multiplier processes matrices of size 64×64 elements.

System Architecture In the prototype developed for the case study, the PL area is divided in two main regions: a static region and a reconfigurable region. The static region contains the communication infrastructure and other support modules, while the reconfigurable region is organized as a single partition divided into S slots, each hosting a HW-task.

In general, since bitstream relocation is not supported by the Xilinx standard tools [105, 190], each HW-task τ_i^H is implemented as a set of n_k^S bitstreams, one for each slot S_j of its associated partition $P(\tau_i^H)$. Each slot S_j can accommodate all the specific implementations of each HW-task τ_i^H that belongs to partition $P(\tau_i^H)$.

Since the slot interface should match the one of the HW-tasks [190], a common interface that all HW-tasks are required to implement is then defined. Such a common interface is similar to the one adopted by Sadri et al. [168]. The interface includes an AXI master interface for accessing the system memory, an AXI slave interface through which the HW-task can be controlled by the PS, and an interrupt signal to notify the PS. The AXI master interface logic allows HW-tasks to retrieve data autonomously from the memory space.

Algorithm		Mult	Sobel	Sharp	Blur
Observed HW execution times	Average (ms)	0.785	12.710	24.631	24.628
	Longest (ms)	0.785	12.712	24.633	24.629
Observed SW execution times	Average (ms)	1.980	115.518	304.975	374.785
	Longest (ms)	2.017	115.521	304.994	374.811
Speedup	Average	2.523	9.089	12.381	15.217
	Minimum	2.515	9.087	12.380	15.216

Table 5.5: Speedup evaluation.

In the current experimental setup, the AXI master interfaces exported by the HW-tasks are attached to high-performance slave ports exported by the PS, while the AXI slave control interfaces are attached to the general purpose master ports. The software part consists of a user-level library for the FreeRTOS operating system. The library abstracts the reconfiguration mechanism and provides a simple API that enables SW-tasks to request the execution of HW-tasks on the PL through the `rcfg_manager_execute_hw_task()` function, as described in Listing 5.1.

Experimental Setup The system prototype has been deployed on a ZYBO board that includes the Z-7010 Zynq SoC and 512 MB of DDR3 memory. The ARM core included in the PS of the Z-7010 runs at 650 MHz, while the clock frequency for the PL is set to 100 MHz.

In the experimental setup, 50% of the logic resources of the PL are allocated to the reconfigurable partition, while the remaining 50% are allocated to the static part. The reconfigurable partition is divided into two slots of equal size. Each slot contains half of the resources available in the reconfigurable partition. Since both slots contain an equal number of resource, the corresponding bit-streams (resulting from the logic synthesis of HW-task in each slot) have the same size, equal to 338 KB. Considering the size of the RAM memory available on the platform (512 MB), a large number of partial bitstreams can be stored without any relevant impact on the available memory.

Speedup Evaluation A first experiment has been carried out to measure the speedup factors achievable by the HW-task implementation of the four algorithms used in the case study. For each of such algorithms, the execution time of the corresponding HW-task has been compared with the equivalent full software implementation for 1000 runs. The results of this test are reported in Table 5.5. The minimum speedup has been computed as the ratio between the minimum observed execution time of the software implementation and the maximum observed execution time for the HW-task.

As can be seen from the table, even though the FPGA is running at a lower clock frequency ($100MHz$) compared to the processor ($650MHz$), HW-tasks provide a consistent speedup ranging from 2.5 to 15.2. The small differences between average and worst-case execution times can be explained by the fact that the functions are essentially stream processing operations with no branches depending on the input data.

Response-time Evaluation A second experiment has been performed to evaluate the system behavior in a scenario where the number of HW-tasks to be executed exceeds the number of slots available on the FPGA fabric. Please note that such a scenario *is only possible by exploiting DPR*. The task set used for this experiment consists of four periodic SW-tasks with implicit deadline (i.e., deadlines equal to task periods). Each SW-task requests the execution of the HW-task corresponding to the algorithm of the case study (Section 5.3.2). SW-tasks priorities are assigned according to the Rate-Monotonic algorithm. As mentioned in Section 5.3.2, each SW-task executes a flush operation (denoted as *cache flush*) before calling the HW-task and invalidates the cache when the HW-task completes (*cache invalidate* operation).

Table 5.6 reports the periods of the SW-tasks, the execution times of the cache flush and cache invalidate operations, and the response-times of the SW-tasks observed in 8 hours of execution.

Based on the collected data, it is worth observing that the considered application *cannot be scheduled without DPR* for the following reasons:

- due to the large execution times (see Table 5.5), the application cannot be scheduled with a full software implementation;
- since the FPGA fabric has only two slots, it is not possible to statically configure all the four HW-tasks of the application;
- if the algorithms that cannot be allocated on the FPGA as HW-tasks are executed on the processor as pure software implementation, *any* possible combination of HW-tasks and software implementations leads to a non schedulable system.

This example shows that virtualizing the FPGA by the proposed timesharing mechanism can effectively improve the schedulability of applications on current heterogenous platforms.

The longest observed response time for the *Mult* SW-task shows that, even if this task has the highest priority in the system, it may experience high delays due to slot contention with other HW-tasks issued by lower-priority SW-tasks.

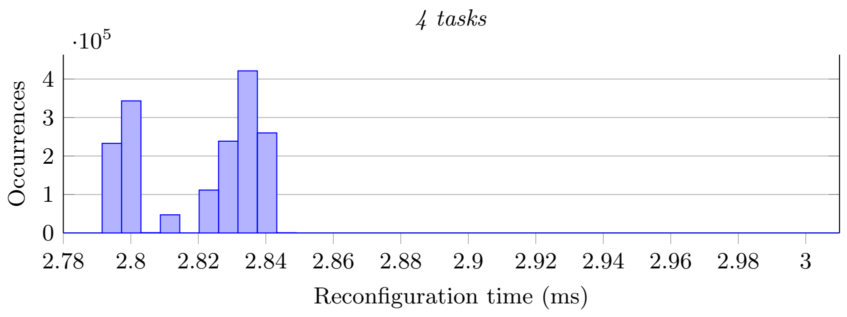
This happens because of the FIFO ordering of the semaphores used in the implementation. The execution of HW-tasks can hence be delayed by the reconfiguration and the execution of all the HW-tasks requested by other SW-tasks (independently of their priority). The analysis of such a delay is beyond the scope of this thesis.

For some applications, the response-times can be improved by adopting different scheduling policies (i.e., different from FIFO) to manage HW-tasks. However, since HW-tasks execute in a non-preemptive manner, the largest execution time of the HW-tasks will always impose a lower-bound for the slot contention delay.

Reconfiguration Times Profiling Finally, a third experiment has been conducted to profile reconfiguration times. The reconfiguration of the FPGA fabric is performed by the DevC subsystem described in Section 5.3.2. Such a module transfers bitstreams from the main memory to the PL configuration memory trough the PCAP port, which exploits the DevC DMA engine. The DMA accesses the system memory (where bistreams are stored) through an AXI master

SW-task		Mult	Sobel	Sharp	Blur
Period (ms)		30	50	80	100
Cache flush (ms)		0.030	1.123	1.754	1.754
Cache invalidate (ms)		0.017	1.240	1.939	1.939
Observed	Average (ms)	3.829	17.603	31.416	35.624
Response time	Longest (ms)	24.017	20.418	33.086	43.160

Table 5.6: Hardware accelerated task-set.



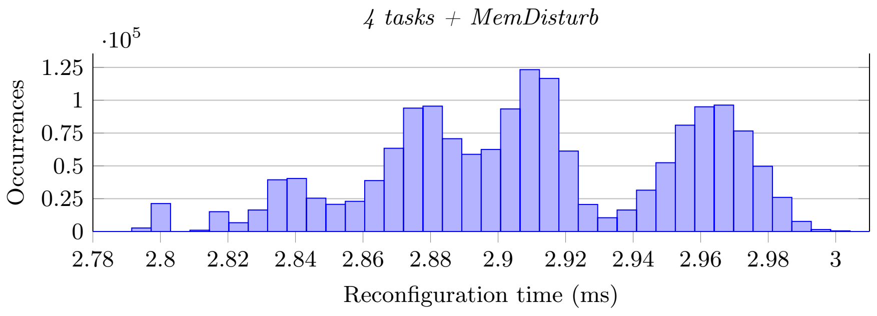


Figure 5.15: Distribution of reconfiguration times.

Experiment	Reconfiguration time (ms)		
	Min	Avg	Max
4 tasks (Section 5.3.2)	2.791	2.820	2.846
4 tasks + MemDisturb	2.795	2.910	3.012

Table 5.7: Observed reconfiguration times.

interface connected to the internal AXI Interconnect. Unlike the processor and the HW-tasks connected to the AXI slave ports, the DevC subsystem is not directly connected to the DRAM controller. In fact, it contends the access to the DRAM controller with other peripherals in the PS side.

In general, the throughput achievable by the DevC DMA depends on the traffic conditions on the AXI Interconnect, and the load on the DRAM controller. Modeling the bus contention on the AXI Interconnect and evaluating its performance goes beyond the scope of this thesis. However, a first test was carried out to evaluate how a memory intensive SW-task interferes with the DevC, and hence affects reconfiguration times.

The task set used for this test includes the four tasks described in the experiment of Section 5.3.2, and an additional *memory intensive* software activity (referred to as *MemDisturb*) continuously running in background without invoking HW-tasks. The MemDisturb software activity performs memory transfers between two memory buffers of 32 MB. The sizes of the buffers exceed the size of the processor L2 cache. Therefore, such a memory transfers generate a continuous stream of request to the DRAM controller that simulates a memory intensive SW-task.

Table 5.7 compares the reconfiguration times with and without the MemDisturb activity. Figure 5.15 illustrates the reconfiguration times distribution in both cases. The results of this experiment show that, despite a memory intensive software activity can affect reconfiguration times, its impact is very small and in the order of 0.1 ms. We believe that this result, although preliminary and far from being complete, is encouraging for exploiting partial reconfiguration in real-time systems, where bounded reconfiguration delays are essential to guarantee the system predictability. Given the size of the partial bistreams (338 KB), the average observed throughput for the DevC amounts to 117 MB/s without MemDisturb and to 113 MB/s with MemDisturb.

Conclusions

This work presented an experimental study aimed at evaluating the use of dynamic partial reconfiguration for implementing a timesharing mechanism to virtualize the FPGA resource in heterogeneous platforms that also include a processor. Hence, an application consists of both software computational activities (running on the processor) and hardware modules implemented in programmable logic to be dynamically allocated on the FPGA, as requested by the software tasks. The temporal parameters involved in such a system (e.g., reconfiguration and execution times) have been profiled for a case study application. The achieved results are encouraging and clearly show that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to a fully static approach.

Besides the encouraging results, the experimental studies highlighted two major bottlenecks of today's platforms. First, all the evaluated FPGA platforms provide only a single reconfiguration interface, which is then contended by all the HW-tasks. Second, when the main memory is used to store both data and bistreams, an additional contention there exists on the Interconnect and the DRAM controller, which introduces further complications in the timing analysis. As a consequence, the presence of memories dedicated to bitstream storage would significantly improve both performance and predictability.

The issues that are still open include the design and the analysis of scheduling algorithms for HW-tasks, provided in Section 5.3.3, and the design of real-time operating system mechanisms to support such a dynamic approach and an improved inter-task communication, presented in Section 5.3.4.

5.3.3 Analysis

Framework and Modeling

This work considers a platform consisting of a processor and a DPR-enabled FPGA module that comprises b logic blocks. The FPGA and the processor share a common memory $\mathcal{M}$. The blocks of the FPGA module are statically partitioned into a set $P = \{P_1, ..., P_{n_P}\}$ of n_P *partitions*, where each partition P_k is composed of b_k logic blocks, with $\sum_{k=1}^{n_P} b_k \leq b$. Blocks are not shared among partitions. Furthermore, each partition P_k is split into n_k^S *slots* of b_k^S logic blocks, such that $\forall P_k \in P$, $n_k^S \cdot b_k^S \leq b_k$. Blocks are not shared among the slots.

As described in Section 5.3.1, a slotted approach is more suitable for real-time systems because reconfiguration delays are shorter and more predictable than in a slotless solution, since there is no overhead related to task allocation management and to instantiation of communication channels. On the other hand, a slotted approach introduces a granularity that may increase the wasted area of the FPGA. This phenomenon can be mitigated by a proper design of slots and partitions as a function of the tasks. However, this issue is not addressed in this thesis.

Hardware Task Model The activities executed on the FPGA are modeled as a set $\Gamma^H = \{\tau_1^H, ..., \tau_{n_H}^H\}$ of n_H HW-tasks. Each HW-task τ_i^H requires b_i logic blocks and has a *worst-case execution time* (WCET) C_i^H. A HW-task can execute only if it has been programmed on a slot of the FPGA.

The considered platform is equipped with a *FPGA reconfiguration interface* (FRI) able to dynamically reconfigure a slot at run-time by programming a specific HW-task τ_i^H. Each slot can accommodate at most one HW-task [25, 154]. As true in real-world platforms (such as [105, 190]), the following assumptions are made:

1. the FRI can reconfigure a slot without affecting the execution of the HW-tasks currently running in other slots;
2. no processor cycles are used for reconfiguring a slot (i.e., the FRI is an external peripheral, like DMA [204]); and

3. the FRI can program at most one slot at a time.

To program a given HW-task τ_i^H into a slot, the FRI has to program all its logic blocks, independently of the number b_i of logic blocks required by τ_i^H, because unused blocks have to be disabled to "clean" the previous slot configuration.

Each HW-task τ_i^H can be programmed in any of the slots belonging to a single partition. The partition hosting a HW-task τ_i^H is denoted as $P(\tau_i^H)$ and referred to as *affinity.* For all HW-tasks with affinity $P(\tau_i^H) = P_k$, it must be $b_i \leq b_k^S$.

The FRI is characterized by a *throughput* ρ, meaning that $r_k^S = b_k^S/\rho$ units of time are needed to program a slot of a given partition P_k. Hence, the time r_a needed to program a HW-task τ_a^H is $r_a = r_k^S \ : \ P(\tau_a^H) = P_k$.

Software Task Model The activities executed on the processor are modeled as a set $\Gamma^S = \{\tau_1, ..., \tau_{n_S}\}$ of n_S SW-tasks. Each SW-task can make use of HW-tasks to accelerate specific functions and is subject to timing constraints. In particular, each SW-task τ_i

- uses a set $\mathcal{H}(\tau_i) \subseteq \Gamma^H$ of m_i HW-tasks;
- alternates the execution of $m_i + 1$ *sub-tasks* (also referred to as *chunks*) with the execution of the m_i HW-tasks in $\mathcal{H}(\tau_i)$; thus, the execution of a SW-task τ_i can be represented as a sequence

$$\tau_i := \langle \tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \ldots, \tau_{i,m_i+1} \rangle,$$

  where $\{\tau_a^H, \tau_b^H, \ldots\} \in \mathcal{H}(\tau_i)$ and $\tau_{i,j}$ is the j-th sub-task of τ_i. Whenever the execution of a HW-task τ_a^H is requested, the corresponding SW-task *self-suspends* until the completion of τ_a^H. The beginning of the self-suspension phase coincides with the termination of the sub-task that issued a request for a HW-task. In a dual manner, the completion of a HW-task coincides with the release of the next sub-task.
- has a total WCET C_i, composed of the WCETs $C_{i,j}$ of all its sub-tasks $\tau_{i,j}$; that is,

$$C_i = \sum_{j=1}^{m+1} C_{i,j}.$$

- is periodically (or sporadically) released with a period (or minimum inter-arrival time) of T_i units of time, thus generating an infinite sequence of execution instances (denoted as jobs);
- is subject to timing constraints; that is, each of its jobs must complete its execution within a *deadline* D_i relative to its activation time.

Each HW-task can be used by at most one SW-task, that is

$$\bigcap_{\tau_i \in \Gamma^S} \mathcal{H}(\tau_i) = \emptyset.$$

Listing 5.2 reports the pseudo-code defining the implementation skeleton of a SW-task τ_i that uses $m_i = 2$ HW-tasks in the set $\mathcal{H}(\tau_i) = \{\tau_a^H, \tau_b^H\}$. The

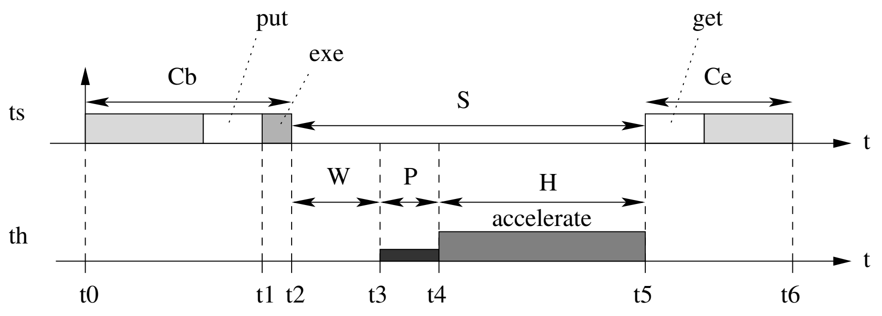


Figure 5.16: Execution behavior of a SW-task calling a HW-task.

statement `<...>` has been used to represent a generic set of instructions that are part of a computation executed by the SW-task on the processor.

```
TASK($\tau_i$)
{
  <...>
  <prepare input data for $\tau_a^H$>
  EXECUTE_HW_TASK($\tau_a^H$);
  <retrieve output data from $\tau_a^H$>
  <...>
  <prepare input data for $\tau_b^H$>
  EXECUTE_HW_TASK($\tau_b^H$);
  <retrieve output data from $\tau_b^H$>
  <...>
}
```

Listing 5.2: Pseudo-code of the implementation skeleton of a SW-task.

The SW-task illustrated in Listing 5.2 is described by the sequence $\langle\tau_{i,1}, \tau_a^H, \tau_{i,2}, \tau_b^H, \tau_{i,3}\rangle$: the first sub-task $\tau_{i,1}$ consists of lines 3-5, the second sub-task $\tau_{i,2}$ of lines 6-9 and the third sub-task $\tau_{i,3}$ of lines 10-11. `EXECUTE_HW_TASK(`τ_j^H`)` is a *blocking* system call, which is in charge of (i) requesting the execution of τ_j^H and (ii) *suspending* the execution of τ_i until the completion of τ_j^H. Note that at line 4, $\tau_{i,1}$ prepares the input data for τ_a^H. Similarly, $\tau_{i,2}$ retrieves the output data produced by τ_a^H (line 6) and prepares the input data for τ_b^H (line 8). Further details on the inter-task communication mechanism are discussed in Section 5.3.3.

Figure 5.16 illustrates the execution behavior of another SW-task τ_i:=$\langle\tau_{i,1}, \tau_a^H, \tau_{i,2}\rangle$, visualizing the delays experienced when requesting the execution of τ_a^H.

As clear from the figure, task τ_i is activated at time t_0. At time t_1, the first sub-task $\tau_{i,1}$ requests the execution of the HW-task τ_a^H and self-suspends its execution at time t_2, where $(t_2 - t_1)$ corresponds to the system overhead to issue the request. This example assumes that all the slots of partition $P(\tau_a^H)$ are busy (i.e., occupied by other HW-tasks that are currently executing), hence a delay Δ_a is introduced from time t_2 until time t_3, at which one slot of $P(\tau_a^H)$ becomes free. Once there is a free slot in $P(\tau_a^H)$, the HW-task can be programmed, from time t_3 to t_4, by using the FRI: such an operation takes at most r_a units of time, where $r_a = b_k^S/\rho$ (being k the affinity of τ_a^H).

After the programming phase, τ_a^H starts executing at time t_4 on the FPGA

b	total number of logic blocks in the FPGA
n_P	number of partitions in the FPGA
P_k	k-th partition in the FPGA
b_k	number of logic blocks in partition P_k
b_k^S	number of logic blocks in a slot of P_k
n_k^S	number of slots in partition P_k
ρ	throughput of the reconfiguration interface
r_k^S	time to program a slot of partition P_k
n_S	number of software tasks
n_H	number of hardware tasks
τ_i	i-th software task
$\tau_{i,j}$	j-th sub-task of the i-th software task
τ_a^H	a-th hardware task
$P(\tau_a^H)$	partition hosting the a-th hardware task
r_a	time to program the HW-task τ_a^H
C_a^H	worst-case execution time of HW-task τ_a^H
Δ_a	delay experienced by τ_a^H to wait for a free slot
b_a	number of logic blocks required by τ_a^H
C_i	worst-case execution time of SW-task τ_i
$C_{i,j}$	worst-case execution time of sub-task $\tau_{i,j}$
π_i	priority assigned to SW-task τ_i
T_i	period (or minimum inter-arrival time) of τ_i
D_i	relative deadline of SW-task τ_i
m_i	number of HW-tasks used by SW-task τ_i

Table 5.8: Symbols used throughout the chapter.

and completes at time t_5 within C_a^H units of time. Then, the SW-task is resumed and executes the second sub-task $\tau_{i,2}$, which completes at time t_6. Note that τ_i is suspended for the interval $[t_2, t_5]$, which is no longer than $S = \Delta_a + r_a + C_a^H$.

While the example presented above has a single SW-task, the system considered in this chapter includes multiple SW-tasks and HW-tasks that contend the resources available on the platform. This means that a SW-task τ_i can suffer a temporal *interference* from the execution of other SW-tasks that, if not properly managed, can determine the violation of its deadline D_i. Such interference also depends on the contention for the FPGA slots and the FRI caused by the other HW-tasks. For such reasons, a *scheduling infrastructure* is needed to support a set of concurrent HW-tasks and SW-tasks.

The symbols used in the chapter are summarized in Table 5.8.

Scheduling Infrastructure Each SW-task τ_i is assigned a fixed priority π_i, also inherited by all its sub-tasks. A SW-task is denoted as *ready* when (i) it has a pending job (i.e., a job released but not yet completed) and (ii) it is not self-suspended waiting for the completion of a HW-task. SW-tasks are assumed to be scheduled according to a fixed-priority (FP) preemptive scheduling algorithm, so that, at any point in time, the ready task with the highest priority is executed on the processor.

Besides the processor, two other resources are contented by SW-tasks: the slots in the FPGA partition (shared with other HW-tasks having the same affinity) and the FRI. Hence, multiple requests for such resources have to be scheduled. The overall scheduling infrastructure managing the slots and the FRI is based on a multi-level queue structure, illustrated in Figure 5.17.

The scheduling policies used for each resource are first described; then, is presented the scheduling rules that apply to every request for HW-tasks when

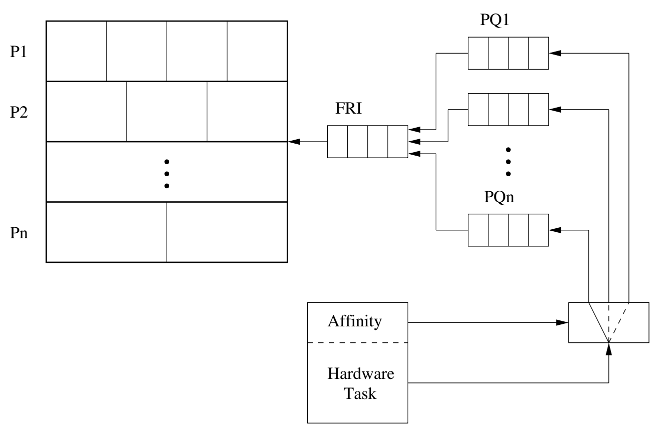


Figure 5.17: Scheduling infrastructure for HW-tasks requests in FRED.

traversing the multi-level queue structure of Figure 5.17.

Slot Scheduling For the purpose of scheduling, each slot can be *free* or *busy*. A busy slot can in turn be *active*, when there is a HW-task programmed on it that is executing, or *reserved*. A HW-task τ_i^H with affinity $P(\tau_i^H) = P_k$, that is waiting for a free slot in partition P_k, is kept in a queue Q_k managed according to a first-in-first-out (FIFO) policy. Note that such a scheduling policy guarantees a starvation-free progress mechanism. Moreover, it does not require preempting the execution of HW-tasks, which is known to be a challenging issue [74, 139] leading to non-negligible run-time overheads.

FRI Scheduling Whenever there are x free slots into a given partition P_k, such x slots are *reserved* for the first x HW-task requests waiting into Q_k which then have to contend the FRI to program their corresponding HW-task. While slots are shared only among the HW-tasks belonging to the same partition, the FRI is a single resource contented by all the requests for HW-tasks in the system. HW-task requests contending the FRI are kept in a queue denoted as Q^{FRI}.

In this chapter, slot programming requests are managed according to a *ticket-based scheduling* policy, which is described below and can be configured to be executed either in a preemptive or non-preemptive fashion. Please note that HW-task execution is assumed to be non preemptive to contain the preemption overhead associated to FPGA reconfigurations. Hence, here preemptive and non-preemptive policies are only related to the FRI programming phase. Under a non-preemptive policy, the programming phase cannot be interrupted, whereas under a preemptive policy, the programming phase can be interrupted to serve another programming request.

Ticket-based Scheduling The ticked-based scheduling policy is described by the following rules that apply to both non-preemptive and preemptive management of the FRI:

R1 Each execution request $\mathcal{R}_a$ for an HW-task τ_a^H is assigned a "ticket" marked with the absolute time $t(\mathcal{R}_a)$ at which $\mathcal{R}_a$ has been issued.

R2 Every partition queue Q_k and the FRI queue Q^{FRI} enqueues execution requests for HW-tasks by *increasing* ticket time.

R3 When a request $\mathcal{R}_a$ for HW-task τ_a^H is issued, $\mathcal{R}_a$ is inserted in the partition queue Q_k with $P_k = P(\tau_a^H)$.

R4 At any point in time t, for every partition queue Q_k, the first $\eta_k(t) \geq 0$ requests in Q_k are removed from Q_k and inserted in Q^{FRI}, where $\eta_k(t)$ is the number of *free* slots in P_k at time t. Contextually, these $\eta_k(t)$ slots become *reserved* (and hence *busy*).

R5 Once the HW-task τ_a^H related to a request $\mathcal{R}_a$ has been programmed onto a slot, $\mathcal{R}_a$ is removed from Q^{FRI}, that slot becomes *active*, and τ_a^H starts executing.

R6 When a HW-task τ_a^Hcompletes its execution, the corresponding slot becomes *free*.

The following scheduling rules distinguish between non-preemptive and preemptive management of the FRI. In the case of preemptive FRI scheduling, the following rule holds:

R-P1 Whenever Q^{FRI} is not empty, the FRI programs the HW-task related to the first request in Q^{FRI} (i.e., the one having the earliest ticket time).

For non-preemptive FRI scheduling the following rules hold:

R-NP1 When the FRI is programming a HW-task it cannot be interrupted to serve another request.

R-NP2 When the FRI completes a programming phase, or Q^{FRI} becomes not empty, the FRI starts programming the HW-task related to the first request in Q^{FRI}.

Example Figure 5.18 shows an example of preemptive FRI management schedule under FRED for an FPGA module containing two partitions P_1 and P_2, each consisting of a single slot.

The application consists of three SW-tasks: $\tau_1 = \langle \tau_{1,1}, \tau_a^H, \tau_{1,2}, \tau_b^H, \tau_{1,3} \rangle$, $\tau_2 = \langle \tau_{2,1}, \tau_c^H, \tau_{2,2} \rangle$, and $\tau_3 = \langle \tau_{3,1}, \tau_d^H, \tau_{3,2} \rangle$. The priority assignment is such that $\pi_1 > \pi_2 > \pi_3$. HW-tasks τ_a^H and τ_b^H share partition P_1 (i.e., $P(\tau_a^H) = P(\tau_b^H) = P_1$), whereas HW-tasks τ_c^H and τ_d^H share partition P_2 (i.e., $P(\tau_c^H) = P(\tau_d^H) = P_2$).

All the SW-tasks are synchronously released at time 0. Being the highest-priority one, τ_1 starts executing as first and at time $t = 1$ completes its sub-task $\tau_{1,1}$ by issuing a request $\mathcal{R}_a$ for HW-task τ_a^H. Contextually, τ_1 self-suspends its execution. According to Rule R3, $\mathcal{R}_a$ is inserted in the partition queue Q_1. Since partition P_1 is empty, at time $t = 1$ there is a free slot ($\eta_1(1) = 1$); hence, according to Rule R4, $\mathcal{R}_a$ is moved to Q^{FRI} and the slot of P_1 becomes reserved. Moreover, according to Rule R-P1, the FRI starts programming τ_a^H. At time $t = 5$, τ_a^H has been programmed and according to Rule R5 it starts executing.

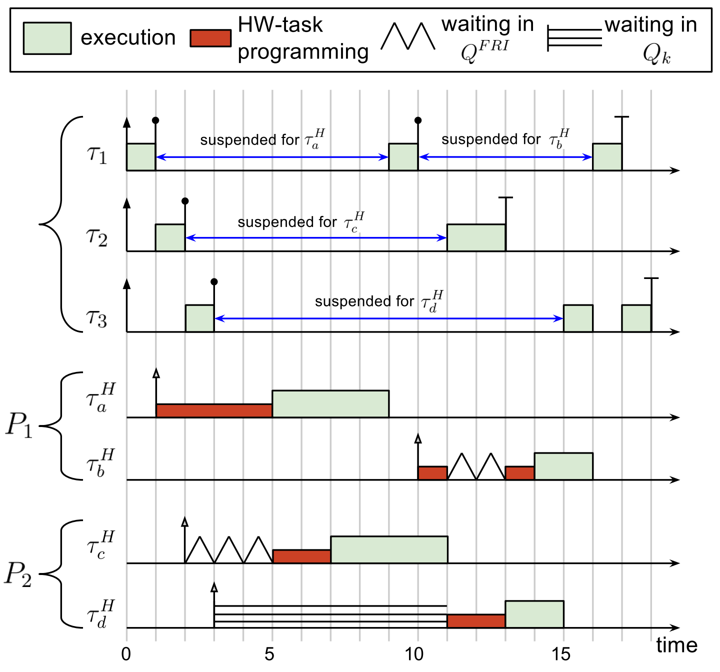


Figure 5.18: Example of preemptive FRI scheduling under FRED.

At time $t = 1$, τ_2 starts executing being the highest-priority SW-task ready. At time $t = 2$, τ_2 concludes its sub-task $\tau_{2,1}$ by issuing a request $\mathcal{R}_c$ for τ_c^H. According to Rule R3, $\mathcal{R}_c$ is inserted in the partition queue Q_2. Since partition P_2 is empty, at time $t = 2$ there is free slot ($\eta_2(2) = 1$); hence, according to Rule R4, $\mathcal{R}_c$ is moved to Q^{FRI} and the slot of P_2 becomes reserved. However, since $\mathcal{R}_c$ has a later ticket time than $\mathcal{R}_a$, $\mathcal{R}_c$ is delayed until τ_a^H has been programmed (time $t = 5$). Then, τ_c^H can be programmed and be executed.

At time $t = 2$, τ_3 is the highest-priority SW-task ready, thus it starts executing until time $t = 3$, when it terminates its first sub-task $\tau_{3,1}$ by issuing a request $\mathcal{R}_d$ for τ_d^H. According to Rule R3, $\mathcal{R}_d$ is inserted in the partition queue Q_2. However, being the slot of P_2 busy (specifically, reserved in [5,7] and active in (7,11]), $\mathcal{R}_d$ waits in Q_2 until time $t = 11$. At time $t = 11$, τ_c^H completes its execution, Rule R6 is applied and the slot of P_2 becomes free. According to Rule R4, $\mathcal{R}_d$ is moved to Q^{FRI}, the slot of P_2 becomes again busy (specifically, reserved) and τ_d^H starts to be programmed.

Now, consider again τ_1. At time 9, τ_a^H is completed and hence the sub-task $\tau_{1,2}$ can be released. At time 10, $\tau_{1,2}$ completes by issuing a request $\mathcal{R}_b$ for HW-task τ_b^H. By Rule R3 and Rule R4, $\mathcal{R}_b$ is inserted into Q^{FRI}. Being Q^{FRI} empty, τ_b^H starts to be programmed. However, as explained above, at time $t = 11$, $\mathcal{R}_d$ (issued by τ_3) is inserted into Q^{FRI}. Being $t(\mathcal{R}_d) = 3 < t(\mathcal{R}_b) = 10$, according to Rule-R-P1 the programming of τ_b^H is preempted to program τ_d^H until time $t = 13$. Hence in [11, 13) $\mathcal{R}_b$ is delayed. Finally, note that the FRI queue is not managed in a pure FIFO manner.

Communication Between SW and HW Tasks As stated in Section 5.3.3, SW-tasks make use of HW-tasks to accelerate specific computations; that is, a SW-task offloads a computation to the FPGA by requesting the execution of a HW-task and then retrieves the output of such a computation to continue the execution on the processor. As shown in Listing 5.2, the communication between a SW-task τ_i and a HW-task τ_a^H includes two phases:

1. sub-task $\tau_{i,j}$ prepares the input data for τ_a^H;
2. sub-task $\tau_{i,j+1}$ retrieves the data produced by τ_a^H.

It is worth observing that the approach used to enable such a communication can affect the real-time performance of the system by introducing different worst-case scenarios. For instance, suppose that the output data produced by a HW-task are stored in its internal memory area and that phase (ii) comprises a copy from the local memory of the HW-task to a memory area accessible by the SW-task. In such a case, the HW-task must remain programmed onto the FPGA module until the sub-task in charge of executing the phase (ii) will be executed, otherwise output data would be lost.

Due to the scheduling delays suffered by SW-tasks, the actual time a HW-task occupies a slot is hence dependent on SW-tasks' execution behavior. Longer slot occupation times increase the delays suffered by HW-tasks, which in turn increase the delays suffered by SW-tasks by inflating their suspension time when waiting for the completion of a HW-task. Such a circular dependency can originate pathological scenarios that significantly increase the worst-case response time of SW-tasks, thus making this approach not attractive for a real-time system.

To overcome this problem, FRED adopts a different approach inspired by the capabilities of state-of-the-art platforms, where the communication between SW-tasks and HW-tasks is supported by allowing HW-tasks to directly access the shared memory $\mathcal{M}$. Hence, the two communication phases are implemented as follows:

1. sub-task $\tau_{i,j}$ prepares the input data for τ_a^H in a memory area $\mathcal{M}_a^{\text{IN}}$ inside $\mathcal{M}$, and τ_a^H retrieves the input data by directly accessing $\mathcal{M}_a^{\text{IN}}$;
2. τ_a^H stores the output data into a memory area $\mathcal{M}_a^{\text{OUT}}$ inside $\mathcal{M}$, and $\tau_{i,j+1}$ retrieves them directly from $\mathcal{M}_a^{\text{OUT}}$, hence τ_a^H can release its slot as it finishes.

References (i.e., memory pointers) to both $\mathcal{M}_a^{\text{IN}}$ and $\mathcal{M}_a^{\text{OUT}}$ are assumed to be provided to the HW-task or known a priori. By adopting this solution, the time τ_a^H must hold a slot is totally decoupled from the scheduling delays of SW-tasks and is always upper-bounded by the WCET C_a^H plus the slot reconfiguration time r_a.

As done in most real-time analysis, bus contention times due to the interaction between HW-tasks and SW-tasks can be accounted in the WCETs. Finally, please note that such a communication approach is not limited to platforms having a main memory shared between the processor and the FPGA module, but it can also be used in platforms where a dedicated memory is reserved for such a communication. Indeed, the latter solution is more suitable for safety-critical systems requiring a higher level of predictability.

Real-time Analysis

The goal of this section is to derive a sufficient response-time analysis for the SW-tasks running under FRED. That is, for each SW-task τ_i, this section provides an upper-bound R_i on its maximum response-time such that the system is guaranteed to be schedulable if

$$\forall \tau_i \in \Gamma^S,\ R_i \leq D_i. \tag{5.15}$$

The upper-bounds are derived by building on Nelissen et al.'s [141] response-time analysis for *real-time fixed-segment self-suspending tasks* (SS-tasks). The SS-task model is a generic model for real-time computational activities where multiple execution phases are alternated to self-suspension phases, exactly like the execution behavior of the SW-tasks under FRED. Similarly to a SW-task, a SS-task τ_ℓ alternates the execution of $m_\ell + 1$ sub-tasks, each having WCET $C_{\ell,j}$ (j goes from 1 to $m_\ell + 1$) and m_ℓ suspension phases, each lasting *at most* $S_{\ell,j}$ time units (j goes from 1 to m_ℓ). The C, T and D parameters of a SS-task are consistently defined as the corresponding ones of a SW-task, as stated in Section 5.3.3.

Each SW-task τ_i can hence be mapped (i.e., translated) into a SS-task τ_ℓ according to the following rules:

1. $C_{\ell,j} = C_{i,j}$, $\forall j = 1, \ldots, m_i + 1$;
2. $S_{\ell,j} = r_a + C_a^H + \Delta_a$, $\forall j = 1, \ldots, m_i$, where $\tau_a^H \in \mathcal{H}(\tau_i) \ :\ \tau_i := \langle \ldots, \tau_{i,j}, \tau_a^H, \tau_{i,j+1}, \ldots \rangle$.
3. unless differently specified, $\mathcal{X}_\ell = \mathcal{X}_i$, where $\mathcal{X}_i$ is a parameter of τ_i.

Intuitively speaking, Rule 1 maps each sub-task of the SW-task τ_i into a sub-task of the SS-task τ_ℓ, while Rule 2 defines a suspension phase of the SS-task for each HW-task τ_a^H used by τ_i; such a suspension phase includes the reconfiguration time r_a, the WCET C_a^H and the the worst-case delay Δ_a suffered by the HW-task under the FRED scheduling infrastructure. Finally, Rule 3 enforces that the other parameters of the SS-task τ_ℓ are equal to the ones of the SW-task τ_i.

Please note that all the parameters mentioned in the rules above are known, except for the delay Δ_a: computing a safe upper-bound on such a delay is the main challenge of the proposed real-time analysis.

For the sake of completeness, the next section briefly summarizes the Nelissen et al.'s response-time analysis for SS-tasks.

Summary on Nelissen et al.'s Analysis The response-time analysis of fixed-priority SS-tasks is a problem studied since several years in the real-time community. However, Nelissen et al. [141] discovered a number of errors in many papers concerning the analysis of SS-tasks, and proposed a safe and accurate response-time analysis for SS-tasks based on *mixed-integer linear programming* (MILP).

A MILP formulation is instantiated for each SS-task τ_ℓ whose objective is to *maximize* the response-time upper-bound $R_{\ell,j}$ of each sub-task $\tau_{\ell,j}$ composing τ_ℓ. Each response-time upper-bound is expressed as $R_{\ell,j} = C_{\ell,j} + \sum_{\tau_k \in hp(\tau_\ell)} \sum_{z=1}^{m_k} NI_{\ell,k,z} \times C_{k,z}$ where $hp(\tau_\ell)$ is the set of tasks that have higher priority than τ_ℓ and $NI_{\ell,k,z}$ is an *integer optimization variable* modeling the number of jobs of $\tau_{\ell,j}$ interfering with τ_ℓ. Several constraints are enforced to bound the value of $NI_{\ell,k,z}$: please refer to [141] for further details. Finally, once the MILP has been solved, the total response-time upper-bound R_ℓ of τ_ℓ is computed as $R_\ell = \sum_{j=1}^{m_\ell} R_{\ell,j} + \sum_{j=1}^{m_\ell-1} S_{\ell,j}$.

Upper-bound for the Delay Δ_a As stated above, computing a response-time upper-bound for SW-tasks is crucial to bound the maximum time a SW-task can be suspended to wait for the completion of a HW-task. This in turn requires bounding the delay Δ_a suffered by each HW-task request $\mathcal{R}_a$, which is the goal of this section.

As a first step, the following lemma establishes that the ticket-based scheduling policy introduced in Section 5.3.3 is *work-conserving.*

Lemma 1. *A HW-task request $\mathcal{R}_a$ for τ_a^H with affinity to partition $P_k = P(\tau_a^H)$ is delayed at time t if and only if either*

- all *the n_k^S slots of P_k are* busy *serving other HW-tasks $\tau_b^H \neq \tau_a^H$ with $P(\tau_b^H) = P_k$; or*
- *the FRI is busy programming other HW-tasks $\tau_b^H \neq \tau_a^H$.*

Proof. A HW-task request $\mathcal{R}_a$ can be delayed either (i) when it is in the partition queue Q_k or (ii) when it is in the Q^{FRI} queue. In case (i), according to Rule 4, $\mathcal{R}_a$ can wait as long as *all* the n_k^S slots of P_k are *busy.* In case (ii), here is made the distinction between preemptive and non-preemptive FRI management. Under preemptive FRI management, being Q^{FRI} non-empty (at least $\mathcal{R}_a$ is inside Q^{FRI}), by Rule R-P1 the FRI is still programming a HW-task τ_a^H in a reserved slot in $P(\tau_a^H)$. By Rule R4, for every request inserted into Q^{FRI} there is a reserved slot in the corresponding partition. The same argument holds under non-preemptive FRI management by considering Rules R-NP1 and R-NP2. $\square$

By relying on the fact that SW-tasks issue HW-task requests in a sequential fashion, it is possible to establish another key property.

Lemma 2. *Let $\mathcal{R}$ be an arbitrary HW-task request issued by a SW-task τ_i at time $t(\mathcal{R})$ and let $t_s(\mathcal{R})$ be the time at which $\mathcal{R}$ is removed from Q^{FRI} (i.e., $\mathcal{R}$ is satisfied). Let also $pend(\mathcal{R})$ be the set of pending HW-task requests during $[t(\mathcal{R}), t_s(\mathcal{R}))$ that have earlier (or equal) ticket time, i.e., $\forall \mathcal{R}_a \in pend(\mathcal{R})$, $t(\mathcal{R}_a) \leq t(\mathcal{R})$. If HW-task requests are serialized as specified in the model presented in Section 5.3.3, then each SW-task $\tau_j \neq \tau_i$ can possibly issue* at most one *HW-task request in $pend(\mathcal{R})$.*

Proof. By contradiction. Suppose that $pend(\mathcal{R})$ contains more than one HW-task request from SW-task τ_j, say $\mathcal{R}_a$ and $\mathcal{R}_b$, respectively issued at times $t(\mathcal{R}_a)$ and $t(\mathcal{R}_b)$ and satisfied at times $t_s(\mathcal{R}_a)$ and $t_s(\mathcal{R}_b)$. Without loss of generality assume $t(\mathcal{R}_a) \leq t(\mathcal{R}_b)$. By definition of set $pend(\mathcal{R})$, it results $t(\mathcal{R}_a) \leq t(\mathcal{R}_b) \leq t(\mathcal{R})$, $t_s(\mathcal{R}_a) > t(\mathcal{R})$ and $t_s(\mathcal{R}_b) > t(\mathcal{R})$. Hence is obtained $t(\mathcal{R}_a) \leq t(\mathcal{R}_b) \leq t(\mathcal{R}) < t_s(\mathcal{R}_a)$, which implies that $\mathcal{R}_b$ has been issued before the completion of $\mathcal{R}_a$. This contradicts the assumption, according to which HW-task requests issued by the same SW-task are serialized. Hence, the lemma follows. $\square$

It is now possible to derive the upper-bound for the delay Δ_a. To this end, it is necessary to distinguish between preemptive and non-preemptive management of the FRI.

Preemptive FRI Management

Theorem 4. *Consider an arbitrary HW-task request $\mathcal{R}_a$ for τ_a^H issued by a SW-task τ_i. Let $P_k = P(\tau_a^H)$ be the affinity of τ_a^H. Under* preemptive *management of the FRI, the maximum delay Δ_a incurred by $\mathcal{R}_a$ is upper-bounded by*

$$\overline{\Delta_a^P} = \sum_{\tau_j \neq \tau_i} \max_{\tau_b^H \in \mathcal{H}(\tau_j)} \left\{ \Delta_b^{slot} + r_b \right\} \tag{5.16}$$

where

$$\Delta_b^{slot} = \begin{cases} \frac{C_b^H}{n_k^S} & \text{if } P(\tau_b^H) = P_k \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Let $\mathcal{X}$ be the set of HW-task requests that delay $\mathcal{R}_a$. Since here is considered a preemptive management of the FRI, Rule R-P1 applies. Because of such a rule and the FIFO ordering of the partition queue Q_k, $\mathcal{R}_a$ can only be delayed by other requests that have *earlier* (or equal) ticket time. Hence, $\forall \mathcal{R} \in \mathcal{X}$, $t(\mathcal{R}) \leq t(\mathcal{R}_a)$.

To help the presentation the set of HW-tasks $\Gamma_{\mathcal{X}}^H$ is defined by mapping each HW-task request $\mathcal{R} \in \mathcal{X}$ into the corresponding HW-task $\tau^H \in \Gamma_{\mathcal{X}}^H$. By Lemma 2, each SW-task $\tau_j \neq \tau_i$ can have issued at most one HW-task request in $\mathcal{X}$. Hence, $\forall \tau_i \neq \tau_j, |\Gamma_{\mathcal{X}}^H \cap \mathcal{H}(\tau_j)| \leq 1$. Moreover, since one HW-task cannot be used by multiple SW-tasks (i.e., $\bigcap_{\tau_j \in \Gamma^S} \mathcal{H}(\tau_j) = \emptyset$), each request in $\mathcal{X}$ corresponds to *one and only one* HW-task in $\Gamma_{\mathcal{X}}^H$, hence $|\Gamma_{\mathcal{X}}^H| = |\mathcal{X}|$.

The total workload W that can delay $\mathcal{R}_a$ cannot be greater than the sum of **(i)** the execution time of HW-tasks in $\Gamma_{\mathcal{X}}^H$ and **(ii)** the reconfiguration time of HW-tasks in $\Gamma_{\mathcal{X}}^H$. In addition, by Lemma 1, $\mathcal{R}_a$ cannot be delayed by the execution of HW-tasks τ_b^H with $P(\tau_b^H) \neq P_k$. Hence, it is

$$W \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) = P_k}} \left(C_b^H + r_b \right) + \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) \neq P_k}} r_b,$$

and being HW-task requests scheduled in a work-conserving manner (by Lemma 1), $\Delta_a \leq W$ holds.

Now, note that any interval in which $\mathcal{R}_a$ is delayed can be considered as an alternating sequence of sub-intervals of two types:

1. *type X*: if Q^{FRI} contains at least a request with a ticket time less than $t(\mathcal{R}_a)$, i.e., $\exists \mathcal{R} \in Q^{FRI} \; : \; t(\mathcal{R}) \leq t(\mathcal{R}_a)$);
2. *type Y*: otherwise, i.e., $\nexists \mathcal{R} \in Q^{FRI} \; : \; t(\mathcal{R}) \leq t(\mathcal{R}_a)$.

Being such intervals complementary, this classification is well defined.

Let Δ^X be the total delay suffered by $\mathcal{R}_a$ during intervals of type X and let Δ^Y be the one suffered during intervals of type Y. Hence, the total delay is given by $\Delta_a = \Delta^X + \Delta^Y$. To derive a bound on Δ_a, it is possible to proceed by deriving a bound on each of these two terms.

Type X) Consider an arbitrary time instant during an interval of type X. Let $\mathcal{R}$ be the request at the head of Q^{FRI}. By definition of type X interval and Rule R2, $t(\mathcal{R}) \leq t(\mathcal{R}_a)$: hence $\mathcal{R}$ contributes to the workload W. According to Rule R-P1, the FRI is programming the HW-task related to $\mathcal{R}$, and hence $\mathcal{R}_a$ is delayed anyhow by such an operation.

In total, $\mathcal{R}_a$ cannot be delayed by more than the overall reconfiguration times in the workload W, hence $\Delta^X \leq \sum_{\tau_b^H \in \Gamma_{\mathcal{X}}^H} r_b$.

Type Y) In this case, the queue Q^{FRI} can be *empty* or *non-empty.* If Q^{FRI} is empty, being $\mathcal{R}_a$ delayed, it must be waiting into its partition queue Q_k. If Q^{FRI} is non-empty, $\mathcal{R}_a$ cannot be into Q^{FRI} otherwise it would not be delayed according to Rule R2 and Rule R-P1 (i.e., it would be at the head of Q^{FRI}); hence $\mathcal{R}_a$ is waiting in Q_k anyway.

According to Rule 4, if $\mathcal{R}_a$ waits into Q_k, then all the n_k^S slots of P_k are busy. Moreover, none of these slots can be reserved: this holds because of the FIFO ordering of Q_k (in fact no request with affinity P_k can be into Q^{FRI}). Hence, in this case, *all* the n_k^S slots of P_k are *active* serving the execution of HW-tasks τ_b^H with $P(\tau_b^H) = P_k$. As a consequence, if $\mathcal{R}_a$ is delayed by Δ^Y time units across all intervals of type Y, then partition P_k served $n_k^S \cdot \Delta^Y$ execution time units.

By looking at the workload W that can interfere with $\mathcal{R}_a$, it must be that

$$n_k^S \cdot \Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) = P_k}} C_b^H$$

and hence

$$\Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) = P_k}} \frac{C_b^H}{n_k^S}.$$

Rewriting the expression for Δ_a becomes

$$\Delta_a = \Delta^X + \Delta^Y \leq \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) = P_k}} \left(\frac{C_b^H}{n_k^S} + r_b \right) + \sum_{\substack{\tau_b^H \in \Gamma_{\mathcal{X}}^H \\ P(\tau_b^H) \neq P_k}} r_b. \tag{5.17}$$

The upper-bound on Δ_a follows by maximizing Equation (5.17) over all possible sets $\Gamma_{\mathcal{X}}^H$. Since by construction of $\Gamma_{\mathcal{X}}^H$ each SW-task $\tau_j \neq \tau_i$ can have at most one request for a HW-task $\tau_b^H \in \Gamma_{\mathcal{X}}^H$, Equation (5.16) accounts for the maximum contribution to Equation (5.17) given by each SW-task $\tau_j \neq \tau_i$. □

Non-preemptive FRI Management Building on the bound $\overline{\Delta_a^P}$ stated by Theorem 4, it is possible to derive a bound on the delay incurred in the case of non-preemptive FRI.

Theorem 5. *Consider an arbitrary HW-task request $\mathcal{R}_a$ for τ_a^H issued by a SW-task τ_i. Let $P_k = P(\tau_a^H)$ be the affinity of τ_a^H. Under* non-preemptive *management of the FRI, the maximum delay Δ_a incurred by $\mathcal{R}_a$ is upper-bounded by*

$$\overline{\Delta_a^{NP}} = \overline{\Delta_a^{P}} + NH_k^{max} \times r_k^{max} \tag{5.18}$$

where

$$NH_k^{max} = \left|\{\tau_b^H \in \Gamma^H \ : \ P(\tau_b^H) = P_k\}\right|$$

and

$$r_k^{max} = \max_{\tau_b^H \in \Gamma^H} \{r_b \ : \ P(\tau_b^H) \neq P_k\}.$$

Proof. Any interval in which $\mathcal{R}_a$ is delayed can be considered as an alternating sequence of sub-intervals of two types:

1. *type X*: if Q^{FRI} contains at least a request corresponding to a HW-task τ_b^H with $P(\tau_b^H) = P_k$ (i.e., same partition of τ_a^H) waiting into Q^{FRI};
2. *type Y*: otherwise.

Note that these intervals have a different definition with respect the ones used in the proof of Theorem 4, but the same properties apply (i.e., the classification is well defined). Each type is later considered separately.

Type X) Consider a single interval of type X. Because of the FIFO ordering of partition queues Q_k, whenever $\mathcal{R}_a$ is delayed (during the considered interval), it must be waiting for the reconfiguration of HW-tasks with affinity P_k (present into Q^{FRI} by definition). According to Rule R-NP1 and Rule R-NP2, their corresponding request (or $\mathcal{R}_a$ itself) can be delayed by **(i)** other requests with lower (or equal) ticket time that are into Q^{FRI}; and **(ii)** *at most one* request with *higher* ticket time which is (non-preemptively) served by the FRI. Because of the FIFO ordering of partition queue Q_k, this latter request must be related to a HW-task with affinity $\neq P_k$. In case (i), the same consideration argued in the proof of Theorem 4 holds (preemptive FRI). In case (ii), the delay suffered by $\mathcal{R}_a$ cannot be higher than $r_k^{max} = \max_{\tau_b^H \in \Gamma^H}\{r_b \ : \ P(\tau_b^H) \neq P_k\}$ time units. By Rule R-NP2, such a delay can occur at most once for each interval of type X.

The total number of intervals of type X is maximized when, during each of such intervals, there is only *one* request into Q^{FRI} that corresponds to a HW-task with affinity P_k. Hence, such a number is bounded by the total number of HW-tasks with affinity P_k, given by $NH_k^{max} = |\{\tau_b^H \in \Gamma^H \ : \ P(\tau_b^H) = P_k\}|$. Hence, the total delay in case (ii) across all intervals of type X is upper-bounded by $NH_k^{max} \times r_k^{max}$.

Type Y) By definition, Q^{FRI} contains no requests corresponding to HW-tasks with the same affinity of τ_a^H. This clearly implies that $\mathcal{R}_a$ is either completed (and hence not yet delayed) or waiting inside Q_k. This is the same situation discussed in the proof of Theorem 4 when considering the term Δ^Y.

In summary, the total delay suffered by $\mathcal{R}_a$ in intervals of type Y, plus the delay in case (i) during intervals of type X, is bounded by the upper-bound $\overline{\Delta_a^P}$ of the delay suffered under preemptive FRI management. Hence, the theorem follows. □

Experimental Results

This section presents a set of experiments aimed at evaluating the performance of the FRED scheduling infrastructure in terms of schedulability analysis with synthetic workload. The experiments are performed to verify the schedulability under different architecture configurations and are obtained applying the sufficient response time analysis presented in Section 5.3.3.

Task Set and Architecture Generation The synthetic workload has been generated as follows.

Hardware Architecture The FRI throughput is defined as $\rho = 100$ and the total number of logic blocks is set as $b = 1,000,000$: the ratio between these two values yields an FRI throughput similar to the one observed in Section 5.3.2. The logic blocks of the FPGA are equally distributed among the partitions. The same holds for the logic blocks of each partition, which are equally distributed among its slots.

SW and HW-tasks For simplicity, the focus in this chapter is on the case where each SW-task accesses a single HW-task, i.e., $m_i = 2, \forall \tau_i \in \Gamma^S$. Because of this restriction, HW-tasks are denoted with the same index of the corresponding SW-task (i.e., τ_i^H is the HW-task used by SW-task τ_i). For each partition P_k, a bucket of possible task periods is defined according to the following rules: (i) the interval of periods covered by any two buckets do not overlap; (ii) all the values in every bucket are in the range $\left[10^5, 10^6\right] \mu s$. For each SW-task τ_i, a random period T_i is chosen from the bucket corresponding to partition $P(\tau_i^H)$ and then removed from the bucket. The UUniFast algorithm [30] is used to generate the utilization factor U_i of each SW-task τ_i, such that $\sum_{\tau_i \in \Gamma^S} U_i = U$, where U is parameter defined in the experiments. The minimum task utilization is set to $U^{min} = 0.005$. The WCET of each SW-task τ_i is then computed as $C_i = U_i \cdot T_i$. Such a value has been then randomly split to obtain the WCETs $C_{i,1}$ and $C_{i,2}$ of each sub-task. Finally, a parameter U^H has been defined to "mimic" a notion of utilization of the FPGA (also referred to as *hardware utilization*). Note that, due to the intrinsic interaction between SW- and HW-tasks, this parameter cannot be related to a pure concept of utilization like the parameter U. Again, the UUniFast algorithm [30] algorithm was used to generate the hardware utilization for each hardware task as follows: $\forall \tau_i^H \in \Gamma^H, C_i^H = U_i^H \cdot T_i$. Like U, U^H is another parameter varied in the experiments. Task priorities π_i are assigned according to the rate-monotonic policy.

Experiments on Schedulability Analysis This set of experiments has been carried out to measure the schedulability ratio of the tested task sets under four different configurations:

1. *Static*: the FPGA is supposed to have an infinite area, so that all the HW-tasks are statically assigned to a fixed slot, thus reconfiguration is not needed.

2. *FRED-P*: the proposed approach is used with preemptive FRI management and scheduling delays are computed according to Theorem 4;

3. *FRED-NP*: same as FRED-P but with non-preemptive FRI management and delays computed by Theorem 5;

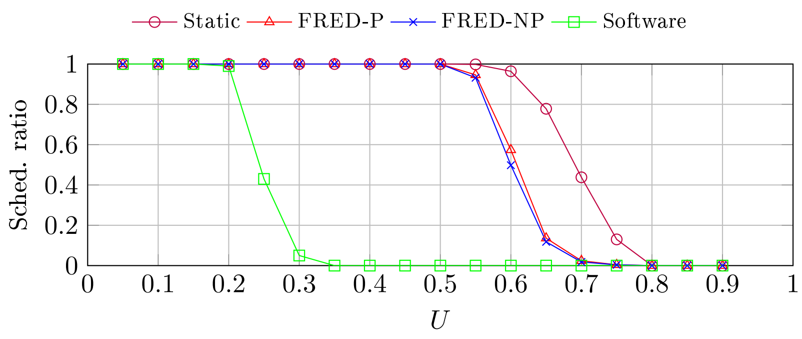


Figure 5.19: Schedulability ratio as a function of U.

4. *Software*: all task sets are implemented in software. Worst-case execution times of the resulting software implementation of HW-tasks are computed considering a speedup factor Φ, assumed to be the same for all the HW-tasks.

A first experiment varied the utilization U of a fixed number of tasks (both HW and SW), using $n_P = 3$ partitions, each with $n_k^S = 2$ slots, and 3 HW-tasks with affinity to each partition. The total number of tasks n_S has been chosen to overload the system: each partition P_k has $n_k^S + 1$ tasks, hence *all the tested task sets are not feasible without enabling DPR.* Figure 5.19 reports the results of an experiment where U was varied from 0.05 to 0.95 with a step of 0.05, with $\Phi = 1$ and $U^H = 0.1$. Please keep in mind that the small value of U^H cannot be interpreted as in the classical software semantics.

As evident from the plots, the *Software* approach quickly degrades at utilization values that are much lower than the ones at which *FRED-P* and *FRED-NP* starts being no more able to schedule the task sets. Even with an *unrealistic* and limit-case value for the speedup ($\Phi = 1$), this happens because the FPGA allows for intrinsic parallelism with respect to a single processor. The small difference between *FRED-P* and *FRED-NP* is due to the fact that the reconfiguration time (chosen according the profiling of Section 5.3.2) results almost negligible with respect to the generated execution times of HW-tasks. The *Static* approach represents a theoretical upper-bound (i.e, not practically achievable because assumes FPGAs with infinite area), whose performance depends on the absence of reconfiguration times and contention for slots and the FRI. In this experiment FRED obtains a schedulability ratio higher than 50% up to $U = 0.6$, outperforming the pure *Software* approach. Moreover, the results are quite close to the ideal scenario provided by a fully *Static* approach.

Figure 5.20 reports the result of another experiment where the the software utilization has been fixed to $U = 0.1$ and U^H has been varied from 0.05 to 0.95 with a step of 0.05. Also in this case, FRED outperforms pure *Software* approach, guaranteeing more than 50% of the tested task sets up to $U^H = 0.4$.

A third experiment has been carried out to investigate the benefits of FRED for applications that fully saturate the FPGA area and hence cannot be extended to include additional tasks without exploiting DPR. Here are considered systems with 2 partitions, 2 slots per partition and $\Phi = 3$. Starting from a fixed scenario ($U^H = 0.1$, $U = 0.1$) where all the slots are occupied by HW-tasks, Figure 5.21

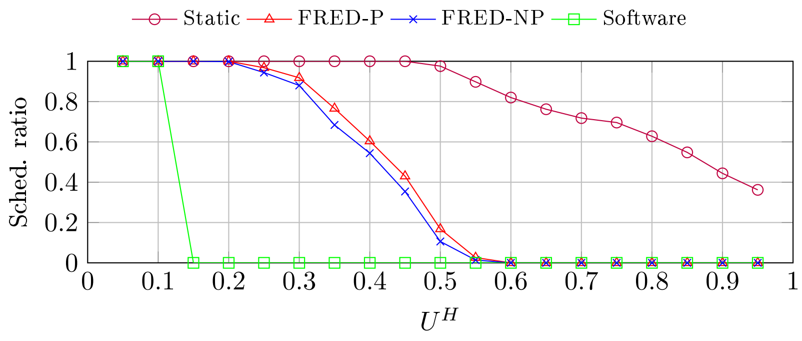


Figure 5.20: Schedulability ratio as a function of U^H.

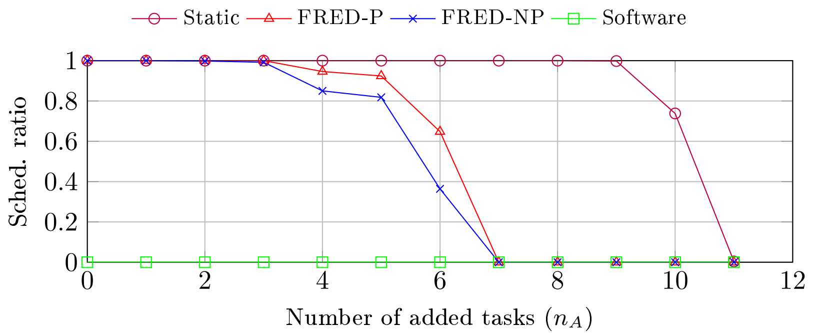


Figure 5.21: Schedulability ratio as a function of the number n_A of added tasks.

reports the schedulability ratio obtained by *adding* n_A *new tasks* (each consisting of a SW-task and the corresponding HW-task), where n_A was varied from 0 to 12. Each added task τ_i has $U_i^H = 0.05$ and $U_i = 0.05$, while the other parameters were generated as in the previous experiments. The affinity of each new task was chosen as specified in Section 5.3.3. This experiment clearly shows that FRED allows guaranteeing real-time applications extended with *more than 5 tasks*, which otherwise could not be executed.

Conclusions

This chapter presented a framework for supporting the development of safety-critical real-time applications on computing platforms that include a processor and an FPGA module with dynamic partial reconfiguration capabilities.

After providing a model of the platform and the computational activities, a scheduling infrastructure was proposed to bound the delays experienced by the tasks, and a response-time analysis was derived to verify the schedulability of safety-critical applications with real-time constraints. The experimental results performed on synthetic workloads showed the performance of the analysis in different scenarios.

5.3.4 Operating System Support to FRED

This section proposes a design to support the FRED framework on the Xilinx Zynq-7000 System-on-Chip (SoC) which has been chosen as the reference platform for this work, as detailed in [146]. The Zynq-7000 is a popular heterogeneous platform that includes a dual-core ARM Cortex-A9 processor and a Xilinx 7-Series FPGA fabric. The internal structure of the Zynq-7000 is divided into two main functional blocks: the *processing system* (PS) and the *programmable logic* (PL) [204]. The PS includes the ARM Cortex-A9 processors, interfaces for external memories, a small amount of on-chip RAM memory, and the I/O peripherals. The subsystems in the PS are interconnected among themselves and to the custom logic configured on the PL through an *AMBA AXI* system bus. The main interconnection between the PS and PL consists of a set of memory-mapped AXI (AXI for simplicity) interfaces exported by the PS side to the PL side.

System support design The proposed design is illustrated in Figure 5.22. In order to support the deployment of dynamically-reconfigured hardware accelerators on the PL, the FPGA area is divided into two main regions: a *static* region and a *reconfigurable* region (denoted by the striped boxes in Figure 5.22). The static region contains part of the logic that is needed to realize the communication infrastructure, namely a set of AXI Interconnects (discussed in Section 5.3.4) and other support modules. Following the specifications of the FRED framework, the reconfigurable region is organized by following a slotted scheme to host the hardware accelerators.

To implement the shared-memory communication paradigm between SW-tasks and HW-tasks, each hardware accelerator must be able to read and write a memory area that is also accessible from the processors. The Zynq-7000 provides three alternatives for implementing such memory areas: (i) using the internal on-chip memory, (ii) reserving part of the FPGA area to implement a

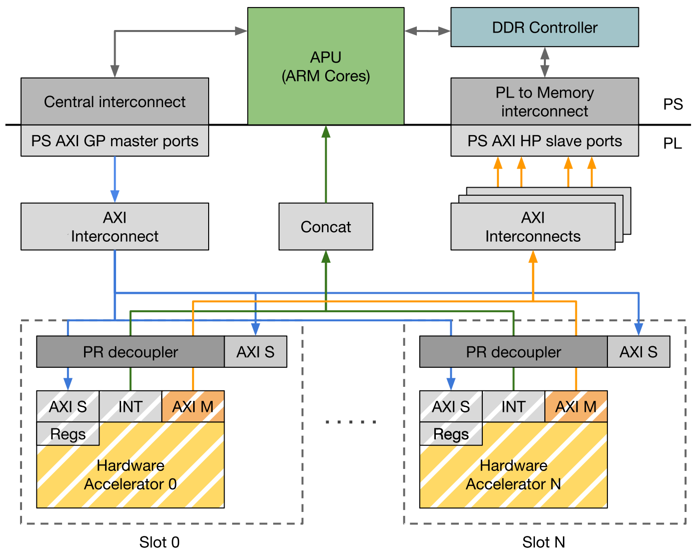


Figure 5.22: Support design for the Zynq SoC.

custom memory (using BRAM logic blocks), or (iii) using the main (off-chip) DRAM memory.

Alternative (i) is not viable since the on-chip memory is too small (256 Kb) and hence may not be suitable for supporting the shared-memory communication with multiple HW-tasks. Alternative (ii) determines a waste of the FPGA area, as the synthesis of efficient hardware accelerators generally requires BRAM logic blocks. Conversely, alternative (iii) allows using high-performance AXI interfaces that are directly connected to the DRAM controller. The availability of such interfaces suggests that the Zynq-7000 is prone to support this approach. As a consequence, the main DRAM memory has been selected for implementing the shared-memory paradigm.

Hardware accelerators must be capable of receiving control commands and arguments from the processor and sending synchronization signals to notify their completion. Since each slot of a partition P must be able to host any of the hardware accelerators that implement the HW-tasks with affinity to P, it necessary to define a *common interface*.

Common Interface The proposed interface consists of (i) an AXI master interface, (ii) an AXI slave interface exporting a set of control registers and eight 32-bit data registers, and (iii) an interrupt signal. The AXI master interface (denoted as AXI M in Figure 5.22) has been provided to allow the hardware accelerators to access the DRAM memory through the PS DDR controller. The control registers allow controlling the execution and the state of each hardware accelerator. Data registers can be used for manifold purposes depending on the specific function implemented by the hardware accelerator.

The most common usage consists in storing pointers to memory area in the DRAM (to implement shared-memory communication) or storing control parameters of the HW-tasks. The AXI slave interface (denoted as AXI S in Figure 5.22) is then used to map the control and data registers into the system memory space, hence making them available from the PS. Finally, the interrupt signal (denoted as INT in Figure 5.22) is used to notify the completion of the HW-task to the PS.

Dynamic Partial Reconfiguration In the Zynq-7000 SoC, the FPGA fabric can be fully or partially (re)configured under the control of the software running in the PS using the *device configuration* (DevC) subsystem. Internally, the DevC includes an interface to the *processor configuration access port* (PCAP) and a DMA engine that can be programmed to transfer a bitstream from the main DRAM memory to the PL configuration memory.

Each hardware accelerator corresponds to a bitstream. However, Xilinx tools do not support the relocation of bitstreams [190], i.e., the same bitstream cannot be used to program the same hardware accelerators in different slots. Since FRED requires that a hardware accelerator can be programmed onto different slots (depending on their availability at run-time), it is necessary to synthesize a bitstream for each slot of the partition to which the corresponding HW-task has affinity. Note that this is not relevant for memory consumption, as bitstreams are typically in the order of a few megabytes.

Slot Decouplers The reconfiguration process may generate transient glitches that can cause troublesome spurious transactions [190]. To solve this issue each slot is protected by a *partial reconfiguration decoupler* (denoted as PR decoupler in Figure 5.22), which is used to tie the interface signals to safe logic values. Each decoupler is controlled by the PS by means of a single control register, which is mapped into the memory space using an AXI slave interface.

Interconnections In the proposed design, the AXI master interfaces exported by each slot are connected to a set of AXI interconnects [195] modules. The proposed connection scheme is based on the rationale of equally distributing the memory bandwidth across the slots using fair arbitration [195]. More articulated connection schemes may be enabled by a fine-grained analysis of the interference incurred by the memory transactions, which is out of the scope of this chapter and is left as an open challenge. The AXI slave interfaces of the hardware accelerators and the decouplers are connected to a single AXI interconnect, which is turn connected to one of the Zynq general purpose master ports (denoted as PS AXI GP in Figure 5.22). Note that this does not constitute a bottleneck, since the PS is the only master. Finally, the interrupt signals exported by the slots are gathered together in a vector signal and routed to the IRQ_F2P port of the PS.

Linux Support

This section describes the implementation of the FRED framework for the GNU/Linux operating system. The FRED software support has been designed in a modular fashion relying, as much as possible, on userspace implementation to improve maintainability, safety, and extendability.

The internal architecture of the system is shown in Figure 5.23. The central component is a userspace daemon, named *FRED server*, which is in charge of managing acceleration requests from SW-tasks. The server relies upon two

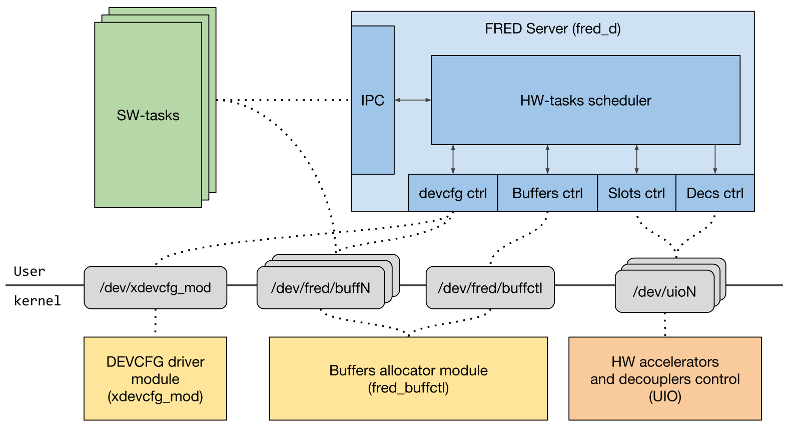


Figure 5.23: FRED software support architecture.

custom kernel modules, and the UIO framework, in order to perform the low-level operations required to control the hardware platform.

Kernel space The two aforementioned kernel modules are used to (i) allocate the memory buffers employed to share data between SW and HW-tasks, and (ii) manage the device reconfiguration. The UIO framework is used for managing hardware accelerators (control and data registers, and interrupt signals) from userspace.

Memory Allocator Module To enforce memory coherence between SW-tasks and HW-tasks, the *shared-memory* infrastructure, has been implemented using a set of *uncached* memory buffers allocated by a custom kernel module.

The custom kernel module uses the Linux DMA layer to allocate physically contiguous (uncached) memory buffers used to exchange data between HW and SW-tasks. When loaded by the system, the module creates a new character device named `fred_buffctl`, used by the FRED server during the initialization phase to request the allocation of memory buffers.

Each allocation request is performed by an `ioctl` operation and includes, as an argument, the size of the required buffer. On the kernel side, when the driver receives an allocation request, it creates a new character device named `fred_buffN` (where `N` refers to the buffer identifier that is assigned by the module) and allocates a new contiguous memory buffer, associated with the device, using the `dma_alloc_coherent()` function of the Linux DMA layer. The character device is the means by which the buffer is accessible from userspace.

Once the buffer device has been created, it can be accessed by a SW-task using the Linux standard `mmap()` syscall. When a SW-task calls (from userspace) the `mmap()` on a buffer character device, the corresponding buffer is mapped into its virtual address space. Inside the driver (on the kernel side) the mapping is performed using the `dma_common_mmap()` function of the Linux DMA layer.

Once the buffer is mapped into the SW-task's virtual space, it can be accessed by the task to read and write data without any system overhead. Since the buffer is uncached, no flush and invalidate operations are required on the cache (note that there are no common cache levels to both the processor and the hardware accelerators, which are directly connected to the DRAM controller). On the other side, a HW-task can access the same buffer through a physical memory address. Such an address is written into the control registers of the HW-task by the FRED server (discussed in details in Section 5.3.4).

In this way, data can be transferred between HW and SW tasks *without any copy operation* or operating system overhead. It is worth observing that under this design the SW-tasks never deal with memory management operations. Each SW-task sees a buffer only as a character device that can be mapped, during its initialization phase, into its virtual memory space. The process of requesting the mapping of such buffers is assisted by a *client support library*.

When the FRED server is shutdown the buffer devices created during the initialization phase are released calling an `ioctl` operation on the `fred_buffctl` device.

Reconfiguration Driver The Zynq FPGA fabric can be reconfigured by the DevC subsystem, as described in Section 5.3.4. Under Linux, the DevC is controlled by a kernel driver module designed by Xilinx. Such a driver allocates a character device named `xdevcfg` that can be used to reconfigure the FPGA fabric from the userspace, taking a bitstream (introduced in Section 5.3.4) as input.

The reconfiguration process is initiated by a `write()` operation on the `xdevcfg` device allocated by the driver. The argument of the write operation is the bitstream file containing the hardware configuration.

The Xilinx's driver has been likely designed with simplicity as a primary design principle. Internally, for each request, the driver allocates a contiguous uncached memory buffer using the `dma_alloc_coherent()` function of the Linux DMA layer. Once the buffer has been allocated and mapped, the driver copies the entire bitstream from the userspace to the buffer, using the `copy_from_user()` function of the Linux kernel. Once the bitstream has been copied into the buffer, the driver starts the DevC internal DMA engine for transferring the bitstream from the system memory to the FPGA configuration memory. After the DMA has been started, the driver performs a busy-wait, polling on a DMA status flag until the transfer has been completed.

This mode of operation is intended to minimize the user efforts to use the driver, but it is clearly unsuitable for the FRED framework because the copy overheads and the busy waits are not compatible with the intensive usage of partial reconfiguration required by the FRED framework. To overcome these issues, the original driver has been modified to take advantage of the allocator module described in the previous section. The rationale is to pre-load all the HW-tasks' bistreams into a set of contiguous memory buffers allocated using the allocator module. Since those operations are performed only once, during the FRED server initialization, they do not produce any overhead at run time.

Once the bitstreams are loaded in physically contiguous memory buffers, they can be reached by the DevC internal DMA engine. For this reason, the driver has been modified to include an `ioctl()` method that allows to start the reconfiguration by passing to the driver a memory reference to a pre-allocated bitstream.

To avoid the busy-wait, the driver has been enhanced with the Linux standard `poll()` method. Once the reconfiguration has been completed, such a method sets the file descriptor of the `xdevcfg` device ready for a read operation. In this way, the reconfiguration process can be easily monitored through POSIX standard I/O multiplexing methods such as `select()` and `poll()`, or the Linux-specific `epoll()`.

With the approach described above, the reconfiguration process is started by an `ioctl()` call on the `xdevcfg` device. The call returns immediately and a userspace application can wait for the end of the reconfiguration without busy-waiting.

Userspace The FRED server is the main userspace component of the FRED software support. Form an architectural perspective it is organized as an event-driven system. Internally, the server includes a core component, named "HW-Tasks scheduler", supported by a layer of software libraries used to perform low-level operations, as shown in Figure 5.23. Conceptually, the FRED server interacts with the rest of the system by means of two main software interfaces, one dedicated to interprocess communications with SW-tasks and the other used to interact with the low-level support.

During the initialization phase the FRED server reads two configuration files containing the description of the hardware design. The first file specifies the layout of the FPGA in terms of partitions and slots. The second file defines the available HW-tasks. According to such files, the FRED server initializes its own data structures and requests the allocator module to allocate the memory buffers used for both bitstreams and data sharing.

Communication Mechanism The communications channels between the FRED server and SW-tasks rely upon *Unix domain sockets*. After the initialization phase, the server instantiates a listening socket, named `fred_sock`, used by SW-tasks to establish a new connection. Once the connection is established, the SW-task can send requests to the server.

To make the system more usable from a client programmer perspective, communication functions between SW-tasks and the FRED server are encapsulated into the client support library. The communication pattern between FRED server and a SW-task is presented in Figure 5.24. Once the server setup is completed, a SW-task can initiate a new connection by calling the `fred_init_hwt()` function (see Figure 5.24). The FRED server replies back with a message containing the buffers description in terms of device files and sizes. Using this data the SW-task can map the buffers into its own address space. Again, such a mapping operation is assisted by the client support library.

At this point, the SW-task can fill the input buffers of its associated HW-task by simply writing into the corresponding memory locations without any additional overhead. Once input data have been prepared, the SW-Task can request the execution of its HW-task by calling the `fred_exec_hwt()` function. This function call causes the SW-task to be suspended until the completion of the HW-task. When the HW-Task completes, the SW-Task resumes its execution and can retrieve the data from the output buffers.

It is worth noticing that SW-tasks never interact directly with the hardware, nor they are required to perform privileged operations. Any interaction between client SW-tasks and the platform hardware are mediated by the FRED server.

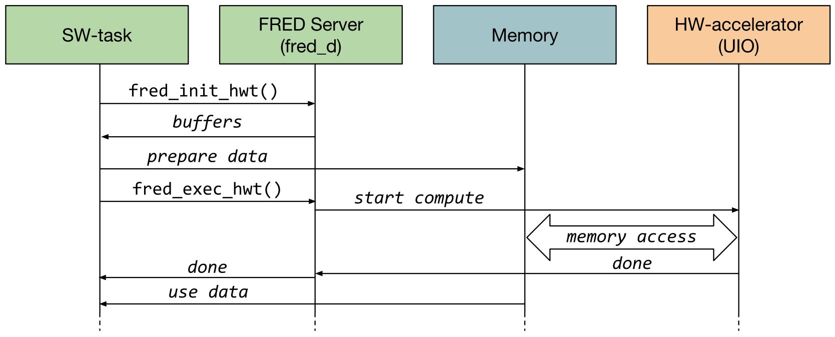


Figure 5.24: Communication between SW-Tasks, FRED server, and HW.

Event Loop The main component of the HW-tasks scheduler is a state machine driven by an event loop. The event loop monitors the file descriptors exported by the low-level and communication interfaces and drives the state machine to handle incoming client requests and hardware events. Internally, the event loop is built around the `epoll` system call. The classes of events handled by the event loop are: (i) Completion of the device configuration; (ii) Completion of a HW-Task; (iii) Connection request from a SW-Task; (iv) Message from a SW-Task.

Experimental Results

This section describes a set of experiments aimed at evaluating the performance of the FRED software support. The experiments have been carried out on the Digilent's Zybo board featuring the Zynq-7010 SoC and running Xilinx's PetaLinux.

Performance Evaluation of the Reconfiguration Driver The first experiment evaluated the improvements achieved using the customized device reconfiguration driver with respect to the Xilinx driver.

Measurements were done by running a single dummy task triggering 10^6 reconfiguration requests to the driver, each of them configuring a 338 KB bitstream into an FPGA slot. During the experiment the system was not loaded, hence the DevC DMA did not suffer from any interference on the system bus while reading the bitstream from memory.

Figure 5.25 shows the reconfiguration times measured when using the original Xilinx driver, and those obtained with the custom driver developed in this work. While the average reconfiguration time using the Xilinx driver is 4.340 ms, the custom one reduces the average reconfiguration time to 2.755 ms, with an approximate speedup of 1.574. Moreover, the worst-case reconfiguration time measured for the original driver is 6.876 ms, improved with a speedup of 2.340 by the custom driver, for which the longest measured reconfiguration time was 2.940 ms.

These results shows that our approach improves the reconfiguration time while decreasing the variance from $4.48 \cdot 10^{-3}$ (for the Xilinx driver) to $1.62 \cdot 10^{-5}$ (for the custom driver).

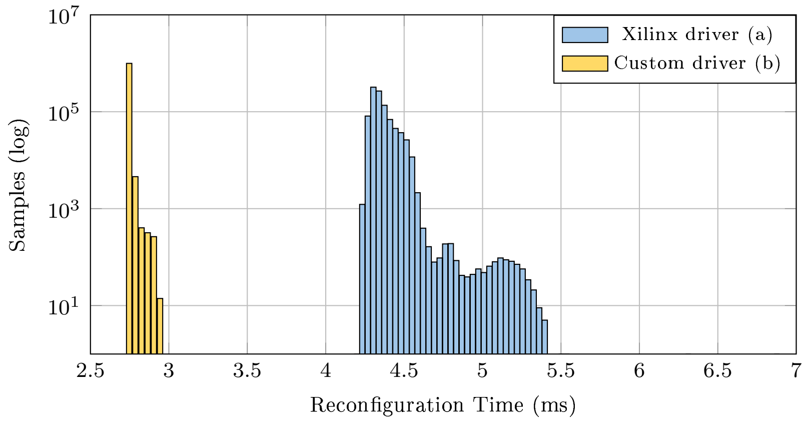


Figure 5.25: Distributions of the reconfiguration times.

Overhead of the Linux Support The second experiment was aimed at measuring the overhead introduced by the FRED software support while serving the requests generated by the SW-tasks.

To evaluate the net overhead introduced by the infrastructure (namely inter-process communication and the management of hardware events), the experiment was performed with a basic system configuration consisting of a single SW-task running in the system, requesting for hardware accelerations to the FRED server. More specifically, the experiment measured the overheads introduced by the FRED software support when calling `fred_exec_hwt()`, that includes:

- The time elapsed from the acceleration request to the instant at which the FRED server triggers the driver to perform the hardware reconfiguration;
- The interval from the time at which the driver notifies the FRED server with the end of reconfiguration to the time at which the FRED server starts the HW-task;
- The interval between the time at which the HW-Task notifies the FRED server its completion and the time at which the SW-Task is awakened.

Figure 5.26 reports the distribution of the sums of the aforementioned latencies measured for each acceleration request performed by the SW-Task. The longest measured overhead resulted 227.125 μs, while the average delay was 77.978 μs. Please note that the overhead introduced by the FRED software support does not depend upon the amount of data shared between SW and HW.

Conclusions

This chapter presented the design and implementation of the FRED framework on the Linux operating system to ease the exploitation of FPGA accelerators in real-time applications running on the Zynq-7000 platform. The software includes a kernel module for implementing shared-memory communication with

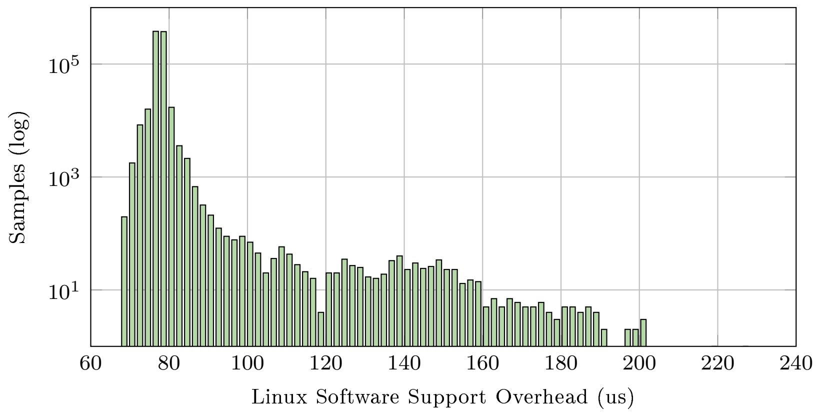


Figure 5.26: Distribution of the overhead introduced by FRED.

hardware accelerators, an improved driver to handle the FPGA reconfigurations, and a userspace daemon to schedule the requests of hardware acceleration.

Experimental results showed that the proposed approach allows halving the reconfiguration times with respect to the official driver released by Xilinx, with a speedup of 2.340. It has also been shown that the features offered by the FRED software support introduce an overhead that can be tolerated by several applications, with a maximum measured overhead less than 228 μs.

5.3.5 Conclusions and Open Challenges

This chapter presented a research activity aimed at improving the predictability of dynamically reconfigured FPGA accelerators requested on-line by real-time tasks. This work has been achieved in different steps.

The first step has been the experimental study aimed at evaluating the feasibility in the use of FPGA dynamic partial reconfiguration features for implementing a timesharing mechanism to virtualize the FPGA resource in heterogeneous platforms that also include a processor. A case study application has been used to profile the temporal parameters involved in the system (i.e., reconfiguration and execution times), and demonstrated that, in spite of the relatively high reconfiguration times of FPGAs, a timesharing mechanism can significantly improve the performance of real-time applications with respect to a fully static approach.

The section also presents the models of the platform and the computational activities that have been developed during this research activity, and proposes a scheduling infrastructure to bound the delays experienced by the tasks, as well as a response-time analysis to verify the schedulability of safety-critical applications with real-time constraints.

The mentioned approaches has also been implemented and validated on a real Zybo platform, embedding a Zynq-7000 platform, to demonstrate its practical applicability. A first implementation relied on the FreeRTOS operating system, while a second complete infrastructure has been obtained by extending the APIs and kernel of Linux for the real-time management of the dynamic reconfiguration of the FPGA, and the communication between the software and hardware tasks.

This work demonstrated the advantages in terms of response time of adopting an FPGA-based dynamic reconfigurable architecture, despite of the hardware reconfiguration times, that still represent a technology bottleneck to fully exploit this hardware feature.

The results achieved in this research activity highlighted some interesting open challenges. A first example is the evaluation of different partitioning approaches for the FPGA area to limit contention on the reconfiguration interface. Another open challenge that has been identified in the dynamic partial reconfiguration of FPGA includes the lack of proper analysis of the delays incurred by the hardware accelerators when accessing the AXI bus and the memory controller. As a future work, it would also be possible to incorporate the proposed framework within a hard real-time operating system (e.g., Erika Enterprise), by developing specific system call that would simplify the development of safety-critical real-time applications on heterogeneous computing platforms using FPGA accelerated components.

Chapter 6

Conclusions

This thesis provided several novel contribution to a number of open problems affecting the available computing architectures, with particular emphasis on the computing activities predictability and the energy saving, spreading from the automotive systems to general computing architectures, and finally to different kinds of heterogeneous architectures.

In the automotive field, the typical requirement for the computing architecture is being able to provide real-time guarantees for the tasks involved in the correct control and management of all the sensing and actuating units. Another important aspect is to provide proper tools for the validation of the system requirements, that are widely expressed as formal languages. This thesis addressed the problem of providing a formal definition of different kinds of *latency*, as well as providing real-time analyses, together with the development of a simulator for a realistic model of an AUTOSAR multicore engine control unit. For that given model, another provided research result is the development of two different methodologies to optimize the data allocation among the system memories to minimize the response times of the tasks, the first using a mixed integer linear programming formulation, the second using genetic algorithms. Finally, the system validation problem has been solved with the development of a system that bridges the requirements definition with the system testing, by translating the requirements expressed through formal languages to modules that can be integrated in the system model and simulated to detect errors.

The thesis also focused on different aspects of reservation-based scheduling, from the modeling of a hierarchical real-time scheduler within the Linux kernel, that has also been widely tested to verify the compatibility of its behavior with the real-time theory. The advantages of using this technique to provide predictable quality of service guarantees is presented for different applications, spanning from the audio synthesis, to virtual machines management in cloud environments. The used algorithm has been discovered to provide a misleading behavior when managing tasks that suspend their execution (e.g., sleeping or waiting for a software or hardware resource), loosing the reservation guarantees. This problem has also been studied and solved in the thesis with the development of a novel scheduling algorithm that solves the issue and its implementation on the Linux kernel to demonstrate its low performance overheads.

Finally, the thesis analyzed different kinds of heterogeneous architectures. In the specific case of architectures with different CPUs sharing the same instruction set, it has been proposed a method to minimize the latency of professional Android audio applications, with the minimal impact on the energy consumption. Despite the popularity of these architectures, the operating systems are not yet able to exploit at the same time both their computational and energy saving potentials, so this thesis presents a power consumption and a computing time models, integrated in a simulator with the aim of supporting the energy-aware scheduling research. Another novel contribution of this thesis is the creation of a novel framework for supporting the real-time execution of task sets that are able to boost their computations by demanding their work to hardware accelerators that can be dynamically programmed in the FPGA area.

All the mentioned works have been carried out with a careful research activity aimed at finding novel and correct solutions to the identified problems, that have been formally and experimentally validated, taking also care of the practicability of the proposals, for which in all the cases have been provided tools or modifications of widespread systems, making the presented contributions potentially available to everyone.

Bibliography

[1] S. H. A. Fernandez-Leon A. Pouponnot, “Esa fpga task force: Lessons learned”, in *Military and Aerospace Programmable Logic Device (MAPLD)*, Sep. 2002.

[2] Y. Abdeddaïm and D. Masson, “The scheduling problem of self-suspending periodic real-time tasks”, in *Proceedings of 20th International Conference on Real-Time and Network Systems*, Pont-à-Mousson, France, 2012.

[3] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, “Qos management through adaptive reservations”, *Real-Time Systems*, vol. 29, no. 2, pp. 131–155, 2005, ISSN: 1573-1383.

[4] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, “Analysis of a reservation-based feedback scheduler”, in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, 2002, pp. 71–80.

[5] L. Abeni, A. Balsini, and T. Cucinotta, “Container-based real-time scheduling in the linux kernel”, in *ACM SIGBED Review - Special Issue on Embedded Operating Systems Workshop (EWiLi ’18)*, 2018.

[6] L. Abeni and G. Buttazzo, “Integrating multimedia applications in hard real-time systems”, in *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[7] L. Abeni, G. Lipari, A. Parri, and Y. Sun, “Multicore cpu reclaiming: Parallel or sequential?”, in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC ’16, Pisa, Italy: ACM, 2016, pp. 1877–1884, ISBN: 978-1-4503-3739-7.

[8] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari, “Resource reservations for general purpose applications”, *IEEE Transactions on Industrial Informatics*, vol. 5, no. 1, pp. 12–21, 2009.

[9] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, “Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling”, in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 74–85.

[10] A. Ali-Eldin, J. Tordsson, and E. Elmroth, “An adaptive hybrid elasticity controller for cloud infrastructures”, in *2012 IEEE Network Operations and Management Symposium*, 2012, pp. 204–212.

[11] M. Alicherry and T. V. Lakshman, “Network aware resource allocation in distributed clouds”, in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 963–971.

[12] S. Altmeyer and G. Gebhard, "WCET analysis for preemptive scheduling", in *Proc. of the 8th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, Prague, Czech Republic, 2008, pp. 105–112.

[13] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for schedulability analysis and code generation of real-time systems", in *Formal Modeling and Analysis of Timed Systems*, K. G. Larsen and P. Niebert, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 60–72, ISBN: 978-3-540-40903-8.

[14] E. Asarin, P. Caspi, and O. Maler, "Timed regular expressions", *J. ACM*, vol. 49, no. 2, pp. 172–206, Mar. 2002, ISSN: 0004-5411.

[15] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, "Power-aware scheduling for periodic real-time tasks", *IEEE Transactions on Computers*, vol. 53, no. 4, pp. 584–600, 2004.

[16] A. Balsini, M. Di Natale, M. Celia, and V. Tsachouridis, "Generation of simulink monitors for control applications from formal requirements", in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2017, pp. 1–9.

[17] A. Balsini. (). Signal template library autogenerator tool, [Online]. Available: `https://github.com/balsini/SignalTemplateLibraryAutogen/`.

[18] A. Balsini, A. Melani, P. Buonocunto, and M. Di Natale, "Fmtv 2016: Where is the actual challenge?", in *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'16), in conjuction with the 289th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, Jul. 2016.

[19] A. Balsini, L. Pannocchi, and T. Cucinotta, "Modeling and simulation of power consumption for real-time embedded heterogeneous architectures", in *ACM SIGBED Review - Special Issue on Embedded Operating Systems Workshop (EWiLi '18)*, 2018.

[20] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems", in *OSDI*, vol. 99, 1999, pp. 45–58.

[21] C. Bartolini and G. Lipari. (2011). The rtsim scheduling simulator, [Online]. Available: `http://rtsim.sssup.it/`.

[22] S. Baruah, "Resource sharing in EDF-scheduled systems: A closer look", in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, December 5-8, 2006.

[23] R. Basmadjian and H. de Meer, "Evaluating and modeling power consumption of multi-core processors", in *2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy)*, 2012, pp. 1–10.

[24] M. Becker, "Consolidating automotive real-time applications on many-core platforms", PhD thesis, Mälardalen University, Embedded Systems, 2017, ISBN: 978-91-7485-359-9.

[25] C. Beckhoff, D. Koch, and J. Torresen, "Go ahead: A partial reconfiguration framework", in *Proc. of the 20th Annual IEEE Int. Symposium on Field-Programmable Custom Computing Machines*, Toronto, Canada, 2012.

[26] M. Behnam, "Synchronization protocols for a compositional real-time scheduling framework", PhD thesis, M
"alardalen University, 2010. [Online]. Available: `http://www.es.mdh.se/publications/1968-`.

[27] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments", *IEEE Transactions on Industrial Informatics*, vol. 5, no. 3, pp. 202–219, 2009.

[28] E. Bini, M. Bertogna, and S. Baruah, "Virtual multiprocessor platforms: Specification and use", in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 437–446.

[29] E. Bini, G. Buttazzo, and M. Bertogna, "The multi supply function abstraction for multiprocessors", in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 294–302.

[30] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests", *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005, ISSN: 0922-6443.

[31] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, *et al.*, "A framework for supporting real-time applications on dynamic reconfigurable fpgas", in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 1–12.

[32] A. Biondi and B. Brandenburg, "Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors", in *ECRTS'16*.

[33] A. Biondi, M. Di Natale, Y. Sun, and S. Botta, "Moving from single-core to multicore: Initial findings on a fuel injection case study", in *SAE Technical Paper, SAE Conference, Detroit, USA*, 2016.

[34] A. Biondi, A. Balsini, and M. Marinoni, "Resource reservation for real-time self-suspending tasks: Theory and practice", in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ser. RTNS '15, Lille, France: ACM, 2015, pp. 97–106, ISBN: 978-1-4503-3591-1.

[35] A. Biondi, M. Di Natale, and G. Buttazzo, "Response-time analysis for real-time tasks in engine control applications", in *Proceedings of the 6th International Conference on Cyber-Physical Systems (ICCPS 2015)*, Seattle, Washington, USA, 2015.

[36] A. Biondi, A. Melani, and M. Bertogna, "Hard constant bandwidth server: Comprehensive formulation and critical scenarios", in *Proc. of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, Pisa, Italy, 18-20 June, 2014.

[37] A. Biondi, A. Melani, M. Bertogna, and G. Buttazzo, "Optimal design for reservation servers under shared resources", in *Proc. of the 26th Euromicro Conference on Real-Time Systems (ECRTS 14)*, Madrid, Spain, 9-11 July, 2014.

[38] A. Biondi, P. Pazzaglia, A. Balsini, and M. Di Natale, “Logical execution time implementation and memory optimization issues in autosar applications for multicores”, in *Proceedings of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'17), in conjuction with the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[39] M. Bohn, J. Schneider, and C. Eltges, “Simtros: A heterogenous abstraction level simulator for multicore synchronization in real-time systems”, *Journal of Systems Architecture*, vol. 59, no. 6, pp. 297 –306, 2013, ISSN: 1383-7621.

[40] B. B. Brandenburg and J. H. Anderson, “The omlp family of optimal multiprocessor real-time locking protocols”, *Design Automation for Embedded Systems*, pp. 1–66, 2012.

[41] R. Bril, “Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption is too optimistic”, *CS-Report 06*, vol. 5, 2006.

[42] R. Bril and W. Verhaegh, “Towards best-case response times of real-time tasks under fixed-priority scheduling with deferred preemption”, in *ECRTS, WiP session*, 2005, pp. 17–20.

[43] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations”, in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 2000, pp. 83–94.

[44] A. Burmyakov, E. Bini, and E. Tovar, “Compositional multiprocessor scheduling: The gmpr interface”, *Real-Time Systems*, vol. 50, no. 3, pp. 342–376, 2014, ISSN: 1573-1383.

[45] G. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems. A survey.”, *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[46] J. A. Butts and G. S. Sohi, “A static power model for architects”, in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, 2000, pp. 191–201.

[47] D. Calvaresi, P. Sernani, M. Marinoni, A. Claudi, A. Balsini, *et al.*, “A framework based on real-time os and multi-agents for intelligent autonomous robot competitions”, in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016, pp. 1–10.

[48] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, “YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms”, in *WATERS 2012*, Italy, Jul. 2012, pp. 21–26.

[49] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, “Hierarchical Multiprocessor CPU Reservations for the Linux Kernel”, in *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, Dublin, Ireland, 2009.

[50] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport, "An FPGA implementation of the two-dimensional finite-difference time-domain (fdtd) algorithm", in *Proceedings of the ACM/SIGDA 12th international symposium on Field programmable gate arrays*, Monterey, California, USA, 2004.

[51] W. H. Cheng and B. M. Baas, "Dynamic voltage and frequency scaling circuits with two supply voltages", in *2008 IEEE International Symposium on Circuits and Systems*, 2008, pp. 1236–1239.

[52] A. Colin, A. Kandhalu, and R. Rajkumar, "Energy-efficient allocation of real-time applications onto heterogeneous processors", in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014, pp. 1–10.

[53] P. Courbin and L. George, "Fortas: Framework for real-time analysis and simulation", in *Proc. of WATERS 2011*, Porto, Portugal, Jul. 2011, pp. 21–26.

[54] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services", in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 2, 2009.

[55] T. Cucinotta, F. Checconi, Z. Zlatev, J. Papay, M. Boniface, *et al.*, "Virtualised e-Learning with real-time guarantees on the IRMOS platform", in *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2010, pp. 1–8.

[56] T. Cucinotta, D. Lugones, D. Cherubini, and E. Jul, "Data Centre Optimisation Enhanced by Software Defined Networking", in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 136–143.

[57] T. Cucinotta, L. Palopoli, and L. Marzario, "Stochastic feedback-based control of qos in soft real-time systems", in *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, vol. 4, 2004, 3533–3538 Vol.4.

[58] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni, "Adaptive reservations in a linux environment", in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, 2004, pp. 238–245.

[59] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Virtual network functions as real-time containers in private clouds", in *11th IEEE International Conference on Cloud Computing (IEEE CLOUD 2018)*, Jul. 2018.

[60] T. Cucinotta, L. Abeni, M. Marinoni, and C. Vitucci, "The Importance of Being OS-aware - In Performance Aspects of Cloud Computing Research", in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018.

[61] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Adaptive real-time scheduling for legacy multimedia applications", *ACM Trans. Embed. Comput. Syst. – Special Section on Embedded Systems for Real-Time Multimedia*, vol. 11, no. 4, 86:1–86:23, Jan. 2013, ISSN: 1539-9087.

[62] T. Cucinotta, D. Faggioli, and G. Bagnoli, "Low-latency audio on linux by means of real-time scheduling", in *Proceedings of the Linux Audio Conference (LAC 2011)*, Maynooth, Ireland, May 2011.

[63] T. Cucinotta, M. Marinoni, A. Melani, A. Parri, and C. Vitucci, "Temporal Isolation Among LTE/5G Network Functions by Real-time Scheduling", in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, Funchal, Madeira, Portugal, 2017, pp. 368–375, ISBN: 978-989-758-243-1.

[64] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari, "On the integration of application level and resource level qos control for real-time applications", *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, Nov. 2010.

[65] K. Danne and M. Platzner, "Periodic real-time scheduling for FPGA computers", in *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded System*, Hamburg, Germany, 2005.

[66] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised", *Real-Time System*, vol. 35, no. 3, pp. 239–272, 2007.

[67] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "The energy/frequency convexity rule: Modeling and experimental validation on mobile devices", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 793–803, ISBN: 978-3-642-55224-3.

[68] K. Deemter, V. E. Krahmer, and M. Theune, "Real versus template-based natural language generation: A false opposition?", in *Computer Linguist*, vol. 31(1), 2005, pp. 15–24.

[69] U. M. C. Devi and J. H. Anderson, "Tardiness bounds under global edf scheduling on a multiprocessor", in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005, 12 pp.–341.

[70] A. Dhodapkar, C. How Lim, G. Cai, and W. Robert Daasch, "Tem2p2est: A thermal enabled multi-model power/performance estimator", in *Power-Aware Computer Systems*, B. Falsafi and T. N. Vijaykumar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 112–125, ISBN: 978-3-540-44572-2.

[71] M. Di Natale and E. Bini, "Optimizing the FPGA implementation of hrt systems", in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, 2007.

[72] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools", in *Proceedings of the IEEE*, vol. 98 (4), 2010, pp. 603–620.

[73] Digilent, *Zybo reference manual*, 2016. [Online]. Available: `https://reference.digilentinc.com/_media/zybo/zybo_rm.pdf`.

[74] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling", in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Nice, France, 2007.

[75] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for stl", in *Proceedings of the 25th International Conference on Computer Aided Verification*, ser. CAV'13, Saint Petersburg, Russia: Springer-Verlag, 2013, pp. 264–279, ISBN: 978-3-642-39798-1.

[76] P. S. Duggirala, C. Fan, S. Mitra, and M. Viswanathan, "Meeting a powertrain verification challenge", in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., Cham: Springer International Publishing, 2015, pp. 536–543, ISBN: 978-3-319-21690-4.

[77] F. Duhem, F. Muller, and P. Lorenzini, "Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA", in *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, Belfast, UK, 2011.

[78] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling", *Real-Time Systems*, vol. 43, no. 1, pp. 25–59, 2009, ISSN: 1573-1383.

[79] C. Eisner and D. Fisman, *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387353135.

[80] E. C. E. Emerson, "Design and synthesis of synchronisation skeletons using branching time temporal logic", in *Logic of Programs, Proceedings of Workshop, Lecture Notes in Computer Science*, vol. 131, Springer, Berlin, 1981, pp. 52–71.

[81] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol", in *2010 22nd Euromicro Conference on Real-Time Systems*, 2010, pp. 90–99.

[82] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics", in *CRTS*, 2008.

[83] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, and P. Gai, "Time and memory tradeoffs in the implementation of autosar components", in *Design, Automation and Test in Europe Conference. DATE'09*, 2009.

[84] J. Flores, "Semantic filtering of textual requirements descriptions", in *Natural Language Processing and Information Systems*, 2004, pp. 474–483.

[85] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, *et al.*, "OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in Openstack", in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: ACM, 2014, pp. 353–354, ISBN: 978-1-4503-2836-4.

[86] S. Ghosh, R. Raj Rajkumar, J. Hansen, and J. Lehoczky, "Integrated qos-aware resource management and scheduling with multi-resource constraints", *Real-Time Systems*, vol. 33, no. 1, pp. 7–46, 2006, ISSN: 1573-1383.

[87] S. Gnesi, G. Lami, and G. Trentanni, "An automatic tool for the analysis of natural language requirements", in *CSSE Journal*, vol. 20(1), 2005, pp. 53–62.

[88] M. Goosman, N. Dorairaj, and E. Shiflet, *How to take advantage of partial reconfiguration in FPGA designs*, 2006. [Online]. Available: `www.eetimes.com/document.asp?doc\_id=1274489`.

[89] I. Gray, Y. Chan, J. Garside, N. Audsley, and A. Wellings, "Transparent hardware synthesis of java for predictable large-scale distributed systems", in *Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP)*, London, UK, 2015.

[90] *GSL gnu scientific library*, `https://www.gnu.org/software/gsl/`, Accessed: Version 1.16.

[91] S. Habinc, "Technical report: Suitability of reprogrammable FPGAs in space applications", Gaisler Research, Tech. Rep., 2002.

[92] M. Happe, A. Traber, and A. Keller, "Preemptive hardware multitasking in reconos", in *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC)*, Bochum, Germany, 2015, pp. 79–90.

[93] M. G. Harbour, J. J. Gutiérrez, J. M. Drake, P. L. Martínez, and J. C. Palencia, "Modeling distributed real-time systems with mast 2", *Journal of Systems Architecture*, vol. 59, no. 6, pp. 331 –340, 2013, ISSN: 1383-7621.

[94] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree, "From control models to real-time code using giotto", in *Control Systems Magazine, IEEE*, 2003.

[95] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming.", in *Proc. International Workshop on Embedded Software (EMSOFT), volume 2211 of LNCS*, Springer, Ed., 2001, pp. 166–184.

[96] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "Poet: A portable approach to minimizing energy under soft real-time constraints", in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 75–86.

[97] C. Imes and H. Hoffmann, "Minimizing energy under performance constraints on embedded platforms: Resource allocation heuristics for homogeneous and single-isa heterogeneous multi-cores", *SIGBED Rev.*, vol. 11, no. 4, pp. 49–54, Jan. 2015, ISSN: 1551-3688.

[98] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, *et al.*, "Microkernel architecture and hardware abstraction layer of a reliable reconfigurable real-time operating system (R3TOS)", *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, 5:1–5:35, 2015.

[99] I. Kalkov, D. Franke, J. F. Schommer, and S. Kowalewski, "A real-time extension to the android platform", in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12, Copenhagen, Denmark: ACM, 2012, pp. 105–114, ISBN: 978-1-4503-1688-0.

[100] I. Kalkov, A. Gurghian, and S. Kowalewski, “Priority inheritance during remote procedure calls in real-time android using extended binder framework”, in *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES ’15, Paris, France: ACM, 2015, 5:1–5:10, ISBN: 978-1-4503-3644-4.

[101] A. Kantee, “The rise and fall of the operating system”, *USENIX login*, vol. 40, no. 5, 2015.

[102] J. Kapinski, X. Jin, J. Deshmukh, A. Donze, T. Yamaguchi, *et al.*, “St-lib: A library for specifying and classifying model behaviors”, in *SAE Technical Paper*, SAE International, Apr. 2016.

[103] N. Khalilzad, F. Kong, X. Liu, M. Behnam, and T. Nolte, “A feedback scheduling framework for component-based soft real-time systems”, in *21th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

[104] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, *et al.*, “Leakage current: Moore’s law meets static power”, *Computer*, vol. 36, no. 12, pp. 68–75, 2003, ISSN: 0018-9162.

[105] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer Publishing Company, Incorporated, 2012, ISBN: 1461412242, 9781461412243.

[106] K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou, “Admission Control for Elastic Cloud Services”, in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 41–48.

[107] K. Konstanteli, T. Cucinotta, K. Psychas, and T. A. Varvarigou, “Elastic admission control for federated cloud services”, *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 348–361, 2014, ISSN: 2168-7161.

[108] H. Kopetz and S. Poledna, “In-vehicle real-time fog computing”, in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, pp. 162–167.

[109] R. Koymans, “Specifying real-time properties with metric temporal logic”, in *Real-Time Systems*, vol. 2(4), 1990, 255–299.

[110] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, *et al.*, “Realizing compositional scheduling through virtualization”, in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, 2012, pp. 13–22.

[111] J. Lelli, D. Faggioli, T. Cucinotta, and S. Superiore, “An efficient and scalable implementation of global edf in linux”, in *Proc. of the 7th Workshop on Operating Systems Platforms for Embedded Real-Time applications*, Porto, Portugal, 2011.

[112] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, “Deadline scheduling in the Linux kernel”, *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016, spe.2335, ISSN: 1097-024X.

[113] J. Levin, *Android Internals - Volume I: A Confectioner’s Cookbook*. Jonathan Levin, 2014, ISBN: 9780991055524. [Online]. Available: `https://books.google.it/books?id=onhDnwEACAAJ`.

[114] J. Levine, *Flex & Bison*, 1st. O’Reilly Media, Inc., 2009, ISBN: 0596155972, 9780596155971.

[115] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees", in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[116] W. Liao, L. He, and K. M. Lepak, "Temperature and supply voltage aware performance and power modeling at microarchitecture level", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1042–1053, 2005, ISSN: 0278-0070.

[117] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers", in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, 2000, pp. 193–200.

[118] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation", in *Proc. of 31st IEEE Real-Time Systems Symposium*, Dec. 2010, pp. 249–258.

[119] ——, "A methodology for designing hierarchical scheduling systems", *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, 2005.

[120] C. Liu and J. H. Anderson, "An O(m) analysis technique for supporting real-time self-suspending task systems", in *In Proc. of the 33rd Real-Time Systems Symposium 2012*.

[121] ——, "Suspension-aware analysis for hard real-time multiprocessor scheduling", in *In Proc. of the 25th EuroMicro Conference on Real-Time Systems (ECRTS 2013)*.

[122] ——, "Task scheduling with self-suspensions in soft real-time multiprocessor systems", in *In Proc. of the 30th Real-Time Systems Symposium (RTSS 2009)*.

[123] J. Liu, Ed., *Real-time systems*. Prentice Hall, 2000.

[124] S. Liu, R. N. Pittman, and A. Forin, "Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent ICAP controller", in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, California, USA, 2010.

[125] R. T. E. Ltd., *Freertos real-time operating system*. [Online]. Available: `http://www.freertos.org/`.

[126] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms*", *Real-Time Systems*, vol. 23, no. 1, pp. 85–126, 2002, ISSN: 1573-1383.

[127] E. Lübbers and M. Platzner, "Cooperative multithreading in dynamically reconfigurable systems.", in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Prague, Czech Republic, 2009.

[128] ——, "Reconos: Multithreaded programming for reconfigurable computers", *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, 8:1–8:33, 2009.

[129] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, *et al.*, "Unikernels: Library operating systems for the cloud", *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 461–472, Mar. 2013, ISSN: 0163-5964.

[130] O. Maler and D. Nickovic., "Monitoring temporal properties of continuous signals", in *Proc. of Formal Modeling and Analysis of Timed Systems/ Formal Techniques in Real-Time and Fault Tolerant Systems*, 2004, pp. 152–166.

[131] O. Maler, D. Nickovic, and A. Pnueli, "Checking temporal properties of discrete, timed and continuous behaviors", in *Pillars of Computer Science: Lecture Notes in Computer Science*, vol. 4800, Springer, 2003, pp. 475–505.

[132] R. Mall, Ed., *Real-time systems: theory and practice.* Pearson Education, 2008.

[133] L. Mangeruca and O. A. F. Ferrante, "Formalization and completeness of evolving requirements using contracts", in *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, 2013.

[134] M. Marinoni and G. Buttazzo, "Elastic dvs management in processors with discrete voltage/frequency modes", *IEEE Transactions on Industrial Informatics*, vol. 3, no. 1, pp. 51–62, 2007.

[135] E. Martins, L. Almeida, and J. A. Fonseca, "An FPGA-based coprocessor for real-time fieldbus traffic scheduling: Architecture and implementation", *Journal of Systems Architecture*, vol. 51, no. 1, pp. 29–44, 2005.

[136] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "IRIS: A new reclaiming algorithm for server-based real-time systems", in *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, 2004.

[137] T. Mathworks., "Simulink user manual", in *Product web page*, 2017.

[138] *MetaSim2.0 event-based simulator*, `https://github.com/balsini/metasim2.0`, Accessed: May 17, 2016.

[139] J. Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, *et al.*, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip", in *Proceedings of DATE*, Munich, Germany, 2003.

[140] C. Möbius, W. Dargie, and A. Schill, "Power consumption estimation models for processors, virtual machines, and servers", *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1600–1614, 2014, ISSN: 1045-9219.

[141] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, "Timing analysis of fixed priority self-suspending sporadic tasks", in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, Lund, Sweden, 2015.

[142] NFV Industry Specif. Group, *Network Functions Virtualisation*, Introductory White Paper, 2012.

[143] S. L. Obispo, "Parsing of natural language requirements", in *Thesis presented to the Faculty of California Polytechnic State University*, 2013.

[144] *Os and libraries document collection*, UG643, v2015.3, Xilinx, 2015.

[145] B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe, "Dynamic partial reconfiguration in space applications", in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, 2009, pp. 336–343.

[146] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, "A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration", in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 96–101.

[147] M. Pagani, M. Marinoni, A. Biondi, A. Balsini, and G. Buttazzo, "Towards real-time operating systems for heterogeneous reconfigurable platforms", in *Proc. of the 12th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2016)*, Toulouse, France, July 5, 2016.

[148] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors", in *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, 1997, pp. 206–218.

[149] L. Palopoli and T. Cucinotta, "Qos control for pipelines of tasks using multiple resources", *IEEE Transactions on Computers*, vol. 59, pp. 416–430, Jul. 2009, ISSN: 0018-9340.

[150] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA — adaptive quality of service architecture", *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009, ISSN: 0038-0644.

[151] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, *et al.*, "An object-oriented tool for simulating distributed real-time control systems", *Softw. Pract. Exper.*, vol. 32, no. 9, pp. 907–932, Jul. 2002, ISSN: 0038-0644.

[152] P. R. Panda, B. V. N. Silpa, A. Shrivastava, and K. Gummidipudi, *Power-efficient System Design*, 1st. Springer Publishing Company, Incorporated, 2010, ISBN: 1441963871, 9781441963871.

[153] M. Park and H. Park, "An efficient test method for rate monotonic schedulability", *IEEE Transactions on Computers*, vol. 63, no. 5, 2014.

[154] *Partial reconfiguration user guide*, UG702, v14.1, Xilinx, 2012.

[155] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems", *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[156] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, *et al.*, "Energy-efficient thread assignment optimization for heterogeneous multicore systems", *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, 15:1–15:26, Jan. 2015, ISSN: 1539-9087.

[157] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, *et al.*, "CARTS: A tool for compositional analysis of real-time systems", *SIGBED Review*, vol. 8, no. 1, pp. 62–63, 2011, ISSN: 1551-3688.

[158] A. Pnueli, "The temporal logic of programs", in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977, pp. 46–57.

[159] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures”, *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[160] Prover., 2017. [Online]. Available: `https://www.prover.com`.

[161] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, “A resource allocation model for qos management”, in *Proceedings Real-Time Systems Symposium*, 1997, pp. 298–307.

[162] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource kernels: A resource-centric approach to real-time and multimedia systems”, in *SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, USA, 1998.

[163] P. Richard, “On the complexity of scheduling real-time tasks with self-suspension on one processor”, in *Proc. of the 15th IEEE Int. Euromicro Conferecnce on Real-Time Systems (ECRTS 2003).*

[164] F. Ridouard, P. Richard, and F. Cottet, “Negative results for scheduling independent hard real-time tasks with self-suspensions”, in *Proc. of the 25th IEEE Real-Time Systems Symposium (RTSS 2004).*

[165] ——, “Some results on scheduling tasks with self-suspensions”, in *Journal of Embedded Computing, 2006.*

[166] N. Roy, A. Dubey, and A. Gokhale, “Efficient autoscaling in the cloud using predictive models for workload forecasting”, in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[167] *RTSIM real-time system simulator extended for waters challenge 2016*, `https://github.com/balsini/waters/`, Accessed: Branch 2016.

[168] M. Sadri, C. Weis, N. Wehn, and L. Benini, “Energy and performance exploration of accelerator coherency port using xilinx zynq”, in *Proceedings of the 10th FPGAworld Conference*, Stockholm, Sweden, 2013.

[169] S. Saha, A. Sarkar, and A. Chakrabarti, “Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms”, *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, 2015.

[170] C. Scordino and G. Lipari, “A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks”, *IEEE Transactions on Computers*, vol. 55, no. 12, pp. 1509–1522, 2006, ISSN: 0018-9340.

[171] C. Scordino, L. Abeni, and J. Lelli, “Energy-aware real-time scheduling in the linux kernel”, in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Pau, France: ACM, 2018.

[172] S. D. Scott, A. Samal, and S. Seth, “Hga: A hardware-based genetic algorithm”, in *Proceedings of the ACM Third International Symposium on Field-programmable Gate Arrays*, Monterey, California, USA, 1995.

[173] D. Shin, W. Kim, J. Jeon, J. Kim, and S. L. Min, “Simdvs: An integrated simulation environment for performance evaluation of dynamic voltage scaling algorithms”, in *Power-Aware Computer Systems*, B. Falsafi and T. N. Vijaykumar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 141–156, ISBN: 978-3-540-36612-6.

[174] I. Shin, M. Behnam, T. Nolte, and M. Nolin, “Synthesis of optimal interfaces for hierarchical scheduling with resources”, in *Proc. of the 29th IEEE International Real-Time Systems Symposium (RTSS 2008)*, Barcelona, Spain, 2008, pp. 209–220.

[175] I. Shin and I. Lee, “Compositional real-time scheduling framework”, in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 57–67.

[176] ——, “Periodic resource model for compositional real-time guarantees”, in *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 3-5, 2003, pp. 2–13.

[177] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, *et al.*, “Temperature-aware microarchitecture: Modeling and implementation”, *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 94–125, Mar. 2004, ISSN: 1544-3566.

[178] H. K.-H. So and R. Brodersen, “A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph”, *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, 14:1–14:28, 2008.

[179] *Standard for Information Technology – Portable Operating System Interface (POSIX) – System Interfaces. IEEE 1003.1, 2004 Edition.* The Open Group, 2004.

[180] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, “The case for feedback control real-time scheduling”, in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999, pp. 11–20.

[181] R. Stefan and S. D. Cotofana, “Bitstream compression techniques for virtex 4 FPGAs”, in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Heidelberg, Germany, 2008.

[182] J. K. Strosnider, J. P. Lehoczky, and L. Sha, “The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-time environments”, *IEEE Transactions on Computers*, vol. 4, no. 1, 1995.

[183] V. Struhar, A. Papadopoulos, and M. Behnam, “Fog computing for adaptive human-robot collaboration”, in *International Conference on Embedded Software 2018*, 2018. [Online]. Available: `http://www.es.mdh.se/publications/5235-`.

[184] L. A. Torrey, J. Coleman, and B. P. Miller, “A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler”, *Softw. Pract. Exper.*, vol. 37, no. 4, pp. 347–364, Apr. 2007, ISSN: 0038-0644.

[185] D. Ulus, T. Ferrère, E. Asarin, and O. Maler, “Legay, a., bozga, m. (eds.) timed pattern matching”, in *FORMATS 2014. LNCS, vol. 8711*, vol. 8711, Springer, Heidelberg, 2014, pp. 222–236.

[186] ——, “Online timed pattern matching using derivatives”, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2016: Tools and Algorithms for the Construction and Analysis of Systems*, 2016, pp. 736–751.

[187] P. Valente, "Providing near-optimal fair-queueing guarantees at round-robin amortized cost", in *Proceedings of the 22nd International Conference on Computer Communication and Networks, ICCCN 2013*, Nassau, Bahamas, 2013, pp. 1–7.

[188] P. Valente and M. Andreolini, "Improving application responsiveness with the bfq disk i/o scheduler", in *Proceedings of the 5th Annual International Systems and Storage Conference*, Haifa, Israel, 2012, pp. 1–12.

[189] P. Valente and G. Lipari, "An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors", in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, IEEE, 2005, 10–pp.

[190] *Vivado design suite user guide: Partial reconfiguration*, UG909, v2015.4, Xilinx, 2015.

[191] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli, "Improving the size of communication buffers in synchronous models with time constraints", in *IEEE Transactions on Industrial Informatics*, vol. 5 (3), 2009, pp. 229–240.

[192] A. Wieder and B. B. Brandenburg, "On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks", in *2013 IEEE 34th Real-Time Systems Symposium*, 2013, pp. 45–56.

[193] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, *et al.*, "RT-Open Stack: CPU Resource Management for Real-Time Cloud Computing", in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 179–186.

[194] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen", in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, 2011.

[195] Xilinx, *Axi interconnect, logicore ip product guide*, PG059, 2016.

[196] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, *et al.*, "Rt-droid: A design for real-time android", *IEEE Transactions on Mobile Computing*, vol. 15, no. 10, pp. 2564–2584, 2016, ISSN: 1536-1233.

[197] Y. Yan, K. Dantu, S. Y. Ko, J. Vitek, and L. Ziarek, "Making android run on time", in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 25–36.

[198] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, *et al.*, "Real-time android with rtdroid", in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14, Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 273–286, ISBN: 978-1-4503-2793-0.

[199] K. Yang and J. H. Anderson, "On the dominance of minimum-parallelism multiprocessor supply", in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 215–226.

[200] X. Yi, J. Duan, and C. Wu, "GPUNFV: A GPU-Accelerated NFV System", in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet'17, Hong Kong, China: ACM, 2017, pp. 85–91, ISBN: 978-1-4503-5244-4.

[201] W. Yuan and K. Nahrstedt, “Energy-efficient soft real-time cpu scheduling for mobile multimedia systems”, in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03, Bolton Landing, NY, USA: ACM, 2003, pp. 149–163, ISBN: 1-58113-757-5.

[202] H. Zeng and M. Di Natale, “Mechanisms for guaranteeing data consistency and flow preservation in autosar software on multi-core platforms”, in *6th IEEE International Symposium on Industrial Embedded Systems (SIES), Vasteras, Sweden*, 2011.

[203] X. Zhong and C. Xu, “Energy-aware modeling and scheduling for dynamic voltage scaling with statistical real-time guarantee”, *IEEE Transactions on Computers*, vol. 56, no. 3, pp. 358–372, 2007, ISSN: 0018-9340.

[204] *Zynq-7000 ap soc technical reference manual*, UG585, v1.10, Xilinx, 2015.