ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Ph.D. THESIS

FACULTY OF ENGINEERING

**PH.D. PROGRAM IN ELECTRONICS ENGINEERING,
TELECOMMUNICATIONS AND INFORMATION TECHNOLOGY**

Hardware/Software Design of Dynamic Real-Time Schedulers for Embedded Multiprocessor Systems

AUTHOR

Primiano Tucci

ADVISOR

Prof. Antonio Corradi

CO-ADVISOR

Prof. Eugenio Faldella

PH.D. COORDINATOR

Prof. Alessandro Vanelli Coralli

Ciclo: XXV – A.A.: 2011/12 – SSD: ING-INF/05 – SETTORE CONCORSUALE: 09/H1

[This page intentionally left blank]

Abstract

The new generation of multicore processors opens new perspectives for the design of embedded systems. Multiprocessing, however, poses new challenges to the scheduling of real-time applications, in which the ever-increasing computational demands are constantly flanked by the need of meeting critical time constraints. Many research works have contributed to this field introducing new advanced scheduling algorithms. However, despite many of these works have solidly demonstrated their effectiveness, the actual support for multiprocessor real-time scheduling offered by current operating systems is still very limited.

This dissertation deals with implementative aspects of real-time schedulers in modern embedded multiprocessor systems. The first contribution is represented by an open-source scheduling framework, which is capable of realizing complex multiprocessor scheduling policies, such as G-EDF, on conventional operating systems exploiting only their native scheduler from user-space. A set of experimental evaluations compare the proposed solution to other research projects that pursue the same goals by means of kernel modifications, highlighting comparable scheduling performances.

The principles that underpin the operation of the framework, originally designed for symmetric multiprocessors, have been further extended first to asymmetric ones, which are subjected to major restrictions such as the lack of support for task migrations, and later to re-programmable hardware architectures (FPGAs). In the latter case, this work introduces a scheduling accelerator, which offloads most of the scheduling operations to the hardware and exhibits extremely low scheduling jitter.

The realization of a portable scheduling framework presented many interesting software challenges. One of these has been represented by timekeeping. In this regard, a further contribution is represented by a novel data structure, called addressable binary heap (ABH). Such ABH, which is conceptually a pointer-based implementation of a binary heap, shows very interesting average and worst-case performances when addressing the problem of tick-less timekeeping of high-resolution timers.

[This page intentionally left blank]

"*Life's a trampoline.*
Sometimes it takes someone
trying to bring you down,
in order to jump higher"

Santa Clara, CA
4 Oct 2012

Preface

Many things have changed in my life in these years as a PhD student. In the last year, in particular, I had the unique opportunity of working in two of the coolest tech companies of our times, respectively Google Inc. and NVIDIA Corp., that, for a curious turn of events, I joined with near-perfect timing. In the former, I had the pleasure to join the *Chrome for Android* team while the most advanced mobile browser of the time was firstly launched and soon thereafter became the official browser of the Android OS. In the latter, I had the amazing opportunity to join the software engineering team in the months in which the NVIDIA Tesla K20 debuted as the GPGPU powering the most powerful supercomputer of the world[1], the Cray XK7 Titan.

Things will unavoidably change in the future years and perhaps one day many of us will smile rereading these lines and thinking back to them with the same nostalgic mood that we have today seeing screenshots of NCSA Mosaic or reproductions of the Cray-1 in museums. However, one thing will remain the same, whatever changes the future will bring: the thought that I was there and was actively contributing to them.

Many of these changes could not have happened without the influence of many people who crossed my path in these years. A grateful acknowledgement, therefore, goes to them, for the spiritual and material support provided.

First of all I would like to thank all the many friends that I had the pleasure and the luck to meet during my early university years. I will never forget all the pleasant moments we shared together, the uncountable dinners and the never-ending nights of our carefree student years spent in the old streets of Bologna.

A further special thank goes to my friends and colleagues Andrea, Giuseppe and Mario, with whom I had the pleasure to share this PhD adventure (and the unforgettable flavors of the campus' cafeteria).

I would like to express my earnest gratitude to my PhD advisor Prof. Eugenio Faldella, for his remarkable wisdom, honesty, enthusiasm and

[1] http://www.top500.org/lists/2012/11/

friendliness (and, lastly, passion for brain teaser games), which I had the pleasure to appreciate during the numerous days (and late-evenings) spent working together.

Some of my acknowledgements go also to companies. In first place, IMA S.p.A. and SACMI Imola, who believed and supported my work and with which I had the pleasure to collaborate, in my early years, on very challenging embedded industrial automation projects. In particular, my thanks goes for the rewarding thought that parts of this thesis will continue to live in many of the tea bags produced worldwide.

Another thanks goes to the B&R automation and Altera corporations, for the interests that they have demonstrated in the academy through the time and the resources dedicated to university contests, which I attended with some of the projects herein presented (see next page).

A remarkable acknowledgement finally goes to the amazing people I had the opportunity to work with during my last internships. To them goes my greatest and most sincere admiration. Besides their unquestionable and unequaled talent, their wisdom and their technical skills, I have been gladly impressed by the great character and charm of many of them that with extreme discretion, great passion and hard commitment contribute to the technological evolution of our world. A particular thanks, in this regard, goes to my mentors Hans and Lucien, who helped me getting started into these two amazing companies and had the patience to endure my bazillion questions and review my code.

Finally yet importantly, I would like to thank my family that have supported and motivated me during all these years. To them goes my constant and profound gratitude, especially in these months of oceanic distance.

Altera InnovateItaly 2011

"Home automation over mains with Altera NIOS-II"

First place.

B&R European Industrial Ethernet Award 2010/11

"Powerlink over PowerLine: the next generation of home automation runs in real-time over mains"

Winner in the category 'relevance for the industry'.

Altera InnovateItaly 2010

"Hard real-time meets embedded multicore SoPCs"

First place, ex-aequo.

Table of Contents

1. Introduction

1.1. Scenario and motivations

The technological advances of microelectronics have radically changed, among many others, the scenario of modern embedded real-time systems. Up until recently, the world of real-time systems was tightly and almost exclusively bounded to specialized computational platforms such as PLCs (programmable logic controllers) and application-specific microcontrollers. [Lee2000]. In recent decades, new generations of multicore processors and multiprocessor systems on chip (MPSoCs) have opened up new vistas as regards the huge computational power that can be exploited to face the ever-increasing complexity of modern embedded applications.

Furthermore, the high level of integration of silicon technology opens up many interesting possibilities also for the world of re-programmable hardware platforms. The high availability of hardware resources offered by modern Field Programmable Gate Arrays (FPGAs), in fact, have made these platforms interesting targets for the development of integrated multiprocessor systems-on-programmable-chip (MP-SoPCs). These MP-SoPCs allow hardware/software co-design patterns that can reap the benefits of both rapid prototyping and large possibilities of customization [BOBSBS2008, DJM+2009, JS2006, Sch2007].

Together with the countless number of evident advantages, however, all these new multiprocessor platforms also brought in many issues, posing new challenges to the already complex matter of real-time systems.

Primarily there are new methodological issues. The degree of freedom introduced by the presence of many processors does not translate straightforwardly into a direct ability of '*doing more work*'.

Even without bringing in any elaborate consideration such as parallelizability of software algorithms, but keeping more simplistic assumptions of independent tasks, the sole problem of choosing how to distribute such tasks on the available processors easily degenerates into complex (NP-Hard [LW1982]) problems.

In addition, there also many new software design issues. The wide variety of multiprocessors chips currently available, in fact, translates into a wide heterogeneity of strongly different computational architectures that, besides the mere performances, have different programming models and a strong impact on the overall software design.

In most desktop processors (e.g. Intel x86/64) the multicore architecture is quite transparent to the software, allowing designers to rely on legacy shared-memory patterns. However, many other embedded processors have followed a different path, mainly for area and power-related concerns, employing more decoupled, yet cost effective, architectures at the expense of more complex (and sometimes esoteric) software programming models [ANA2004, JBP2006].

The purpose of this dissertation is to focus on these latter software and hardware design aspects, in particular as regards the implementation of complex real-time multiprocessor scheduling policies on these new multiprocessor platforms.

1.2. Contributions

Many remarkable works have contributed in the scientific literature to methodological aspects related to multiprocessor real-time systems. In particular, for what concerns the contents of this thesis, many of these studies have introduced new advanced scheduling algorithms, which are able to exploit fruitfully the computational power of these systems, still ensuring the respect of deadlines [DB2011]. The most renowned of these is undoubtedly represented by the multiprocessor variants of the earliest deadline first algorithm (EDF) [BB2009].

Despite what many published works have solidly demonstrated in past years regarding the effectiveness of such new scheduling algorithms [Bak2003a, Bak2005b, Bak2005c, BCL2005, DA2008, EDB2010, MBer2005], the actual support that current real-time operating systems (RTOSs) offer to deal with multiprocessor platforms is still limited. Almost the all RTOS schedulers, in fact, support merely static (i.e. numeric) priority-driven policies and in many cases do not even deal with the notion of *periodic* processes [CG2011, SR2004].

In this regard, the first contribution of this work is represented by an open-source scheduling framework, called X-RT [Tuc2012]. It consists of a runtime framework that offers, to real-time application developers, a set of high-level and OS-agnostic scheduling API, providing the concepts of periodic tasks and deadlines. Furthermore, its runtime library is capable of realizing, in a portable way (i.e. without altering the RTOS kernel), complex multiprocessor scheduling policies such as G-EDF. The operating principle of the runtime library stands on a metascheduler approach. Such metascheduler is a special process, which dynamically mangles at runtime the numeric priorities of the other RTOSs processes in order to emulate the behavior of more complex scheduling policies.

The realization of such a portable scheduling framework presented many interesting challenges. One of these has been definitely timekeeping, the software handling of several outstanding timers, using a limited number (typically just one) of hardware timers. Timekeeping, however, is a more general topic that, besides the specific problems addressed in the realization of the X-RT framework, has historically generated great interests in scientific and technical literature, since it involves not only the area of (real-time) operating systems, but also many other fields as discrete event simulation and networking.

In this regard, this work contributes to this latter topic introducing a novel data structure, called addressable binary heap (ABH). Such ABH, that conceptually is a pointer-based implementation of the traditional array-backed binary heap, shows very interesting average and worst-case performances, especially when compared to other data structures typically employed for the timekeeping purpose (such as self-balancing binary trees and array-backed binary heaps), making it an interesting alternative for addressing the general problem of tick-less handling of fine-grained timers.

The principles that underpin the operation of the X-RT framework, originally designed for symmetric multiprocessing (SMP) architectures, have been further extended to encompass asymmetric (AMP) ones. Many restrictions apply in the case of AMP architectures, for instance the lack of direct support for inter-processor task migrations and the more rigid memory models, making more difficult the plain applicability of global

scheduling policies. In this direction, this work presents a novel approach, based on a *shadow process* model, to support a subset of global scheduling policies such as R-EDF (the restricted migration variant of G-EDF) on AMP platforms.

The concepts that underpin this shadow process model have been validated on a FPGA-based platform using AMP soft-cores, analyzing how different memory layouts influence the schedulability and the overall system performances.

The FPGA-based MP-SoPC architecture has been further employed to make another step towards the improvement of scheduling performances on re-programmable hardware platforms, by means of developing a hardware-accelerated scheduling IP core. Following the research directions that many works in this field have delineated, the last chapter of this work presents the design and experimental validation of a hardware scheduler accelerator that implements the scheduling policy entirely in hardware, freeing the RTOS from most of its scheduling overhead.

1.3. Organization

The rest of this thesis is organized as follows. Chapter 2 presents the background of real-time scheduling on multiprocessors, introducing the basic notation used in the later chapters, reviews the preliminary literature in the field and presents a brief survey about the support offered by mainstream RTOSs currently available. In order to improve the readability of the thesis, the literature more closely related to the specific topics investigated is deferred to the beginning of each chapter.

Chapter 3 presents the operational principles and the design of the X-RT scheduling framework, together with the experimental results carried out on both SMP and AMP platforms, which assess its viability by means of overhead measurements and schedulability tests.

Chapter 4 focuses on algorithmic aspects of the timekeeping topic, first presenting a brief survey about data structures typically employed in modern operating systems for this purpose, and then introducing the novel ABH data structure. At first, the theoretical properties that underpin its physical structure are formally presented, followed by the implementation

details of its main operations. A set of synthetic experiments finally evaluate its actual performances on real-world platforms, comparing them with the other data structures discussed.

Part of the work addressed in Chapter 3, which is based on FPGA MP-SoPCs is further extended in Chapter 5 introducing a hardware multiprocessor scheduling accelerator. Such accelerator, which is supported by a minimal software coordination infrastructure, offloads the scheduling operations of the RTOS to the hardware and exhibits interesting performances especially as regards the release jitter of real-time tasks.

Finally, Chapter 6 presents the concluding remarks, summarizing the results presented in this thesis and outlining possible research directions that could be undertaken to further extend the work herein presented.

2. Background and related work

2.1. Embedded real-time systems

Real-time systems are defined as those systems in which the correctness of the computation not only depends on the logical correctness of the results (functional correctness) but also upon the time at which the results are produced (timeliness). Real-time does not mean fast. Many designers often erroneously assume to need a real-time system just because they are bound to performances. Most of the times those performances can be achieved just by choosing a suitably fast hardware platform. In contrast, an actual real-time system often gets by with slower hardware platforms, which are nevertheless able to make guarantees for the execution of critical operations. Real-time deals with guarantees, not with raw speed. Typical examples of real-time systems space from finance (surprisingly), to industrial control systems, automotive control units and aerospace flight control systems [Mar2006, Sch2007, Let2008].

These kinds of systems, which are very different from each other from a functional perspective, share a common nature of being reactive systems. Most embedded systems typically interface with physical hardware and carry out special purpose functions, aimed at monitoring the state and controlling the evolution of a physical process (for such reason they are sometimes referred to as *cyber-physical systems*).

For instance, in the automotive antilock braking system (ABS), the purpose of the control system is monitoring the car speed and wheels' angular speed during a braking and timely take the proper corrective actions, controlling the brake actuators, in order to avoid, as much as allowed by the car dynamics, the lock of the wheels. The rate of a typical ABS control loop is in the order of 5 milliseconds [PSG1998]. In this apparently short time span, the control system must undertake a large set of concurrent and time-bounded tasks, such as processing the data coming from the wheel sensors, estimating the vehicle dynamics, generating the waveforms that open and close the actuator valves, etc.

Timeliness, thus, is the key concern around which the design of a real-time system focuses.

Two aspects quantitatively define such timeliness:

- *Functional criticality*

  What are the consequences of missing such time requirements?

- *Usefulness function*

  Which concrete relationship exists between the time at which results are produced and the usefulness of those results? Typically, this relationship can be expressed by means of a mathematical function, which analytically describes the usefulness as a function of time.

A real-time system typically consists, from the software viewpoint, in a multiplicity of tasks. In the light of the above definitions, a real-time task can be classified as *soft* real-time or *hard* real-time, as follows:

- *Functional criticality*

  A task is defined *hard real-time* (HRT) if missing the timeliness requirements, even sporadically, can cause the failure of the system. On the other side, a task is defined *soft real-time* (SRT) if missing the timeliness requirements causes a degradation but not a critical failure of the system, thus the system can be still able to operate or even able to fully recover its state.

- *Usefulness function*

  The usefulness of a non-real-time task is independent of the time upon which the results are computed (Figure 1a). The usefulness function of a soft real-time task, instead, is a function that gradually slopes to zero after the deadline (Figure 1b). Some examples of soft real-time systems are services like voice-over-IP, digital TV, video conferencing and many other multimedia systems.

  In the case of a hard real-time task, two cases must be distinguished: the general one in which the usefulness function drops directly to zero (Figure 1c), and the *better-never-than-late* case in which the usefulness function drops to $-\infty$ (Figure 1d). Typical instances of the latter are represented by military systems, in which the consequences of a late action can be even worse than not performing that action at all.

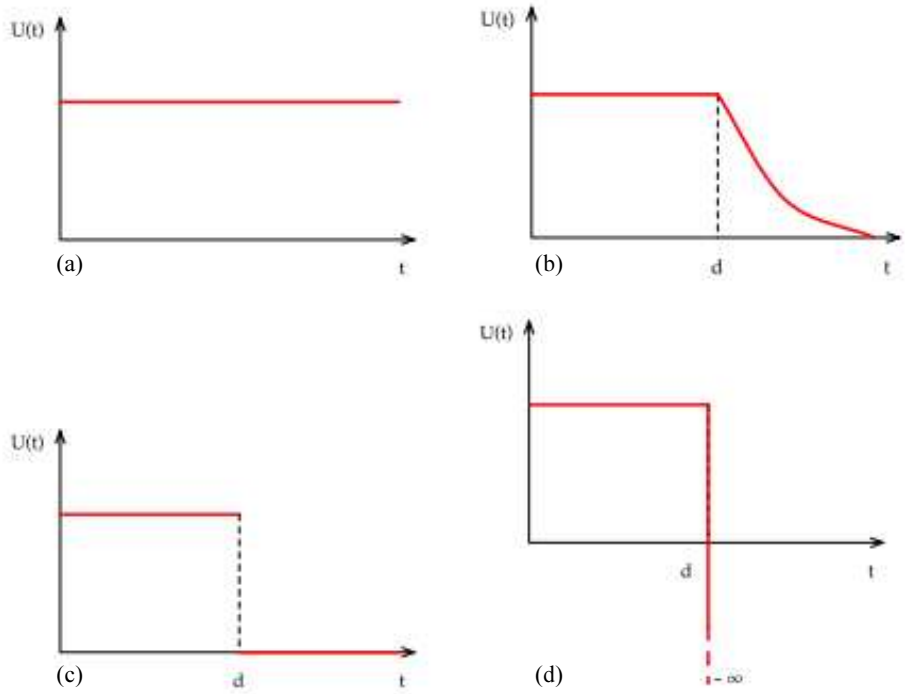


Figure 1: Usefulness function U(t) for different classes of real-time tasks

2.2. Real-time task model

A real-time task is an elementary software unit that is cyclically executed. Each execution of a given task T_i is called *job*, identified by T_i^j with *j* being the j-th invocation of task *i*. From an analytical viewpoint, the fundamental timing parameters of a real-time task T_i are characterized by the tuple $\{p_i, D_i\}$, respectively, period and relative deadline. The absolute time upon a job T_i^j becomes ready for the execution is called release (or arrival) time and is identified by r_i^j. Once released, a job T_i^j should complete within its absolute deadline, that is $d_i^j = r_i^j + D_i$, to meet the real-time requirements of the application, otherwise it is said to be late (or tardy) (Figure 2). Furthermore, each task is characterized also by a worst-case execution time (WCET), denoted with c_i, that is a worst-case assumption on the computation time required on a processor by each job.

Depending on the release policy and the relative deadline, the following three categories of tasks can be identified.

- *Periodic tasks*

  A task T_i is said to be periodic when all its jobs are cyclically and continuously released at a fixed rate, stated by its period p_i, such that the interval between the release of any two consequent jobs is always equal to $r_i^{j+1} - r_i^j = p_i$.

  Periodic tasks are usually characterized by an implicit (relative) deadline, $D_i = p_i$, equal to their period (a job should complete before the next one is released).

- *Sporadic tasks*

  Unlike periodic tasks, sporadic tasks, sometimes called *event-driven* or *reactive* tasks, become ready for their execution in response to an external event. In this case, the period p_i of a sporadic task determines the minimum time between the occurrences of the event that triggers its release (its maximum invocation rate). Thus, the interval between the releases of any two consequent jobs satisfies the condition $r_i^{j+1} - r_i^j \geq p_i$.

  In this sense, a periodic task can be viewed as a special case of a sporadic task, in which the releases of its jobs occur always at the maximum rate. Most works in literature generally deal with the notion of sporadic tasks, consequently their results apply also for purely periodic tasks.

- *Aperiodic tasks*

  It might be noted that in the two aforementioned categories there is no particular value in executing the job immediately after its release time, assuming that the deadline is met. This is not true for the case of aperiodic tasks. As the name suggests, an aperiodic task lacks the notions of period (and often deadline too), thus its jobs can arrive at any time. Usually these kind of tasks are soft-real time event-handlers that have a slightly different semantic: their purpose is to respond within the shortest possible time to external service requests, compatibly with the presence of other sporadic/periodic tasks in the system. Usually their execution is carried out by means of special

software patterns called *real-time servers* [AB1998] (not to be confused with networking topics or client/server software patterns).

All the work in this thesis focuses only on the more general case of sporadic (thus periodic) tasks.

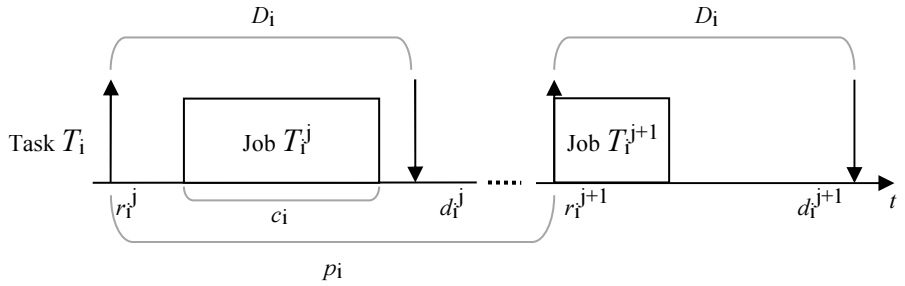


Figure 2: Temporal attributes of a sporadic real-time task.

2.3. The uniprocessor real-time scheduling problem

One of the key points of real-time systems is represented by scheduling, which involves two different aspects: an algorithmic viewpoint, that is, identifying a feasible schedule through an appropriate *scheduling algorithm* (policy), and a software viewpoint, that is, concretely enforcing the decisions of the scheduling policy on the operating system, putting into execution the tasks at the right time, through a *scheduler*.

A key concept introduced by a scheduler is represented by pre-emption.

Pre-emption occurs when a task that is executing is interrupted, its state is saved and the execution is then granted to another task. This switching of tasks on a processor is usually referred to with the term *context switch* and requires a certain amount of time to be performed [Tsa2007].

In the next paragraphs, the scheduling problem will be analysed first for the more simple case of uniprocessor systems. Later, in 2.6 such considerations will be refined and expanded in the light of the more general (and more complex) case of multiprocessor systems.

From an algorithmic viewpoint, the scheduling problem consists in the following: given a task-set τ of n real-time tasks $T_1..T_n$, and their temporal attributes as previously defined in 2.2, produce a *schedule*, which in every instant identifies which task, among the released ones, should be put into

execution. Such schedule must ensure that all released jobs execute and complete on time, under the following constraints: **(i)** jobs are not scheduled before their release time, and **(ii)** precedence among jobs of the same task is respected.

Definitions

- *Schedulability*

  A task-set τ is said to be HRT schedulable on a platform by an algorithm A if A always produces a feasible schedule for τ (i.e., no job of τ misses its deadline under A).

  A task-set τ is said to be SRT schedulable under A if the maximum tardiness of its tasks is bounded**.** [Liu2000]

- *Feasibility*

  A task set τ is feasible on a given platform if there exists a (feasible) schedule in which every job of τ complete by its deadline.

- *Class-feasibility*

  A task set τ is said to be feasible under a class C of scheduling algorithms if τ is schedulable by some algorithm $A \in C$.

- *Optimality*

  An algorithm A is said to be optimal with respect to the class C if $A \in C$ and A correctly schedules every task system that is feasible under C. If the class C is not specified, it is usually assumed to include all possible scheduling algorithms. [JCASB2004]

- *Scheduling performances*

  As frequently used in literature, the expression A_1 has better *scheduling performances* than A_2, typically refers to the ability of a given algorithm A_1 (or a class) to feasibly schedule task-sets with higher utilization factors than another algorithm A_2.

2.4. Taxonomy

This section outlines a taxonomy of uniprocessor scheduling algorithms. Unless otherwise specified, preemption is always assumed to be allowed.

Online vs. offline classification

A first classification is made depending on the time at which the schedule is produced, distinguishing between *online* scheduling and *offline* scheduling.

In *online* scheduling, the decisions regarding how to schedule tasks are taken at runtime during the regular operation of the system. Thus, the scheduling algorithm being employed concretely represent an active (and fundamental) part of the software, which is repeatedly invoked to take the proper scheduling actions in occasion of decisional instants (e.g., the release of a task, the expiration of a timer, the completion of a job). For this reason, online scheduling algorithms, though being extremely flexible, for instance allowing dynamic insertion of new real-time tasks in the system, introduce a source of runtime overhead that must be carefully accounted. In the case of complex algorithms, in fact, the scheduling algorithm itself might demand non-negligible computational costs, making the overall system (real-time tasks + scheduler) unschedulable. Even though, online scheduling represents a very typical scenario in most modern embedded systems [JCASB2004].

Conversely, in offline scheduling, the scheduling policy does not take active part during the runtime of the real-time system, rather it is used to determine the decisional instants and the sequence of actions that the scheduler will take before the activation of the system (thus the offline term). This approach, which static and inflexible nature does not require any further comment, has the major benefit of allowing to adopt (often) optimal, yet very complex, algorithms which in some case represent the last resort, whereas the high computational demand and the tight timing constraints, don't allow online alternatives [CJGJ1978].

There might be cases, in which a scheduling algorithm falls in between these two classes, by means of a two-stage decomposition. For example, the algorithm can be decoupled into a pre-processing stage, in which the

entire task-set is subdivided into smaller subsets and the actual scheduling stage, where an online scheduling algorithm is applied on the identified subsets. The preprocessing stage, in these cases might be performed offline in order to reduce, possibly in an optimal way (with respect to the online algorithm being employed), the complexity of the problem and the runtime overhead. [BBA2011, LMM1998]

Priority-based classification

When differentiating scheduling algorithms according to the priority assignment strategy, three categories can be identified [JCASB2004].

- *Static priorities*

  A unique priority is associated to each task, and all its jobs have the same priority associated to it. Thus, if task T_1 has higher priority than task T_2, then whenever both have active jobs, T_1's job will have priority over T_2's job. An example of a scheduling algorithm in this class is the Rate Monotonic (RM) algorithm [LSD1989].

- *Job-level dynamic priorities*

  For every pair of jobs T_i^j and $T_i^{j'}$, if T_i^j has higher priority than $T_i^{j'}$ at some instant in time, then T_i^j always has higher priority than $T_i^{j'}$. An example of a scheduling algorithm that is in this class, but not the previous class, is the *earliest deadline first* algorithm [Liu2000].

- *Unrestricted dynamic priorities*

  No restrictions are placed on the priorities that may be assigned to jobs, and the relative priority of two jobs may change at any time. An example of a scheduling algorithm that is in this class, but not the previous two classes, is the *least laxity first* (LLF) algorithm [But2011].

  By definition, unrestricted dynamic-priority algorithms are a generalization of job-level dynamic-priority algorithms, which are in turn a generalization of static-priority algorithms. In uniprocessor scheduling, the distinction between job-level and unrestricted dynamic-priority algorithms is rarely emphasized because EDF is

already optimal [Liu2000]. However, in the multiprocessor case, unrestricted dynamic-priority scheduling algorithms might reveal to be more efficient than job-level dynamic-priority algorithms [JCASP2004].

WCET Awareness

A further classification shall be made as regards the information exploited by the scheduling algorithm for taking its decision. As introduced in 2.2, one of the temporal attributes which characterizes a real-time tasks is represented by the WCET parameter (e_i).

While this attribute is fundamental for almost any known schedulability analysis test (determining a-priori if a given algorithm will be able to schedule a given task-set on a given platform, under worst case assumptions), it may or may not be further required at runtime by the algorithm for taking scheduling decisions. In this regards we distinguish WCET-aware algorithms (LLF is a key example of this class), from WCET-unaware algorithms (e.g., RM, EDF). This difference has usually strong implications on the behavior of the scheduling algorithm when the actual duration of the jobs deviates from their worst case estimation.

2.5. Uniprocessor real-time scheduling

The EDF scheduling algorithm (Figure 3) is the most popular job-level dynamic priority algorithm known in literature. It schedules tasks according to a fundamental rule: at any time, the priority of a job is inversely proportional to its absolute deadline. In a uniprocessor system, it implies that, at any time, the processor executes the ready task with the earliest deadline. Ties can be broken arbitrarily in case of an even deadline.

From the scheduling performances viewpoint, EDF, due to its dynamic nature, have better performances than fixed-level priority algorithms.

Task	Period [Time Units]	Computation [Time Units]	C. Demand
T1	10	5	0.5
T2	15	6	0.4
Total computational demand			0.9

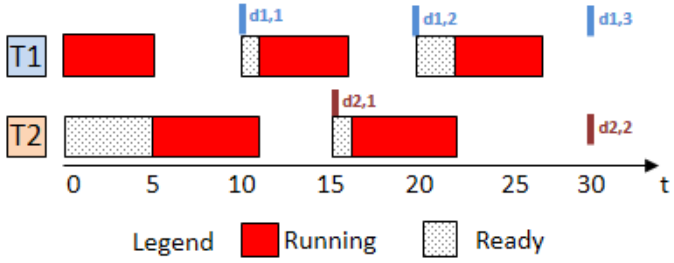


Figure 3: A sample application scheduled on a uniprocessor using the EDF algorithm.

In the particular case of uniprocessor systems, EDF is an optimal scheduling algorithm, i.e., a set of periodic hard real-time tasks with implicit deadlines can be feasibly scheduled on a uniprocessor system if and only if the utilization factor of the task set $U \leq 1$. In the more general case of sporadic tasks with non-implicit deadlines, a task-set is schedulable if (but not necessarily only if) its density $\Delta \leq 1$. (Figure 4). The optimality of EDF can be proved by using a time slice swapping technique [Liu2000], which stands on the principle that any valid schedule for any task set can be transformed into a valid EDF schedule.

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq U_{EDF} = 1 \qquad \Delta = \sum_{i=1}^{N} \frac{C_i}{D_i} \leq 1$$

Figure 4: Schedulability bounds for uniprocessor EDF.

On uniprocessor systems, EDF is typically preferred to other unrestricted dynamic priority algorithms (even if still optimal), for its implementative implications. In fact, the direct consequence of being a job-level dynamic priority algorithm is that the only decisional instants in which a re-schedule can be required are: (i) the release of a new job and (ii) the completion of a job. As will be discussed later in chapter 3, this has a direct impact on the

overheads involved and on the mechanisms required for its concrete implementation.

EDF, furthermore, is a key example WCET-unaware algorithm, as the priority of a job is assigned on release time and depends exclusively on its absolute deadline.

2.6. Multiprocessor systems

All the considerations made so far were considering only the simpler case of uniprocessor systems. In the following, such considerations are going to be extended to the more interesting case of multiprocessors. Before doing that, however, it might be worth making some clarifications about the architectural details of modern multiprocessor systems.

With the term multiprocessor, we identify any kind of computer system that can count on the availability of more than one computational unit for the execution of software processes. Multiprocessors themselves are not a recent concept, as they have been known to the industry since the early sixties. However, while the early multiprocessor systems were based on multiple physical chips requiring a special interconnection bus, thus being extremely expensive and complex to realize, the last generations of *multicore* microprocessors for desktop and server systems and multiprocessor systems on chip (MPSoCs) for embedded systems have definitely changed this situation, turning multiprocessing into an everyday reality [Wol2004, WJM2008].

Today, multiprocessing has clearly established as the mainstream approach for taking advantage of the high integration level of silicon. becoming a reference model for almost every scale of computational platform, from embedded and low power MPSoCs as the popular NVIDIA Tegra and TI OMAP chips employed in modern consumer electronics, to large scale high performance computing clusters as the 48-cores Intel SCC processor or the NVIDIA Tesla family of general purpose graphics processing units (GPGPUs).

Nevertheless, while multicore chips and multiprocessing in general are clearly acknowledged as the standard that will drive electronics for the coming years, the internal architecture of such multiprocessors has not yet

converged to a common reference model. Rather, a wide variety of different architectural trends is currently part of active debates in both the industrial and scientific community [Let2008, Mar2006, Sch2007].

Diversity is not merely a challenge across the range of multiprocessor architectures. Within a single system, the architecture of cores can vary, and current trends are more toward a mix of different cores. Some platforms are endowed with cores that share the same instruction set but different performance characteristics [LBKH2007], since a processor with a small number of large cores is generally inefficient for readily parallelized programs, but, on the other side, a large number of small cores may perform poorly on sequential software. Another trend is represented by having cores with different sets of instructions for specialized functions (e.g. mixing general-purpose and DSP cores), often embedding many further peripherals, such as GPUs, network interfaces, and other application-specific (and programmable) accelerators [HCMP2007, LSK+2005, RSJK2005].

Among the many new issues that multiprocessing brings in both hardware and software design, the choice of the reference architectural scheme, either symmetric multiprocessing (SMP) or asymmetric multiprocessing[2] (AMP), represents the first crucial design issue, involving complex trade-offs between the high-level services exposed by the software platform and the low-level hardware requirements.

SMP involves a set of closely coupled and architecturally identical processors interfaced to a shared bus, which operate as a single resource pool (Figure 5a). The platform exhibits a coordinated environment in which a unique operating system yields a homogeneous view of the physical memory and is able to dynamically execute and migrate tasks on any processor.

This scheme, which is widely used in most desktop and server PCs, is generally very straightforward to handle from a software perspective and

[2] The term asymmetric, itself, does not imply a difference between processors architectures. The prefix heterogeneous is usually preferred for this purpose. In this thesis, only the case of homogeneous processors (sharing the same instruction set) is being considered.

paves the way to dynamic load balancing of computationally intensive multitasking applications by means of software processes migrations. The price to pay for the cleaner programming model (uniform memory view) is represented by a high number of dedicated hardware mechanisms (e.g., interlocked operations, cache coherency management, IRQ routing) which can be very costly for an embedded system, due to both area and power requirements.

For such reasons, in the case of small scale and highly-integrated embedded systems, AMP often reveals to be the leading choice. AMP can be viewed as a multi-uniprocessor scheme (Figure 5b), in which the processors are independent and don't necessarily share all the physical memory space. Typically more loosely coupled mechanisms for inter-processor communication are offered by the hardware, mostly FIFOs and mailboxes. This restriction has a strong impact on the overall software organization, which needs to be approached in a decentralized fashion: distinct OS instances must be independently executed on each processor, working as separate environments, lacking any direct support to process migrations among different processors.

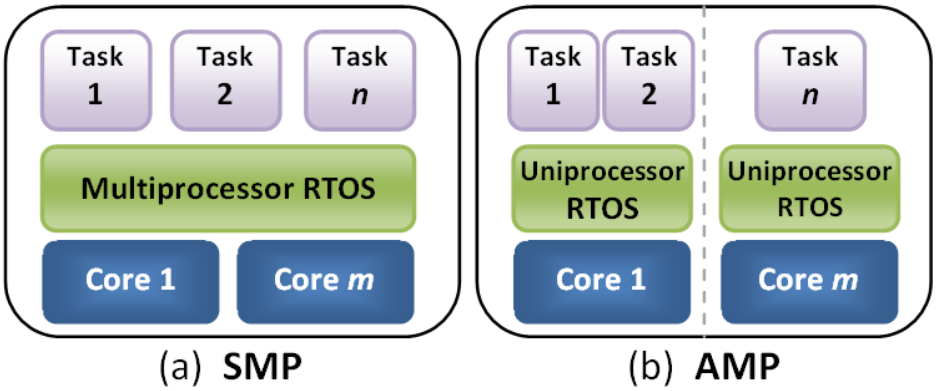


Figure 5: Overview of SMP and AMP processor architectures.

Even within the SMP area, still a variety of architectural models exists. In the last decade, the computational speed of processors has increased faster than the system memory bandwidth [MHSM2009]. As a result, as processors try to access the main memory more frequently, the shared nature of the memory bus creates bottlenecks. In summary, as more processors are added, the scalability of the SMP system becomes

problematic. To overcome these scalability issues, two approaches have been developed: on-chip cache memories and distributed point-to-point memory interconnects.

Cache memories tend to hide the latency for most of the access in system memory, allowing processor speeds to increase at a greater rate than RAM memory speeds by leveraging the spatial and temporal locality of memory accesses. The beauty of caches is their effective operation with very little impact on the programmer or compiler. In other words, details of the cache hierarchy do not affect the instruction set architecture and their operation is completely based on hardware and transparent to software programmers.

While cache memories have marginal issues on a uniprocessor system, they heavily complicate the memory consistency in multiprocessors. The root of the problem lies in store propagation. While two processors in a system (say P1 and P2) may both load the same memory block into their caches, a subsequent store by either of the processors would cause those values in the cache to differ. Thus if P1 stores to a memory block present in both the caches of P1 and P2, P2's cache can hold a potentially stale value. This cache incoherence would not be problematic if P2 never again loads to the block while still cached or if the multiprocessor did not support the shared-memory programming model. But since the point of SMP is to support that, at some point future loads of the block by P2 must receive the new valued stored by P1, as defined by the model. That is, P1's store must potentially affect the status of the cache line in P2's cache to maintain coherence.

As regards interconnection with system memory, the high clock rate reached by modern memories, together with tight power budgets and scalability constraints, do not allow any more solutions like the *front-side bus* pattern, which leaded for years in SMP systems. Instead, the memory interconnects are moving towards point-to-point asynchronous connections (Figure 6) such as, for instance, the Intel QPI or ARM AMBA AXI, were high speed serial lanes interconnect the various cores, which have dedicated channels to the memory controller, ending up in non-uniform memory architectures. While the memory architecture does not reflect on

the software functionality (at least on SMP), the performances of software application can be severely biased by the patterns of memory accesses.

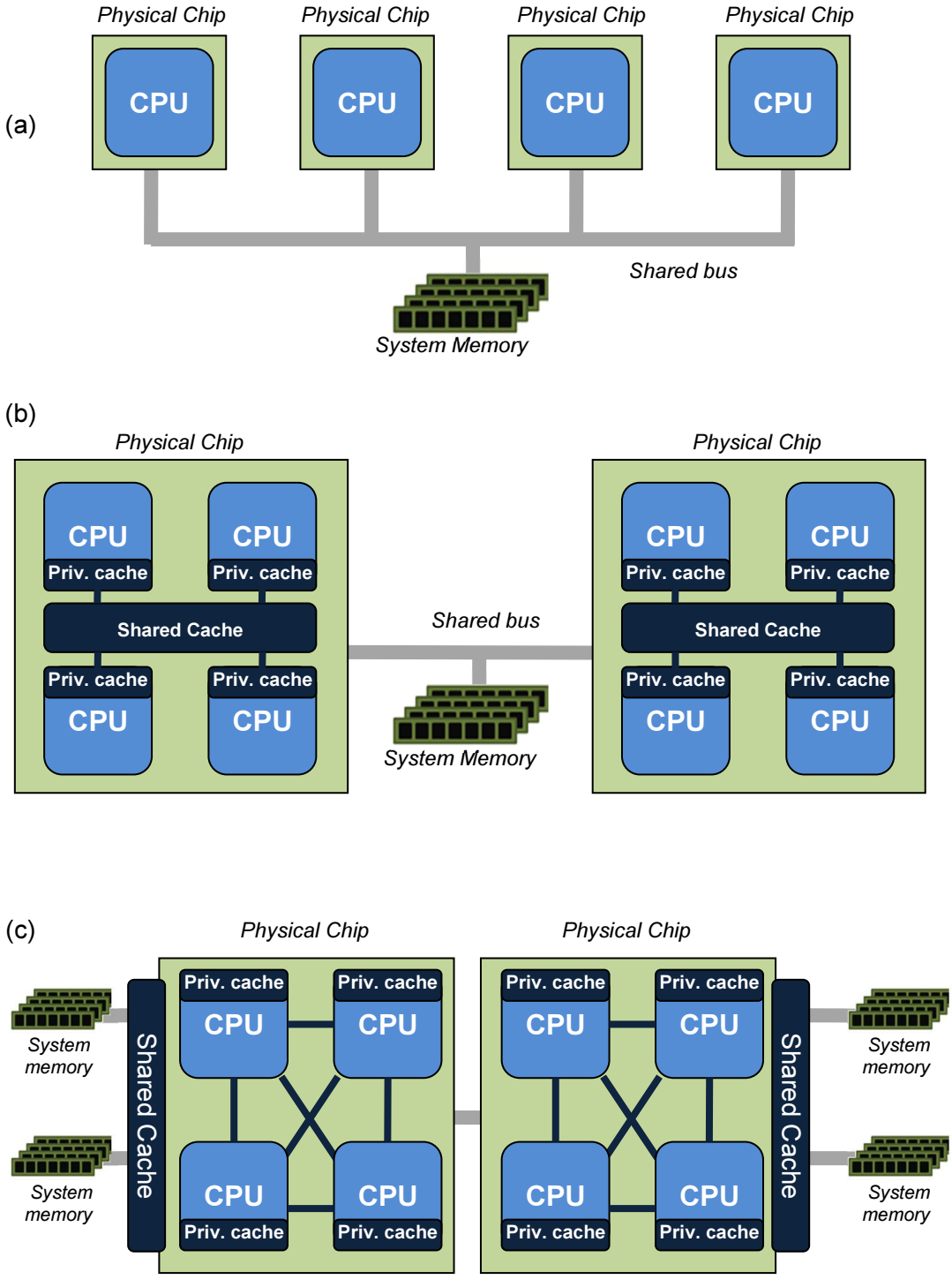


Figure 6: Evolution of SMP computer architectures (a) Multi-chip multiprocessors; (b) FSB-based multicore processors; (c) Interconnect-based multicore processors.

2.7. Multiprocessor real-time scheduling

While the computational power exhibited by multiprocessor platforms sounds appealing, paving the way towards the integration of several computationally intensive real-time tasks in a single system, extending real-time scheduling algorithms to multiprocessors is not straightforward as it might appear, neither from the theoretical viewpoint, nor from the software's.

From an analytic perspective, multiprocessing introduces a new degree of complexity, that is, the task-to-processor mapping. The scheduling problem statement of 2.3 must be refined, introducing:

- A further assignment variable that consists in:

  m identical processors $P_1..P_m$.
- Two additional constraints:

  (iii) Each processor is assigned to at most one job at any time.

  (iv) Each job is scheduled on at most one processor at any time.

Conversely to what may seem obvious, in the real-time domain the increase of computational resources does not always lead to an improvement of the scheduling performances. A key example is represented by the *Dhall's effect* presented in [DL1978], which shows that under particular distributions, the deadline monotonic priority assignment may systematically miss deadlines of task-sets with arbitrary low utilization factors.

Before delving into a deeper analysis of the implications of multiprocessors on scheduling algorithms, a further refinement shall be made to the taxonomy previously introduced. The new degree of freedom introduced by the increased number of processors, can give the possibility (depending on kind of hardware multiprocessor architecture) of performing software migrations, that is suspend the execution of a given process and resume it on a different processor.

In this regard, three new classes of scheduling algorithms, orthogonal to the taxonomy previously delineated, can be identified:

- *No migration (partitioning) algorithms*
  No-migration scheduling algorithms presuppose a static partitioning of the *n* application tasks into *m* disjoint subsets. Each subset is then locally scheduled on each processor employing a uniprocessor scheduling algorithm, as discussed in 2.5. As a result, all jobs of each task are always executed on the same processor. (*Figure 7a*)

- *Full (job-level) migration algorithms*
  Full migration policies, conversely, involve a single system-wide scheduler that is allowed to delegate the execution of each task to any processor. Moreover, the execution can be suspended, possibly more than once, and later resumed on a different processor. (*Figure 7b*)

- *Restricted (task-level) migration algorithms*
  Restricted migration policies, finally, envisage that the execution of different jobs of any task may be delegated to different processors, with the only constraint that every job, even if pre-empted, has to be entirely executed on the same processor. (*Figure 7c*)

Partitioning is widely used, because it is very simple and straightforward to implement and its performances are reasonable when employing popular and well-known uniprocessor algorithms such as EDF and RM. In practice, partitioning approaches reuse the knowledge of well-known and deeply studied uniprocessor algorithms that the scientific community has developed over years [Bak2005, BBA2010, CY2012, LRL2009, NNB2010].

However, partitioning introduces several flaws. The problem of allocating a set of tasks to a set of processors is analogous to the bin-packing problem. In this case, the tasks are the objects to pack, of size equal to their utilization factor. The bins are processors with a capacity equal to the schedulability bound for the chosen algorithm (1 for EDF). The bin-packing problem is known to be NP-Hard in the strong sense [GJ1979]. Hence, optimal partitioning is very unlikely to be computed online due to

the run-time overhead which would be involved. In these cases, the partitioning is typically performed separately as an off-line stage, and only the *m* instances of the chosen uniprocessor algorithm are computed online. Alternatively, fairly good approximation algorithms are known for bin-packing [BC2003, Ken1996]. However these do result in a loss of optimality and might be unable to schedule task-sets that are schedulable using offline optimal partitioning strategies [BLOS1995, CJGJ1978, DL1978, SVC1998]

Furthermore, in dynamic systems, where new tasks may join the system at runtime, partitioning is problematic since the arrival of a new task in the system might require to re-partition the entire task-set, hence incurring in non-negligible runtime overhead.

Task	Period [Time Units]	Computation [Time Units]	C. Demand
T1	4	2	0.5
T2	5	2	0.4
T3	10	8	0.8
Total computational demand			1.7

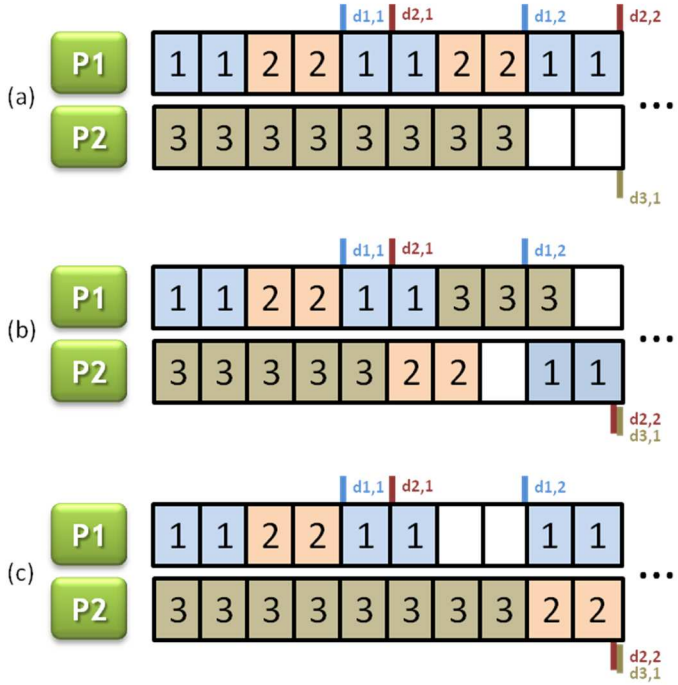


Figure 7: Examples of possible schedules of a task-set produced by: (a) a partitioning algorithm, (b) a full migration algorithm, (c) a restricted migration algorithm.

The biggest problem of partitioning, however, is that it is inherently suboptimal when scheduling periodic tasks. For instance, consider a slight variation of the previous task-set as depicted in Figure 8 with the same 3 tasks, with the same periods of respectively of 4, 5 and 10 time units, but slightly different computation times of, respectively, 2, 3 and 6 time units. It might be immediately noted that even if the total utilization factor has not changed, this time no partitioning scheduling algorithm can be able to schedule this task-set on a system with $m < 3$ processors, since the computational demand of any possible subset of 2 (or more) tasks will exceed the computational capacity of a single processor (1.0).

Task	Period [Time Units]	Computation [Time Units]	C. Demand
T1	4	2	0.5
T2	5	3	0.6
T3	10	6	0.6
Total computational demand			1.7

Figure 8: A slightly different version of the former example, not schedulable anymore with a partitioning approach.

From a theoretical viewpoint, the utilization bounds known so far (thus the guarantee of being able to schedule a task-set with a given utilization factor) for the partitioning versions of RM (p-RM) and EDF (p-EDF) are conservative. In general, no partitioned algorithm has a worst-case utilization on m processors larger than $(m + 1) / 2$. To see why, note that $m + 1$ tasks, each with utilization of $(1 + \varepsilon) / 2$, cannot be partitioned on m processors. As ε tends to 0, the total utilization of such a task-set tends to $(m + 1) / 2$ [BC2003].

In general, better utilization bounds can be found when introducing restrictions on per-task utilization. Supposing that U_{max} is the maximum utilization of any task in the task-set, any task can be assigned to a processor that has a spare capacity of at least U_{max}. This implies that, if a set of tasks is not schedulable, then every processor must have a spare capacity of less than U_{max}. Hence, the total utilization of such a task-set is more than $m(1 - U_{max}) + U_{max}$. Equivalently, any task-set with utilization of

at most $m - (m - 1)\ U_{max}$ is schedulable. In [LGDG2000] Lopetz et al. have used bin-packing techniques to slightly improve this bound, proving that the worst bound achievable on m processors is $(\beta\ m + 1) / (\beta + 1)$, where $\beta = \lfloor 1/\ U_{max} \rfloor$

Full migration algorithms (often known in literature as *global scheduling approaches*), on the other side, generally provide better performances, achieving higher utilization factors, especially in bounded tardiness soft real-time systems [ABD2005, DA2008, EDB2010]. However, they make the assumption of an underlying SMP platform, in order to handle a shared tasks queue and perform inter-processor job migration.

No-migration policies, conversely, may be applied, at least from a conceptual viewpoint, on both SMP and AMP platforms, since they substantially operate as a multiplicity of legacy uniprocessors. As AMP turns out to be the only architectural scheme supported in most low-power embedded MPSoCs, many studies are currently being undertaken, some investigating on partitioning approaches [NVC2010, XWB2007, KBDV2006], others aiming to extend SMP facilities to AMP platforms [HBK2005, HCMP2007]. Although the latter prove to be functionally correct, the overhead they introduce is not negligible and the overall platform does not scale well as the number of cores increases.

A reference paper which analyzes the viability of supporting sporadic real-time task-sets on SMP platforms, taking into account also the resulting overheads, is represented by [BA2009], conducted by Brandenburg and Anderson at the university of North Carolina at Chapel Hill (USA). To facilitate this line of research, the UNC's Real-Time Group has developed an open-source project called LITMUS^{RT} (Linux Testbed for Multiprocessor Scheduling in Real-Time systems), an extension of the Linux kernel for benchmarking multiprocessor scheduling algorithms on SMP hardware platforms supported by Linux.

In [CLB+2006], Calandrino et al. used LITMUS^{RT} to evaluate five well-known multiprocessor real-time scheduling algorithms on a four-processor (non-multicore) 32-bit 2.7 GHz Intel Xeon SMP platform. On this small SMP platform, with relatively large private L2 caches, each tested algorithm proved to be the preferred choice in many of the tested

scenarios. In particular, global algorithms outperformed partitioned algorithms in supporting SRT workloads.

In [BCA2008], Brandenburg et al. analyzed the scalability of several global and partitioned algorithms. This evaluation was conducted on a much larger and slower multicore platform: a SUN Niagara with a small single shared L2 cache and 32 logical processors, each with an effective speed of 300 MHz. As before, each tested algorithm was found to perform better than the others for some subset of the considered scenarios. Particularly, it was observed that global algorithms are heavily affected by run-queue related overheads.

In [BA2009], Brandenburg and Anderson evaluated seven possible implementations of G-EDF in LITMUS^{RT} on the above-mentioned Niagara platform. Tradeoffs involving different synchronization schemes for the scheduler's data structures were found to significantly impact schedulability.

Restricted migration policies have received less attention. In [BC2003, FB2004] a restricted-migration variant of the earliest deadline first algorithm (r-EDF) is proven to be not worse than the highest known utilization bound for global fixed priority scheduling.

Finally, it is also worth noting that non-preemptive versions of global policies, as NP-G-EDF [Bar2006, KM2005] still fall in the case of restricted-migration policies.

Restricted migration policies can bring significant benefits even for very simple real-time applications as the one depicted in Figure 9, which considers the scheduling of three tasks (the parameters associated to each task represent, respectively, its computation time, period and resulting processor utilization) on a 2-way multiprocessor system. It is evident that no-migration policies are not able to feasibly schedule this task-set since the utilization factor of any pair of tasks exceeds the computational capacity of a single processor.

However, an r-EDF policy could be able to schedule successfully the application meeting all deadlines, as shown in Figure 9.

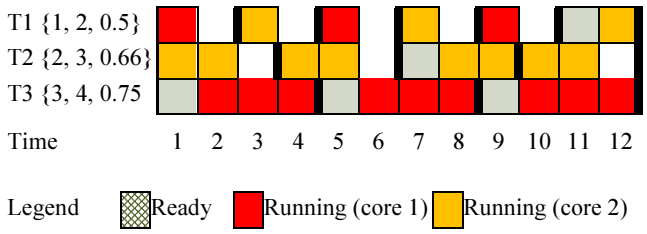


Figure 9: Schedule of a sample application using r-EDF.

2.8. Real-Time operating systems

Operating systems play a key role in the software organization of an embedded system. The main purpose of an operating system is to provide software application developers with uniform and high level interfaces that abstract, as much as possible, the details of the underlying hardware platform.

The variety of services offered by modern operating systems is so wide that even a simple enumeration of their salient aspects is impracticable and would probably require more than the overall length of this thesis, going far beyond the purposes of this work. Instead, if we narrow the scope to the real-time systems scenario, the subset of interfaces and services has a definitely lesser extent. There are, in fact, few but fundamental building blocks that are nowadays considered a must, in order to classify an operating system as real-time.

Process model

The main requirement of an embedded system is to carry out several concurrent activities that must react, within firm deadlines, to a wide number of synchronous and asynchronous inputs. In the era of PLC and bare-metal microcontrollers (which is not yet faded away at the time this thesis is being written), a software pattern typically employed to handle this concurrency is represented by the cyclic executive pattern [Mac1980]. The principle of cyclic executive lays in organizing the software into multiple sub-programs (tasks), smaller, still sequential, portions of code that handle a single activity, processing the corresponding inputs and producing the relative outputs. The sub-program partitioning itself, however, is not a great deal. The challenge, instead, is represented by the

fundamental requirement that, in order not to break the model and ensure the timeliness of the overall system, the code of all these tasks must be strictly asynchronous, i.e., non-blocking.

The ever-increasing complexity of modern embedded applications, however, make the pure-asynchronous software modelling an extremely hard task for application designers. Furthermore, single-threaded patterns like the cyclic executive lack any form of isolation, thus, a single task holding the CPU for more than expected (or even worse, stuck in a loop) have catastrophic consequences on the operation of the overall system.

For such reasons, a desirable feature for modern RTOSs is represented by multitasking support. Multitasking itself is definitely not a new concept, nor an exclusive prerogative of RTOSs, and it is not in the aim of this thesis to discuss the details of its operational principles. What is more interesting to highlight instead is that, from the software viewpoint, there are different ways in which multitasking can be achieved, commonly referred to as *process models*.

- *Heavy process model*, inspired to BSD-style processes. Each process is an independent execution unit, with a private address space and a private set of resources (files, I/O apertures, etc.). This model offers the strongest guarantees in terms of isolation between processes, ensuring that in cases of unhandled faults, only the process causing the fault is aborted and its resources are properly cleaned up by the operating system. However it has two main drawbacks: resource sharing between processes, a feature often desirable, is more complex and expensive; context switching between processes is generally more expensive compared to the other models (for instance in x86/amd64 architectures it further implies a flush of the TLB cache, and in many ARM architectures with virtually-indexes-virtually-tagged caches and no MVA support it requires also flush of the data and instruction caches) .
- *User Threads model*, sometimes also referred to as lightweight process model. Each thread is still an independent execution unit with its own context, but shares its address space and most of the resources with other threads. This model is more flexible than the heavy processes one, allowing transparent sharing of resources and more lightweight

inter-thread coordination patterns, requires generally less context switch overheads, but is definitely less dependable, since an unhandled exception in a thread causes the abort of all the other threads in the same address space.

- *Kernel threads model*, is an extremization of the former case, in which all threads are executed in kernel space. It offers great advantages in terms of performances, since the overhead for invoking system calls is almost zero, and the context switch between threads is extremely fast, since all the system threads share the same address space with the kernel. At the same time this model is extremely dangerous since a bug in any part of the application can corrupt the kernel data structures and jeopardize the reliability of the entire system. This, however, does not stop it being very used in many low latency mission critical systems.

In general, the three models just briefly presented, are not exclusive, and many RTOSs offer all of them, though with completely different interfaces not exchangeable with each other.

Since the principles that will be discussed in this thesis are orthogonal to the particular process models, which is mainly a design choice of the application designer, the term *process* will be generically used to identify the runtime abstraction provided by the operating system, without a particular reference to heavy processes or user/kernel threads.

Besides the specific process model, the strong difference between general-purpose and RTOSs is represented by process scheduling. While the former try to ensure fairness among ready processes, often using very complex yet effective metrics to do that (e.g. I/O ratio, scheduling history, interactivity with the user, etc.), RTOS schedulers obey more simple but very strict rules.

In a RTOS each process is associated to a priority level, typically a number within a predetermined range, which in the most RTOSs can be changed at runtime through a dedicated system call. At any time the RTOS must ensure that the ready process (the *m* ready processes in the case of a SMP RTOS) with the highest priority must be running on the CPU(s). On this (and some others discussed later) apparently simple principle, that takes the name of *static priority driven scheduling*, relies most of the theory of

real-time systems and the operations of the most mission-critical applications.

The reasons why the application of this principle is not as simple as it might seem, are manifold: in the case of SMP, such requirements imply that the RTOS must be able to coordinate the processors and migrate processes among them at any time; secondly, ensuring strict priority-driven scheduling becomes very challenging in presence of events that alter the nominal execution flow of processes, such as critical sections, waiting queues, signals and message exchanges.

Synchronization primitives

Another fundamental requirement for a RTOS is represented by inter-process synchronization primitives. Many aspects are involved into the synchronization topic. The most evident is undoubtedly represented by critical sections and the corresponding patterns to deal with those, which are semaphores and derivatives (e.g., mutexes). Again, semaphores are not an exclusive prerogative of a RTOS, as synchronization is a more general need in many other software engineering fields. The key difference, for a RTOS, is represented by the way critical sections are handled by the scheduler (for instance the order in which processes pending on a semaphore should be awaken when a post operation is performed). General purpose operating systems typically don't follow strict rules, or try to balance fairness also in these situations. In RTOSs, instead, many concerns affect the operations of semaphores. At first, a typical requirement of most real-time applications is to respect the process priority scheme when awakening processes pending on a semaphore. Furthermore, another important issue related to critical sections is represented by priority inversion.

Priority inversion is a problematic scenario that occurs when a high priority task is pending on a critical section which is currently acquired by a low priority task, and that low priority task is pre-empted by a third task which has an intermediate priority (Figure 10). This scenario, apparently legitimate at first glance, causes a scheduling paradox in which a high priority process is effectively pre-empted by a lower priority process (the

medium priority one), which can have a severe impact on the response times of a real-time system.

Among the many approaches that can be adopted to avoid the extent of this problem, a simple one is represented by priority inheritance [SRL1990]. Priority inheritance is a feature required to RTOSs, which provides that the priority of a process executing in a critical section (P_L in the example of Figure 10) is temporally raised when one or higher priority processes are pending on the same critical section, to the highest of those priorities. This allows reducing the waiting time for many cases as the one being considered in the example, and more importantly, keep it under an upper bound that can be determined analytically, provided that the dynamic of the processes is known.

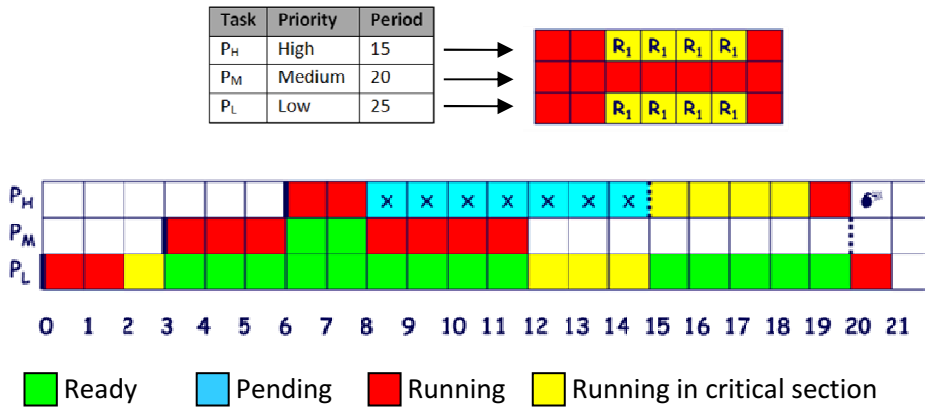


Figure 10: A sample instance of a priority inversion problem.

Critical sections, however, are not the only form of inter-process synchronization, interesting mainly the shared-memory software scenarios. In the case of more loosely coupled interaction, for instance in client-server scenarios, another fundamental mechanism part of most RTOS set of services is message queuing. Message queues are an inter-process communication mechanism that allows many-to-one interaction by means of message exchange.

While the implementation details (like the length and granularity of the messages) usually change from RTOS to RTOS, the main operating principle holds among the various platforms. A message queue is a conventional queue that is typically initialized by the server with a

predetermined maximum length. Messages are typically enqueued in the queue according to two patterns: conventional first-in-first-out (FIFO), or priority-based. In the latter case, the message queue behaves exactly as a priority queue: each message is inserted in the queue with a specific priority, and the consumer peeks always the message with the highest priority. The critical issue, from the scheduling viewpoint, is represented by the enqueue case when the queue is full. In the case of synchronous (blocking) enqueueing, the RTOS must ensure that processes blocked on the queue are awakened according to their priority. Many variants of message queues are typically found in modern RTOS, as for instance byte-oriented message-queues (pipes), or variable length and loosely time-coupled mailboxes.

Timers

A third key aspect of RTOS is represented by timekeeping. For the most time-driven real-time applications, timers represent a fundamental building block. It is very common that a real-time application can need more timers than the effective availability of the hardware platform. In this regard, a RTOS must be able to handle a large number of software timers using a small number of hardware timers. Typically RTOS make a distinction between coarse grained timers, timers with large timeouts which don't need a precise accuracy (for instance most network and I/O timeouts) and high resolution timers (typically below the millisecond range). A detailed discussion about timekeeping is postponed to Chapter 4.

2.9. IEEE POSIX standards for real-time applications

The POSIX ***P****ortable* ***O****perating* ***S****ystem* ***I****nterface* is a collection of standards developed by IEEE that define application program interfaces (APIs) for accessing operating systems services, aimed to provide a platform-independent abstraction layer for maintaining compatibility across different operating systems. POSIX specifications consists in 27 documents grouped in 3 sets: (1) *POSIX core services* (kernel APIs for process creation and control, signals, exceptions, timers, pipes, I/O); (2) *POSIX commands and utilities* (user portability extensions, corrections and extensions, protection and control utilities and batch system utilities); (3) *POSIX Conformance testing.*

Here is a list of the standards defined by POSIX:

```
IEEE 1003.0 Guide to POSIX
IEEE 1003.1 System API (C language)
IEEE 1003.1a System API extensions
IEEE 1003.1b Real-time and I/O extensions
IEEE 1003.1c Threads (was: POSIX.4a)
IEEE 1003.1d More real-time extensions
IEEE 1003.1e Security extensions, ACLs
IEEE 1003.1f Transparent network file access
IEEE 1003.1g Protocol independent communication, sockets
IEEE 1003.1h Fault tolerance
IEEE 1003.1i Technical corrections to POSIX.1b
IEEE 1003.1j Advanced real-time extensions
IEEE 1003.1k Removable media API
IEEE 1003.1m Checkpoint/restart
IEEE 1003.1n Fixes to .1,.1b,.1c,.1i
IEEE 1003.1p Ressource limits
IEEE 1003.1q Trace
IEEE 1003.2 Shell and common utility programs
IEEE 1003.2a More tools and utilities
IEEE 1003.2b More utilities
IEEE 1003.2c Security utilities
IEEE 1003.2d Batch processing utilities
IEEE 1003.2e Removable media utilities
IEEE 2003 Test methodology (was POSIX.3)
IEEE 2003.1 Test methods for POSIX.1
IEEE 2003.1b Test methods for POSIX.1b
IEEE 2003.2 Test methods for POSIX.2
```

In 1993, the POSIX. 1003.1b (formerly 1b-1993) has introduced new definitions and APIs for dealing with real-time applications. These new POSIX extensions focus on the requirements of real-time applications and high performance I/O. Many applications like interactive video games, high performance database servers, multimedia players and control software for all kinds of hardware require more deterministic scheduling.

POSIX Process model

As regards the process model, POSIX defines both the heavy process and the users threads models. The interface specified by POSIX for processes is pretty straightforward, consisting substantially into 3 families of system calls: *exec*, *fork* and *wait*, which handle respectively, process creation, duplication (spawning a process which share the same code with the parent, but with a dedicated address space) and wait for termination. The listing below shows a brief sample of a multitasking application realized using *fork* and *wait* primitives:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5.
6. int main(int argc, char* argv[]) {
7.   /* Opaque type for pid encapsulation */
8.   pid_t child;
9.
10.  /* fork a new child */
11.  child = fork();
12.  if (child != 0) {
13.    /* Parent code */
14.    /* Waits for child completion */
15.    int status = 0;
16.    waitpid(&status);
17.
18.    printf("Child with pid=%d “
19.           “exited with status code %d\n",
20.           child, status);
21.  } else {
22.    printf("Child started with pid=%d\n", getpid());
23.  }
24.  return 0;
25.}
```

While the isolated memory address space represents a winning strategy for ensuring robustness and reliability of applications, it soon turns out to be a very limiting and expensive choice when the taking into account the necessity of associating different concurrent execution flows to the same address space for sharing resources.

For such reasons, POSIX introduces also the concept of threads (usually abbreviated with the term *pthread*) [DM2003]. A pthread cannot be instantiated as standalone item, rather it must be created within a process, which marks the isolation boundaries for the memory address space and resource pool that all its thread will share.

From the scheduling viewpoint, a thread represent the elementary scheduling unit perceived by the operating system. In order to keep this design choice consistent with the process model, each process upon creation is implicitly associated to a thread, corresponding to the *`main()`* process' entry point.

Dedicated primitives allow the instantiation, signalling and synchronization of threads, respectively: *`pthread_create`*, *`pthread_interrupt`* and *`pthread_join`*. The listing below shows a brief example of a multithreading which uses POSIX threads.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #define NUM_THREADS 5
5.
6. /* Entry point where each thread starts its execution */
7. void* thread_entry(void* arguments) {
8.  printf("This is thread #%ld\n", pthread_self());
9.  do_some_work();
10. pthread_exit(NULL);
11.}
12.
13.int main (int argc, char* argv[]) {
14. /* Array of thread descriptors */
15. pthread_t threads[NUM_THREADS];
16.
17. /* Starts a number of threads */
18. for(int i = 0; i < NUM_THREADS; i++) {
19.   /* pthread_create instances a new thread */
20.   rc = pthread_create(&threads[i], NULL,
21.                       thread_entry, NULL);
```

```
22.
23.   /* Check for errors */
24.   if (rc) {
25.     perror("pthread_create() failed.");
26.     exit(rc);
27.   }
28. }
29.
30. /* Waits for completion of all threads */
31. for(i=0; i<NUM_THREADS; i++)
32.   pthread_join(threads[i], NULL);
33.
34. return 0;
35.}
```

Besides the process model, real-time extensions include also rigid standards that define how thread executions should be handled by the operating system. POSIX defines a static priority-driven scheme, in which each thread is associated to a priority level, in the range 0 (lowest) to 99 (highest). Three main scheduling classes are envisaged by the standard, namely, *SCHED_FIFO*, *SCHED_RR* and *SCHED_OTHER*.

SCHED_OTHER is the custom operating system's time-slicing scheduler used for non-real-time threads, which are assigned a symbolic static priority of 0 (most operating systems typically define further private mechanisms for handling scheduling of non-real-time processes, like Linux's nice value).

SCHED_FIFO and *SCHED_RR*, conversely, are intended for real-time scheduling. Both can only be used with priorities higher than 0, which means that when a real-time thread becomes ready, it will always pre-empt immediately any other non-real-time thread.

Both *SCHED_FIFO* and *SCHED_RR* define conventional static priority scheduling policies as previously introduced in this chapter. When a thread belonging to one of these classes becomes ready, it is inserted at the end of the list for its priority level. A change of the priority class or level, through the, *sched_setscheduler* or *sched_setparam* system calls, will put the

thread at the start of the list, if it is ready, allowing it to pre-empt the currently running process if it has the same priority [3].

A call to *sched_yield* has the effect of moving the caller thread at the end of the list for its priority level. No other events move a thread belonging to the *SCHED_FIFO* class in the priority list, thus a ready thread runs until either it is blocked (e.g., for an I/O request or for a mutex), it is pre-empted by a higher priority thread, or it calls *sched_yield*.

SCHED_RR is a simple variant of *SCHED_FIFO*, that adds a further round-robin semantic for ready threads on the same priority level, introducing the concept of scheduling quantum. A ready thread belonging to the *SCHED_RR* class is executed for a maximum time equal to the scheduling quantum, after which it is moved at the end of the list for its priority level. A *SCHED_RR* process that has been pre-empted by a higher priority process and subsequently resumes its execution, will complete the unexpired portion of its quantum and then yield. The length of the scheduling quantum for *SCHED_RR* threads can be retrieved using the *sched_set_interval* system call. Unfortunately, POSIX does not specify any mechanism for controlling the size of the round-robin time quantum. Typically, POSIX compliant operating systems define proprietary (non-portable) system calls to change it.

POSIX real-time signals

A first basic inter-process communication mechanisms defined by POSIX standards is represented by signals. Signals are primarily meant as a notification mechanism, such as exception handling or asynchronous interrupts. Signal delivery in POSIX is on per process basis. A process can signal another process to synchronise and communicate. Each process can define several service functions for registered signals, called signal handlers. When the system delivers a signal to the process, the signal handler is executed asynchronously, in practice implementing in user-

[3] In this regard there is a mismatch between POSIX 1003.1 and the behaviour of some operating systems. The standard specifies that the thread should be placed to the end of the list.

space the idea of immediate and asynchronous interruption, just like interrupt handlers do inside kernels.

Each signal in POSIX is associated to an integer value. Many POSIX signals are pre-defined and used, in an implementation-dependent way, by the operating system. The user, however, has the possibility of defining custom signals for its own purposes and register them to custom signal handlers.

Standard POSIX signals can be generated via the *kill* primitive. A process can request a signal be sent to itself: when a timer expires (i.e. SIGALRM), when asynchronous I/O completes, etc.

The main issue of regarding conventional POSIX signals is that they are unreliable: in most implementations, a signal can be definitely lost if the process is executing a signal handler which has masked that signal. Furthermore, the delivery behaviour is not specified in the standard, leaving to the operating system implementing the standard the freedom of choosing the delivery order and time. For such reasons, in the 1003.1b real-time extensions, a new subset of signals with a more strict semantics have been standardized. The new standards introduces a minimum number of 8 application-definable real-time signals, are numbered from *SIGRTMIN* to *SIGRTMAX*. Only those signals whose numbers lie between the two are considered real-time.

For those signals, POSIX introduces reliability guarantees, specifying that, conversely to what happens with traditional UNIX signal, a signal must be enqueued if the target process cannot accept it (the signal is masked), and delivered as soon as the process unmasks it, following the priority of the signal. Furthermore, real-time signals can optionally carry user-defined extra data, by means of a dedicated pointer to the *siginfo_t struct,* conversely to traditional signals that have only one numeric parameter carrying the number of the signal. This additional structure contains the signal number, a code which indicates the cause of the signal (for example, a timer signal) and an integer or pointer value. This capability increases the communication bandwidth of the signal mechanism. As an example, a server can use this mechanism to notify a client of the completion of a requested service and pass the result back to the client at the same time.

Real-time extensions provide a new and more responsive synchronous signal-wait function called *sigwaitinfo*. The function suspends the calling process until the specified signal is received. To use this function, the signal must be blocked, in order to avoid triggering the asynchronous handler. Similarly, the primitive *sigtimedwait* has the same semantics as *sigwaitinfo*, but allows to specify a timeout, returning an error code if no signals are received by the timeout.

Despite this strong semantic, however, signals are still too inflexible as a communication mechanism for many real-time applications. The reason lies mainly in the limitation of the number of the definable signals, the length of the queue not controllable by the user, and more importantly, their limited compliance only with the heavy process model, but not with the finder grained pthread model (signal handlers are process-wide).

There are few occasions in which signals are an appropriate communication mechanism, for instance, for rare but urgent notifications that require an asynchronous interrupt for the process or for dealing with timers.

POSIX message queues

For such reasons, POSIX defines another communication mechanism, less complex compared to signals, but more flexible as regards also inter-thread communication, that is message queues. POSIX message queues allow an efficient, priority-driven IPC mechanism with multiple readers and writers, for many aspects similar to the concept of named pipes. Conversely to named pipes, however, message queues have internal structure. More importantly, message queues are priority-driven. Whenever a writer sends a message to a queue, a priority is specified for that message. The queue will remain sorted such that the oldest message of the highest priority will always be the first one picked by the receiver.

The user has control over the geometry of a message queue. When a message queue is initialized, the user can define the maximum length of the message queue, and the maximum size allowed for messages, bounding a priori the memory required in the worst-case scenario.

A process can determine the status of a message queue, conversely to pipes, where the of the channel is unknown to the endpoints. With message queues, a process can determine how many messages are outstanding on the queue, the boundaries of the queue and the number of processes that are blocked for sending or receiving.

Like pipes and FIFOs, all message queue operations are performed based on message queue descriptors (*mqd_t*). Message queues are created and opened using the *mq_open* system call. The returned descriptor is used to refer to the open message queue in later calls. Each message queue is identified by a unique identifier. Different processes or threads can operate on the same queue by passing the same name to the *mq_open* call. Messages are transferred to and from a queue using *mq_send* and *mq_receive*. When a process has finished using the queue, it closes it using *mq_close*, and when the queue is no longer required, it can be deleted using *mq_unlink*.

The code listing below shows a brief example of use of POSIX message queues.

```
1. #include <mqueue.h>
2. #include <stdio.h>
3.
4. #define MSG_SIZE     4096
5. #define MSG_PRIORITY 0
6.
7. void main () {
8.   struct mq_attr attr;
9.   mqd_t mq;
10.  char buf[MSG_SIZE];
11.  unsigned int prio;
12.
13.  // Set up the queue attributes
14.  attr.mq_maxmsg = 100;
15.  attr.mq_msgsize = MSG_SIZE;
16.  attr.mq_flags = 0;
17.
18.  // Open the queue.
19.  mq = mq_open("/queuename", O_RDWR | O_CREAT, 0, &attr);
20.
21.  /* Producer-side code */
22.  mq_send(mq, &buf[0], MSG_SIZE, MSG_PRIORITY);
```

```
23.
24.  /* Consumer-size code */
25.  while (mq_receive(mq, &buf[0], MSG_SIZE, &prio) != -1))
26.  {
27.      printf ("Received message, priority: %d.\n", prio);
28.  }
29.
30.  // Close the message descriptors.
31.  mq_close (mqdes);
32.  mq_unlink(“/queuename”);
33.}
```

2.10. Linux as a real-time operating system

Many developers, in the last years, have been adding real-time support to Linux, trying to fill the major gap in its capabilities for real-time processing. According to a recent end-user survey [Gee2004], in the last quarter of 2004, Linux owned the highest percentage of new embedded-development projects of any operating system.

A number of real-time extensions of Linux have been proposed and implemented during the last years, for instance the compliance with POSIX 1003.1b interfaces previously introduced. In addition, further features such as high-resolution timers, priority inheritance, and shortened non-preemptible kernel sections, which enhance kernel responsiveness, have been also recently introduced [DW2005].

However, despite its high numbers in the embedded systems panorama, Linux is not yet strictly classifiable as a RTOS. A key concern for a RTOS is represented by latency, that is, the delay that might take place between the triggering of an event and the time the corresponding software application actually processes it. For instance, in the case of hardware device drivers, the amount of time that elapses between an interrupt request and the execution of the associated handler, called interrupt latency.

Many factors can contribute to latency, either hardware-related (bus contentions, DMA operations, cache misses) or software-related (interrupt masking). While the former mostly depend on the choice of a proper hardware platform, the latter require a dependable operating system that guarantees at design-time bounds on them.

At the time this thesis is being written, Linux, despite its popularity in the embedded system panorama, is not yet able to make strong guarantees on latency bounds. The main reasons for this are the non-negligible complexity level that the Linux kernel has reached after years of development, and the fact that it still inherits (and it will very likely continue to do in the future) many design choices oriented to optimize its performances in the average case behavior, as a general-purpose operating system, at the price of non guaranteed worst-case scenarios (for instance, there is no strict guarantee for how long a non-preemptible kernel section can delay the execution of a real-time process).

All these considerations, however, do not necessarily mean that Linux is 'bad' for real-time (and its numbers, in fact, contradict this). Undoubtedly, it is not the top choice for hard real-time and mission critical systems, and for real-time systems with timing requirements below the seconds range in general. However, the amazing speed of modern processors, nowadays often higher than the computational demands of the most embedded applications, tend to hide the impact of the kernel unpredictability, making Linux still an optimal compromise, especially for the world of soft real-time systems.

There have been many different approaches to real-time in Linux through the years, most of them made by third-party companies aiming at provide commercial version of Linux tailored for embedded systems.

Early co-kernel approaches

The earliest solution found for adding real-time capabilities to Linux was represented by co-kernel approaches, that is, running a small non-Linux real-time kernel side-by-side with the Linux one on the same hardware, instead of turning the standard kernel into a RTOS.

The basic principle of a co-kernel design is that the real-time co-kernel, which is responsible of the critical real-time activities for the system, has the precedence over the Linux kernel, handling all the interrupt requests (in particular timer ones), and re-dispatching them to the Linux kernel, deferring their execution with respect of the other real-time tasks. Thus, all device interrupts must go through the co-kernel before they can be

processed by Linux, so that the latter cannot delay the execution of the real-time tasks, ensuring predictable response times. Sometimes this interrupt re-dispatching, which can sound expensive for the performances non real-time application, can be optimized in presence of programmable interrupt controllers with different priority levels.

Practically speaking, a co-kernel is usually available as a set of modules, which are either dynamically linked or compiled in the Linux kernel tree, like a regular driver. Some implementations (notably RTAI and Xenomai) support the execution of real-time software in user space just like any regular application. Others (notably RTLinux) require real-time applications to be embodied in kernel modules.

Co-kernel designs exhibit great advantages in terms of predictability and isolation of the real-time applications from the unpredictable behaviour of the Linux kernel. However, such isolation have major impacts on the software design process. Since the Linux kernel is treated as *untrusted* (from the timing viewpoint), real-time applications (either in user or kernel space) are restricted to use only the real-time co-kernel services and system calls. In other words, the whole set of regular Linux drivers and libraries (which typically represent the first reason that leads towards the choice of Linux) cannot be used for real-time applications, unless forked and ported to work, in a predictable manner, in the co-kernel. For the same reason, many user-space libraries (for instance the GNU standard C library itself) cannot be relied on, since they can cause unexpected latencies due to their invocations of Linux kernel system calls. This strongly affects the programming model and the design complexity of real-time applications, basically keeping the benefits of Linux only for the subset of non real-time tasks in the application.

The Linux-RT approach

A more interesting approach for real-time in Linux is represented by the *RT* (formerly *CONFIG_PREEMPT_RT*) patch-set, currently part of the official Linux kernel tree. This solution, that appeared in 2004 as a patch for the Linux 2.6 kernel, introduced full pre-emption (not enabled by default) to the Linux kernel, aiming to turn it into a native RTOS.

The approach used for enabling full pre-emption to the original Linux kernel (and its huge set of drivers) relies on two major changes: (i) all the critical sections based on spinlocks (and rw-locks) are automatically turned into semaphore-based equivalents, making those critical sections pre-emptible; (ii) All the interrupt handlers (with a few exceptions, as the timer ones) are turned into kernel threads. The advantage of this choice is that non-critical interrupt handlers can be set to a lower priority than more critical real-time tasks, thus avoiding unexpected latencies.

The RT patch strives to covert the Linux kernel into a full RTOS with few modifications, without changing its original general-purpose design. For most applications that need real-time determinism, the RT-Linux kernel provides very adequate services. However, many mission-critical applications that also require reliability guarantees and software certifications, the Linux kernel, with or without the RT patch, is not sufficient. In contrast, it turns out to be an excellent compromise for most of the remaining non mission-critical real-time scenarios, such as robotics and many industrial control systems.

Real-time scheduling in the Linux kernel

As regards scheduling, the design of the Linux kernel reflects in many aspects its conformance to POSIX standards. The elementary scheduling unit in Linux is represented by a thread. A thread can correspond either to a user-space POSIX thread or to a process, or a kernel space thread (and derivatives, such as work queues). From the scheduling viewpoint, these three concepts are indistinguishable inside the kernel scheduler, since they are treated exactly in the same way.

In order to handle the schedule of its threads, the Linux kernel introduces several scheduling classes, three for non real-time applications, namely SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE, and two for real-time targets, exactly as provided by the POSIX standard, SCHED_FIFO, SCHED_RR.

SCHED_NORMAL is the POSIX equivalent of SCHED_OTHER, and is the default class associated to any process. The scheduling of SCHED_NORMAL threads is carried out through the *completely fair*

scheduler (CFS), a scheduler (which replaced the former O(1)-scheduler since Linux 2.6.23) that strives to distribute fairly the CPU using an approach very similar to the fair queuing algorithm used in network packet scheduling [DKS1989]. The fairness of the CFS scheduler (which details are not covered in this thesis, since irrelevant for real-time purposes) can be biased by means of a *nice* level. A common misconception, which is worth clarifying, is that the *nice* level is a concept related exclusively to *non* real-time processes/threads.

Conversely, real-time threads, associated to the SCHED_FIFO and SCHED_RR classes, are handled by a separate scheduler (that in this thesis is referred to as *Linux RT scheduler*), which recalls from many aspects the former O(1) scheduler.

The Linux RT scheduler is organized as a distributed scheduler, arranged in so-called *runqueues.* A runqueue is a data structure that holds, from a logical viewpoint, the information about the tasks enqueued on a processor, grouped by priority levels. Such information, however, is encoded in a redundant fashion (soon explained in its details). The purpose of the redundancy is to minimize the time required for the various operations of the RT scheduler.

Each of the *m* runqueues (`struct rt_rq`) holds, among other things:

- An array of 100 doubly-linked lists (one for each priority level), called the *active* threads array. Each list of the array holds the threads enqueued on that particular processor for that particular priority level.
- A bitmap of 100 bits, where each bit reflects the presence or absence of one or more tasks in the corresponding list of the *active* array.
- An integer value (`highest_prio`) which reflects the priority of the highest-priority thread enqueued in that runqueue (that corresponds to the index of the most significant high bit in the bitmap)

A further global structure (`struct cpupri`) holds summary information about the processors state. In particular it holds:

- A bitmap (*pri_active*) of 100 bits, where each bit reflects the presence of absence of one or more tasks with that priority in any of the *m* runqueues (ideally is the logical *or* of the *m* runqueue bitmaps).
- An array of *m* integer values (*cpu_to_pri*) which reflects the priority of each processor, which can be INVALID; IDLE, NORMAL, RT1...RT99 (INVALID is typically used when the processor disabled).
- An array of 100 bitmaps (*pri_to_cpu*), where each bit reflects, for each priority level, the presence of a CPU with that priority.

The basic idea behind this complex distributed organization is to try to avoid, as much as possible, contention when several processors take scheduling decisions at the same time, reducing the length of global (inter-processor) critical sections.

Multiprocessor scheduling of threads in the RT scheduler, is handled by means of two fundamental operations, called *push* and *pull*, the purpose of which is to re-establish the global priority ordering after threads are inserted and removed into the runqueues, and ensure that, at any time, the *m* threads with the highest priority are executing.

The logic behind the RT scheduler is that any event that causes a modification of the runqueues, subsequently invokes push and/or pull operations to complete the reschedule. For instance, when a thread becomes ready for execution (e.g., because a semaphore on which it was pending is posted), it is optimistically inserted on a runqueue (which is determined by a pre-balancing algorithm). However that runqueue might be running another thread with a higher priority (thus not suitable for being pre-empted), while another runqueue might have been more suitable for the new thread, e.g., because completely idle.

The push operation has the responsibility of (trying to) migrate to other runqueues those threads which are not eligible to run on the current runqueue due to a lower priority. There are several events that require a push operation, for instance: (i) a thread being enqueued on a runqueue which is currently running another higher priority thread; (ii) a thread

being pre-empted on a runqueue due to the schedule of another higher priority thread; (iii) a thread which priority is being raised (through a *sched_setpriority* call), but not enough to overtake the priority of the running thread on that runqueue.

The push operation (which is invoked on a particular runqueue) looks at the highest-priority non-running thread on the runqueue and then considers all the runqueues to find a processor where it can run (i.e. a processor which priority level is lower than the highest priority thread on the current runqueue). If such a processor is found, the thread running on that processor is pre-empted, and the thread is moved to that runqueue.

The global *cpupri* structure allows to make this decision without interfering (read locking) the other runqueues. Critical sections are entered only when the thread needs to be effectively moved across runqueues. While searching for a new runqueue, the push operation looks first to the processor on which the thread last executed, as it is likely to be cache-hot in that location (or a closer one, in NUMA systems). The push operation is repeated until a thread fails to be migrated or there are no more threads to be pushed. Because the algorithm always selects the highest non-running task for pushing, the rationale is that, if a thread cannot be migrated, then the lower priority threads cannot be migrated as well.

The pull operation is symmetrical to the push, and is invoked consequently to those events which cause one or more threads to be removed from a runqueue, in order to determine the next highest priority thread that should be executed on that processor. Such events are, for instance: (i) a thread being not anymore eligible for execution (e.g. blocking on a I/O operation, or pending on a semaphore); (ii) a thread which priority is being lowered.

The pull operation looks at all the status of the other runqueues (using again the *cpupri* structure) and checks whether they have a thread with a higher priority than the target runqueue and if that thread can run on the target runqueue (according to its affinity mask). If so, the thread is queued on the target runqueue. The pull operation may pull more than one task to the target runqueue. The rationale is that if the pull operation selects only one candidate thread to be pulled in the first pass and then performs the actual pull in the second pass, there is a possibility that the selected

highest-priority thread is no longer valid, due to another parallel scheduling operation on another processor. To avoid this race between finding the highest-priority runqueue and having that still be the highest-priority thread on the target runqueue, the pull operation pulls several threads. In the worst case, this might lead to a number of threads which would later get pushed off back to other processors, leading to task bouncing. Task bouncing, however, is known to be a rare occurrence [LJ2009].

3. X-RT: A portable framework for real-time scheduling

3.1. Introduction

This chapter discusses the theoretical foundations, the design and the guiding principles of an open-source cross-platform run-time framework called X-RT, which has been developed as a part of this thesis work [Tuc2012]. The aim of this framework is to provide real-time application designers with high level and platform-independent APIs for handling periodic real-time tasks and support advanced multiprocessor scheduling policies, such as G-EDF. The X-RT framework acts as a scheduling middleware which exploits, for its operations, only the common services offered by mainstream RTOSs and their conventional static priority-driven scheduler. Thus it can be easily ported to most of them (the current implementation supports POSIX-compliant RTOSs), requiring no kernel-level modifications to take advantage of modern multiprocessor scheduling policies.

3.2. Motivations

Motivated by the outstanding possibilities offered by the new generations of multi-core processors, which provide exceptional computational capabilities in highly integrated embedded platforms, on one side, and by the ever increasing computational demands of modern real-time applications, constantly flanked by compelling timeliness requirements, on the other side, this work aims at tackling the multiprocessor real-time scheduling problem from an implementative viewpoint. While these topics have received considerable attention in the scientific literature, especially as regards theoretical aspects such as the study and analysis of efficient scheduling algorithms, more practical considerations aiming at concretely putting such algorithms into operation on current platforms and RTOSs have a lesser extent.

The current software scenario counts a wide variety RTOSs, some, as QNX or the many Linux flavors, endowed with a rich set of drivers and services, others more lightweight and tailored for small scale and hard real-time applications like VxWorks, FreeRTOS and µC/OS-II.

Many of those (such as Linux, VxWorks and QNX Neutrino) are deployable on multiprocessor platforms. Unfortunately, they have been developed without much regard to recent algorithmic advances on multiprocessor real-time scheduling and resource allocation. For example, dynamic global real-time scheduling policies are almost never available, despite the fact that such policies are provably superior to conventional (read static) scheduling policies in many ways [Bar2007, BBMS2010, BCL2009].

When taking a look in detail at the services and interfaces exhibited by this wide variety of RTOSs, a huge gap becomes immediately evident: while offering remarkable extra-functional advantages, such as a formally verified or safety certified software architecture [KEH+2009] or very fast and time-bounded critical sections, when it comes to actual real-time scheduling, most of them provide, surprisingly, a very limited support even for the most common and recurring patterns.

The run-time model they exhibit, in fact, merely consists of a set of straight processes whose execution is transparently carried out using a static priority-driven scheduler, further supported by conventional synchronization mechanisms and control system calls. Even the notion of periodic execution is completely lacking, leaving to the designer the burden of realizing these abstractions using low level mechanisms such as timers and semaphores.

This mismatch between theory and practice cannot be blamed solely on experimentalists. Indeed, in the last few years scores of papers have been written on multiprocessor real-time scheduling algorithms, but working implementations do not exist for many (if not most) of the algorithms that have been recently proposed.

3.1. Related work

Many studies compare different real-time multiprocessor scheduling algorithms by means of measuring the percentage of schedulable task sets among a number of randomly-generated ones, as in [Bak2002, Bak2003a, Bak2005, Bak2006, BB2009]. These approaches often rely on schedulability tests or simulations, and they do not involve real tasks

running on a real system, thus they cannot collect such run-time metrics as the actually experienced tardiness due to platform related issues such as cache misses, context switches, RTOS activity, etc. Often, these overheads are assumed to be known a priori and to be accounted in the initial WCET estimation [CA2009]. However, the scheduling policy itself may strongly impact the WCET, for instance due to frequent task preemptions or inter-processor migrations. In this regard, some of the main theoretical properties of EDF and RM are analyzed in [But2005], but the study refers only to uniprocessor systems.

In the field of WCET analysis, [HP2008, JCR2007, LHPo2009, LDN1997] propose a methodology to bound the cache-related migration delay in multi-cores, while in [CGKS2005, YZ2008] the focus is on devising proper task interference models. On a slightly more practical basis, memory access traces of program executions have been used to feed cache architectural simulators in [MB1991, SA2004], while in [BBA2010a, DCC2007, LDS2007, Tsa2007a] some micro-benchmarks have been run on a Linux system in order to quantify the cache-related context switch delay in some specific scenarios (e.g., because of interrupt processing).

In [CLB+2006] Calandrino et al. studied the behavior of some variants of G-EDF and Pfair; in [BCA2008] Brandenburg et al. explored the scalability of a similar set of algorithms. In [BBA2010] Bastoni et al. concentrated on partitioned, clustered and global EDF on a large multi-core system. In all these works, samples of the various forms of overhead that show up during execution on real hardware are gathered and are then plugged in schedulability analysis tests, making them more accurate. However, the final conclusions about the performance of the various scheduling algorithms are actually influenced by the accuracy of the best known schedulability tests, which are often quite conservative.

Some other studies carried out in-depth analysis on implementative aspects of multiprocessor scheduling policies. A milestone, in this regard, is undoubtedly represented by the LITMUS^{RT} testbed, which has been developed (and is being continuously expanded) by the Real-Time Systems Group at University of North Carolina at Chapel Hill. LITMUS^{RT} is an

extension to the Linux Kernel that allows different multiprocessor scheduling algorithms to be linked as plug-in components. In [BA2009] Anderson et al. exploited LITMUS^{RT} to analyze and compare the performances of several variants of the G-EDF policy implementations, evaluating different data structures and different synchronization techniques for the implementation of the G-EDF policy. In particular their study shown as, when it comes to the handling of the most critical sections, as the task release and ready queues, simple but more efficient approaches, such as handling the release queue on a single processor or using only coarse grained locking for synchronizing accesses to the ready queue, are preferable.

Despite the large number of ongoing works and the maturity reached by the project, however, the goal of LITMUS^{RT}, as stated by the authors in [BBC+2007] is not to create a production runtime platform, rather to provide a stable experimental testbed to rapidly implement, study and evaluate different real-time scheduling policies on multicore platforms.

A slightly different direction, instead, is taken by the works of Faggioli et al. in [FCTS2009, FTC2009]. Recently, they have made available an EDF scheduling policy implementation for the Linux kernel in the form of a new scheduling class. It is called SCHED_DEADLINE and implements EDF scheduling with both hard and soft resource reservation capabilities [BLAC2005, MST1994]. SCHED_DEADLINE implements a variant of the Constant Bandwidth Server (CBS) algorithm [AB1998] achieving temporal isolation among concurrently running tasks.

While the original version of the work did support only partitioned scheduling, recent updates introduces support also for clustered and global EDF. A recent approach has been taken by Lelli et al. in [LFCL2012] extending the original version of SCHED_DEADLINE. Conversely to what happens in the LITMUS^{RT} CE1 implementation of Anderson et al., the implementation of G-EDF relies on a distributed run-queue: each processor maintains private queue and tasks migrations, when required, are achieved by means of a push/pull approach similar to the one employed by the standard Linux RT scheduler.

Both LITMUS^{RT} and SCHED_DEADLINE aim at introducing support for G-EDF within the Linux Kernel. While undoubtedly optimal as regards keeping low overheads for the enforcement of the scheduling decisions, this kind of approach, in our viewpoint, introduces some drawbacks as regards the maintainability and the portability across different ranges of RTOSs and process models. Furthermore, such kernel-level interventions can turn out to be problematic from a legal viewpoint, when using third-party customized or certified kernels (in which cases kernel modifications would void the certification).

In this work, instead, the purpose is to provide an alternative approach to implement global scheduling policies, with particular interest in G-EDF, by means of a cross-platform run-time framework, which exploits the basic priority-driven scheduler made available by every RTOS known. While expecting some obvious performance degradations due to the additional context switches required invoke OS system calls and to interact with the runtime framework, the purpose of this work is to analyze if and how such approach is viable, and how does it compare to the existing kernel-level approaches. A similar approach has been taken by Li et al in [LRSF2004], which, however, deals only with uniprocessor scheduling.

3.2. Software architecture for SMP

Overview

The main challenge of concretely putting into operation an advanced multiprocessor scheduling policy, such as G-EDF, on a conventional SMP RTOS is represented by the limited support offered by the static priority-driven RTOS scheduler, which at any time puts into execution the ready processes with the highest priority number.

The goal of the X-RT framework is to raise the abstraction level perceived by the real-time application designer, and introduce the notions of periodic real-time tasks, characterized by temporal attributes such as periods and deadlines, and global scheduling policies.

The purpose is not only freeing the real-time application developer from the burden of dealing with platform-specific and RTOS-specific details,

but also offering a uniform programming interface which remains unchanged when moving across different RTOSs or different process models. This allows the application developer to just declare the temporal attributes of the real-time application, and lets the framework handling its concrete execution on the target RTOS, hiding the operational details.

Abstracting the platform details and elegantly wrapping RTOS primitives, however, is not sufficient to fill the gap that exists between the simplistic scheduling policy implemented by the RTOS, and the complex dynamic global policies offered by the framework. The bigger contribution of this framework, in fact, is represented by the run-time support offered for the concrete implementation of more sophisticated scheduling policies, though leaving the RTOS kernel untouched.

The foundation of the X-RT run-time operations lays on a metascheduler approach. From a concrete viewpoint, such metascheduler is a conventional high-priority RTOS process that is triggered by certain events (user requests, task completion notifications, timers). Such process properly manipulates the numeric priorities of the other RTOS processes (which wrap the user-provided application real-time tasks), in order to let their overall execution to evolve according to the global scheduling policy chosen.

In this regard, the X-RT framework supports the implementation of different policies by means of a plug-in interface, which decouples the algorithmic aspects of the policy from the interactions between the framework and the RTOS. The current release is bundled with a plug-in implementing the G-EDF policy.

Overall architecture

From the architectural viewpoint, the X-RT framework consists into three major components:

- *A system abstraction layer (SAL)*, which abstracts the interface towards the underlying RTOS and hardware platform, as regards process creation, priority mangling, synchronization and timers.
- *An Application Programming Interface (API)*, which defines, independently of the RTOS and hardware platform, the services and the primitives offered to the real-time application developer that realize the periodic tasks abstraction.
- *The metascheduler*, the core engine that implements the API and interacts with the native operating system through the SAL, concretely executing the scheduling policy.
- *A scheduling policy plug-in,* which implements the core logic for the chosen scheduling policy, interacting exclusively with the metascheduler for the enforcement of scheduling decisions (task activations and pre-emptions).

In order to avoid ambiguities, in this chapter the term *process* is used to identify the run-time abstraction of software execution flow exposed by the operating system, in order to distinguish it from the concept of periodic real-time *tasks* which is exposed by the X-RT framework to the application developer. Depending on the target operating system and software model chosen, a process concretely corresponds to a *pthread* when targeting user-space POSIX threads, a *BSD process* when targeting isolated Unix processes, a *system task* when targeting kernel-space VxWorks' taskLib processes.

In order to carry out the implementation of complex scheduling policies, the X-RT framework requires some simple but stringent features in the target RTOS. As presented in the next pages, it can be noted as the set of requirements perfectly matches the base support offered by the manifold contemporary RTOSs. Figure 11 presents a graphical overview of some possible mappings.

In order to guarantee its proper operation and be able to handle the execution of *n* periodic real-time tasks on a SMP system of *m* processors, the X-RT framework requires the following.

Priority driven scheduler requirements

1. The RTOS must be able to handle the execution of *n*+1 (required for the metascheduler) processes on the *m* processors. The execution of such processes must be carried out according to a strict, yet very simple, priority-driven policy: the *m* ready processes with the highest priorities must be executing on the *m* processors at any time (with the exception of short non-preemptible sections such as interrupt service routines, which are almost unavoidable on any actual operating system).
2. When a high priority process becomes ready for execution and *m* lower priority processes are already executing, the RTOS must pre-empt the one with the lowest priority and immediately yield the execution to the higher priority one.
3. When a process is not anymore eligible for the execution the RTOS must yield to the next equal or lower priority ready process (if any). In case of parities, ties must be broken by means of first-in-first-out (FIFO) ordering.
4. The RTOS must provide a system call, herein generically called `SysSetPriority`, which allows to arbitrarily raise or lower the priority of a given process. Whenever the priority of a process is changed, the RTOS must rearrange the processes execution in order to meet the previous requirements.
5. Only three priority levels in total are required by the X-RT framework for handling the execution of an arbitrary number of tasks, respectively:
    - HIGH: is the priority level that is associated to the metascheduler. Whenever the metascheduler has a pending event to process, it must be able to pre-empt to execution of the other real-time tasks,

in order to promptly enforce the new decisions envisaged by the implemented scheduling policy.

- MEDIUM: is the priority level associated to (at most) *m* of the *n* processes, which wrap the user-defined real-time tasks.
- LOW: is the priority level associated to the (at most) *n-m* remaining processes, when the scheduling policy decides to pre-empt them.

Inter process communication

The RTOS must exhibit a facility for allowing message-based point-to-point inter-process communication. From a functional viewpoint, the requirement consists in an abstract data type, called `sys_mqueue_t` in the SAL, which is equipped with two operations, respectively, `SysMsgSend(queue, msg)` and `SysMsgReceive(queue)`.

The semantic required by X-RT is that, upon a call to `SysMsgReceive`, if no message is available on that queue the calling process is suspended indefinitely until a new message is available, and the execution yields to the next higher priority ready process. However, when a message becomes available, as a result to a `SysMsgSend` being invoked from another process, the process suspended on `SysMsgReceive` must be promptly resumed, compatibly with the priorities of the other ready processes. The analogous blocking semantic is required for the `SysMsgSend` operation.

The design of the X-RT framework guarantees that:

- The length of the message queue is bounded: at most *n* + 1 outstanding message can be present on the queue at any time.
- All the `SysMsgReceive` calls for a given message queue are performed by the same thread (thus no thread-safety is required for the message reception system call).

From a practical viewpoint, these requirements find a very straightforward mapping to *POSIX message queues* on POSIX systems, *SysV message queues* on many Unix systems, *MsgQLib message queues* on VxWorks systems.

Timers

In order to carry out the timekeeping activities required by the framework, the RTOS is requested to provide a retriggerable absolute-counting monotonic timer facility. Two operations are envisaged for such purpose in the SAL in order to start/retrigger a timer and stop it, respectively `SysTimerReset(abs_expiration)` and `SysTimerStop()`.

Only a single timer is required by the X-RT framework for all its timekeeping activities, including the ones of the scheduling policy. The RTOS is requested to handle such timer in a strict monotonic fashion (i.e. the timer must not be affected by time-of-day adjustments) and is expected to call a framework-provided asynchronous handler when the `abs_expiration` time is reached. No assumptions are made by the X-RT framework on the execution context of the timer handler routine, since a simple `SysMsgSend` call is involved in the handler.

From a practical viewpoint, this requirement finds a very straightforward mapping to *CLOCK_MONOTONIC* timers on POSIX systems and the equivalent *TimerLib timers* on VxWorks systems.

As regards the time resolution, there is no requirement directly enforced by the X-RT framework. However, the resolution offered by the RTOS or the underlying hardware directly reflects on the maximum resolution that the framework will exhibit to the end-user (e.g., for periodic tasks periods and deadlines).

The X-RT framework handles its timer queues on its own, requiring a single timer to the RTOS. The reasons of this choice are twofold: on one side, while the availability of a timer is ensured on every platform, it is not legitimate to expect an arbitrary availability of timers on all platforms. In some cases, the RTOS merely reflects the availability of hardware timers offered by underlying hardware (typically a few). Furthermore, even in the cases in which the RTOS handles in software timekeeping of an arbitrary number of timers, relying on the RTOS would imply letting the performances of the framework depend strongly on the implementation details of the RTOS (A more in-depth discussion of this point is postponed to Chapter 4.). While such a dependency is not avoidable in the other

cases (process handling), a performance decoupling can be established for timekeeping.

Therefore, in absence of, or in the case of a very inefficient, timer support of the RTOS, the underlying hardware timer (e.g. HPET of Intel processors) can be directly employed by the SAL to ensure the seamless operation of the framework.

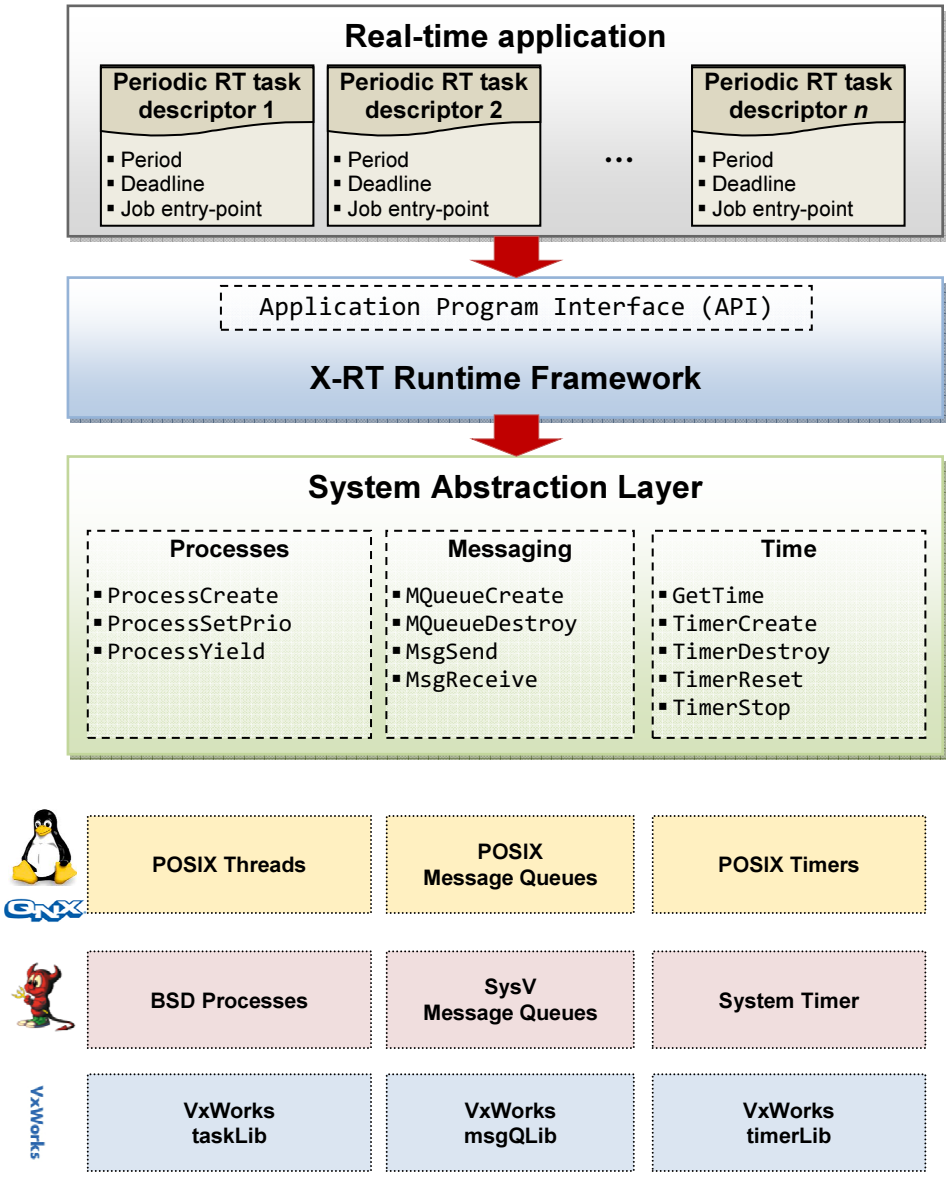


Figure 11: Mapping of SAL operations of the X-RT framework on different RTOSs.

Thread-safety remarks

As a final remark, it might be noted that no locking primitives (e.g., semaphores or mutexes) are required by the X-RT framework. Despite the architecture of a multiprocessor scheduler being intrinsically highly concurrent, the design choices adopted in X-RT have focused on a loosely coupled message-passing architecture.

Concurrency races, which typically arise in shared-memory designs, are avoided by design, modelling the X-RT framework as a set of independent runtime components. Each of them is uniquely responsible of handling its data structures in a thread-local only fashion and interacts with the other components by means of message-passing. The reasons behind this choice are manifold. The experience gained by other research works in this field shows as, in modern cache-coherent multiprocessor architectures, shared memory patterns typically perform poorly. For instance in their study [BA2009] Anderson at al. shown as highly tuned concurrent data structure which take advantage of fine-grained critical sections perform unexpectedly poorly due to cache affinity issues. Similar considerations can be found in [SBas1994].

On the other side, the fast point-to-point network-based interconnects of modern multiprocessor architectures definitely favour inter-processor signalling patterns to cache-coherent memory sharing [MHSM2009]. It is easy to see that with a very simple example: consider two cores *c1*,*c2* of a modern multicore processor accessing (read-write) a simple byte of memory on a shared location, not even necessarily at the same time, in a write-back cache scenario (that is the standard). Assume that *c1* accesses the shared memory location first. Later, when *c2* tries to modify the same byte, the cache coherency protocol will require *c1* to write-back in memory (MESI) or transfer to the second processor (MOESI) the entire stale cache line, thus involving a transfer on the interconnect of 32 bytes (the typical size of a cache line). On the other side, if the software instead employs a message-passing pattern, for instance with *c1* being the responsible of handling the data structure in memory, and P2 just requesting the update through an IPI (inter-processor interrupt), only the bytes of the message (typically a few) need to be transferred through the interconnect, and the

cache line being modified by P1 remains local to P1, thus not needing any write-back or transfer of ownership and reducing the time spent on the interconnect.

Finally, as will be later discussed in 3.5, shared memory sometimes isn't a viable option at all, for instance in the case of AMP systems. In this view, the decoupled architecture of the X-RT framework enhances the portability across different hardware architectures.

Metascheduler design

Upon initialization, the user requests the instantiation of the *n* periodic real-time tasks, through the *CreateNewPeriodicProcess* API method, defined as follows:

```
xrt_task_id_t XRT_CreateNewPeriodicTask(
                  xrt_periodic_task_desc_t *taskDesc);
```

where its unique input argument is the task descriptor, a structure that defines the attributes of the task as follows:

```
typedef struct
{
    char                 name[XRT_TASK_NAME_MAXLEN];
    xrt_rel_time_t       release_period;
    xrt_rel_time_t       initial_phase;
    xrt_rel_time_t       relative_deadline;
    xrt_overrun_policy_t overrun_policy;
    void                 (*job_entrypoint)(void* argument);
    void*                job_argument;
} xrt_periodic_task_desc_t;
```

Apart the temporal attributes of the task, which are self-explicative, the user has to provide a *job_entrypoint*, a function pointer to the task body, which is the routine that will be periodically executed by the framework, as specified by the *release_period* field. Together with that, the user can specify also an optional argument that will be passed back upon each job invocation, for instance an identifier to distinguish the specific task instance if several tasks share the same entry-point.

From the runtime viewpoint, for each periodic task requested by the user, the framework instantiates a dedicated RTOS process. The entry-point of such process, however, doesn't directly point to the user-provided

`job_entrypoint` (which is a straight function with no notion of cyclic execution). Rather, another module of the framework, called *task shell*, which resides in the same process space of the real-time task, acts as entry-point for the process.

Concretely, each task-shell is a cyclic event-processing loop, which coordinates the execution of the wrapped task with the metascheduler. The communication between each task-shell and the metascheduler is handled uniquely by means of message passing, exploiting the messaging abstraction provided by the SAL. For such purpose, each task shell has an ingress message queue for receiving the following messages from the metascheduler:

- *MSG_RELEASE_JOB*: triggers the release of a new job, invoking the invocation of the `job_entrypoint` function.
- *MSG_TERMINATE_TASK*: causes the termination of the process associated to the real-time task.

On the other way, the task-shell transmits back the following messages to the metascheduler (which has its own message queue too):

- *MSG_TASK_INITIALIZED*: sent once upon task creation, notifies that the task has completed its initialization phase.
- *MSG_JOB_COMPLETED*: sent every time a job execution completes (i.e. the `job_entrypoint` function returns).
- *MSG_TASK_TERMINATED*: sent once, after receiving the *MSG_TERMINATE_TASK*, to notify that the task has completed its clean-up phase.

Figure 12 depicts the interaction, the run-time organization and the message exchange between the task-shells and the metascheduler.

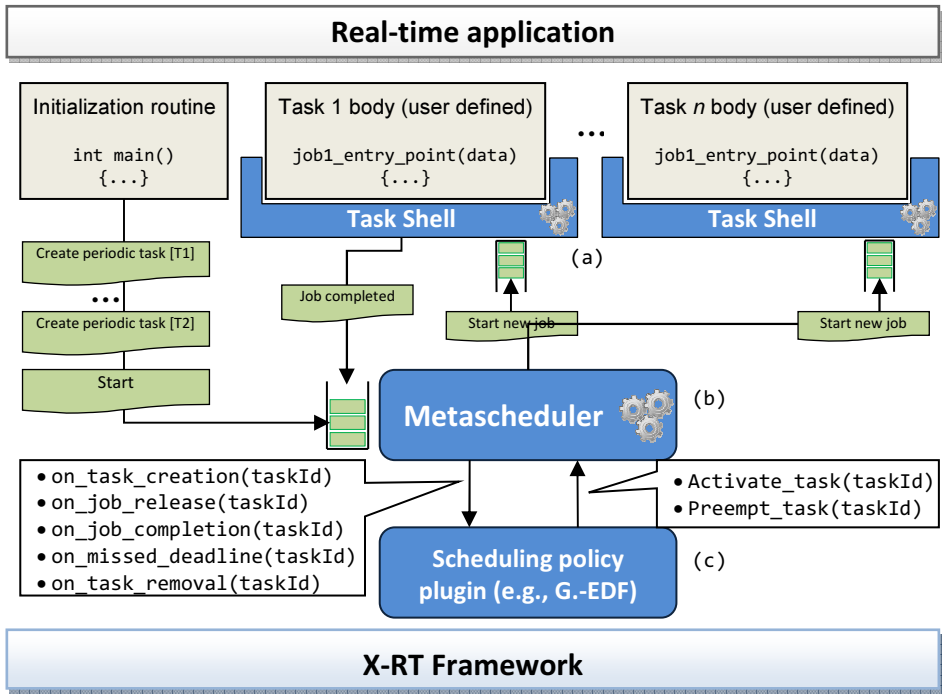


Figure 12: Overview of the interaction between the X-RT framework components: (a) task-shell(s); (b) Metascheduler; (c) Scheduling policy plug-in.

When the execution of a job completes, the task-shell re-enters its message loop, waiting for new messages. From the RTOS scheduler viewpoint, therefore, the process associated to the real-time task is suspended after each job execution, until a new message comes from the metascheduler. The operation of the task-shell is summarized, in its essential, in the code listing below.

```
1.  void TaskShell(const xrt_task_t* task)
2.  {
3.    bool exit = false;
4.
5.    /* Initialization code, omitted for sake of brevity. */
6.
7.    while (not exit)
8.    {
9.      SysMsgReceive(& rx_queue, & rx_message);
10.
11.     switch (rx_message.id)
12.     {
13.     case MSG_RELEASE_JOB:
14.       task->job_entrypoint(task->job_argument);
15.       SysMsgSend(& tx_queue, MSG_JOB_COMPLETED, task_id);
```

```
16.       break;
17.
18.     case MSG_TERMINATE_TASK:
19.       exit = true;
20.       break;
21.     }
22.   }
23.
24.   /* Cleanup code, omitted for sake of brevity. */
25.   SysMsgSend(& tx_queue, MSG_TASK_TERMINATED, task_id);
26. }
```

It might be noted that the task-shell is only responsible for the local execution of the task jobs, but is completely unaware about the current process priority and how to mangle it.

Such operation, instead, is carried on by the metascheduler, which is the core module of the X-RT scheduling framework. From the metascheduler viewpoint, each periodic task can be in one of the following states (Figure 13):

- *CREATED*: state associated to newly created tasks, which task-shell has not yet completed the initialization phase.
- *IDLE*: the initialization phase has completed and the last job execution has completed. The task-shell is suspended on its message queue, waiting for the metascheduler to trigger the execution of a new job (or issuing termination of the task).
- *READY*: state associated to tasks which release timer has expired but that have not yet initiated the execution of the job (e.g. due to the presence of some other higher priority tasks). This state is entered after the metascheduler releases the task (sending a MSG_RELEASE_JOB to the corresponding task-shell) and persists until the metascheduler is requested (by the scheduling policy plug-in) to run the task.
- *RUNNING*: state associated to tasks that are currently running a job on one of the *m* processors. This state is entered when the scheduling policy plug-in requests a task activation to the metascheduler.

- *PREEMPTED*: state associated to tasks previously RUNNING, for which the scheduling policy plug-in requested a pre-emption, in order to make room for a higher priority task. From the scheduling viewpoint a PREEMPTED task is analogous to a READY task, with the only exception that a PREEMPTED task have already executed a part of its job (this aspect will have a fundamental importance later for restricted-migration policies).

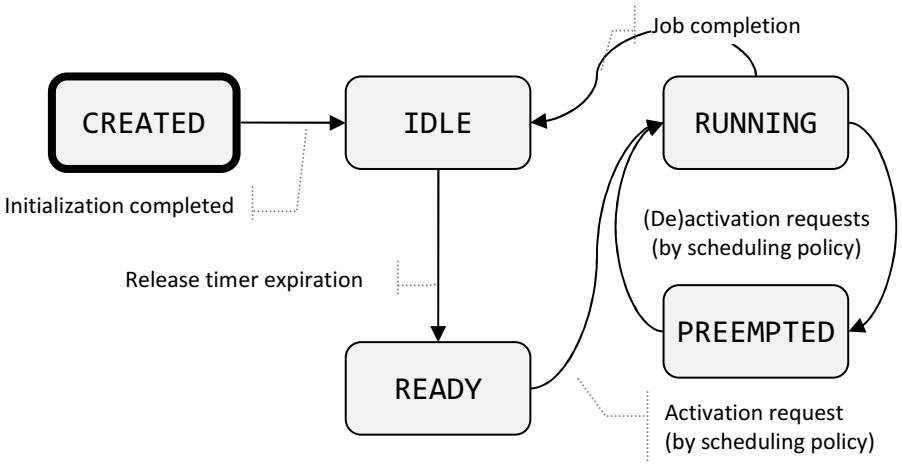


Figure 13: States of the real-time tasks handled by the X-RT metascheduler.

The state of a task, as described so far, is only an internal representation of the metascheduler, which is not directly perceived by the RTOS. The mapping between the state of a task in the metascheduler and the state of the corresponding process in the RTOS scheduler is established by the metascheduler, as follows. An IDLE task has its corresponding process suspended in the task-shell waiting for a message (line 9). When its release timer expires, the metascheduler simply updates its state to READY, without performing any further action on the RTOS process. Furthermore, the `OnJobRelease` method is invoked on the scheduling policy plug-in, in order to notify it about this new event. Upon this notification, the scheduling policy plug-in has two options: activate the task (issuing a `MetaschedulerActivateTask` call back on the metascheduler) or keep it deactivated, in the case that other *m* tasks are already running and the scheduling policy considers them more prioritary than the newly released

task. The scheduling policy is assumed to be work-conserving (i.e. non-idling).

In the former case, the `MetaschedulerActivateTask` call causes the metascheduler to raise the priority of the task's process to MEDIUM and send the MSG_RELEASE_JOB message to the task-shell. The RTOS scheduler, at this point, has no other option[4] than moving the process to one of its *m* running queues and carrying out the execution of the task-shell that will in turn start the execution of the user-provided job entry-point (line 14). In the latter case, instead, the process will simply remain suspended waiting for the release message (line 9), which will occur when, in a next event, the scheduling policy plug-in will finally decide to activate the task.

When a new task is released, the scheduling policy plug-in can decide to pre-empt another RUNNING task in order to respect the metascheduler invariant (keep at most *m* RUNNING tasks), issuing a `MetaschedulerPreemptTask` call. In this case, the metascheduler reacts lowering the priority of the pre-empted task's process to LOW through the `SysProcessSetPriority` primitive of the SAL. At the end of the new task activation + task pre-emption sequence, the RTOS scheduler will find again *m* processes with MEDIUM priority, thus making them reflect the *m* RUNNING tasks of the metascheduler.

When a job execution completes, the execution flow of the task's process returns to the task-shell (line 15), which will simply notify the event to the metascheduler through the MSG_JOB_COMPLETED message and self-suspend waiting for a new metascheduler message. Correspondingly, as the metascheduler receives the completion message, the state of the task is changed to IDLE and the scheduling policy plug-in is notified about the event through the `OnJobCompletion` method.

At this point, if there are any READY or PREEMPTED tasks, the scheduling policy plug-in must pick and activate one of them, in order to

[4] In the case the RTOS schedules the task's process on the same processor where the metascheduler is currently running, the task-shell execution will continue as soon as the metascheduler (which has a HIGH priority) completes the handling of the release queue and suspends itself again waiting for a new message.

keep the RTOS scheduled fed with *m* running processes. Conversely, if no any other task is being activated, the RTOS can either idle that processor or grant the execution to other non real-time processes.

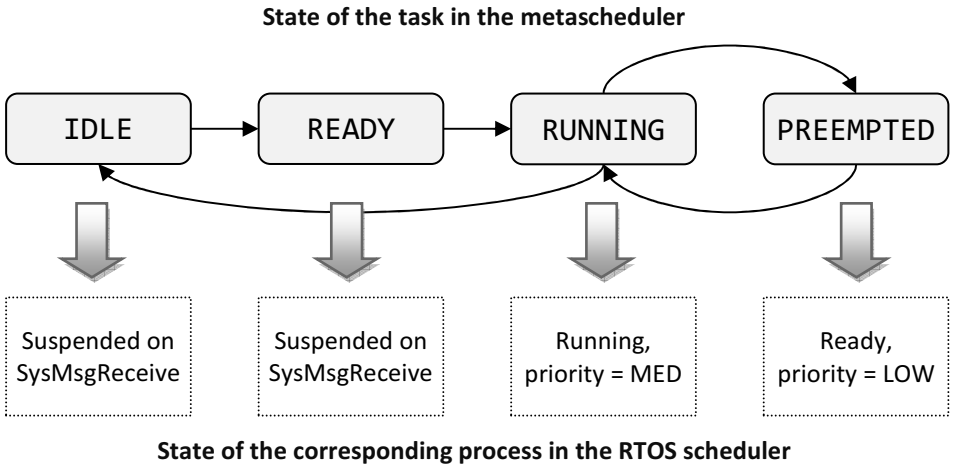


Figure 14: Mapping of metascheduler task states to RTOS processes states.

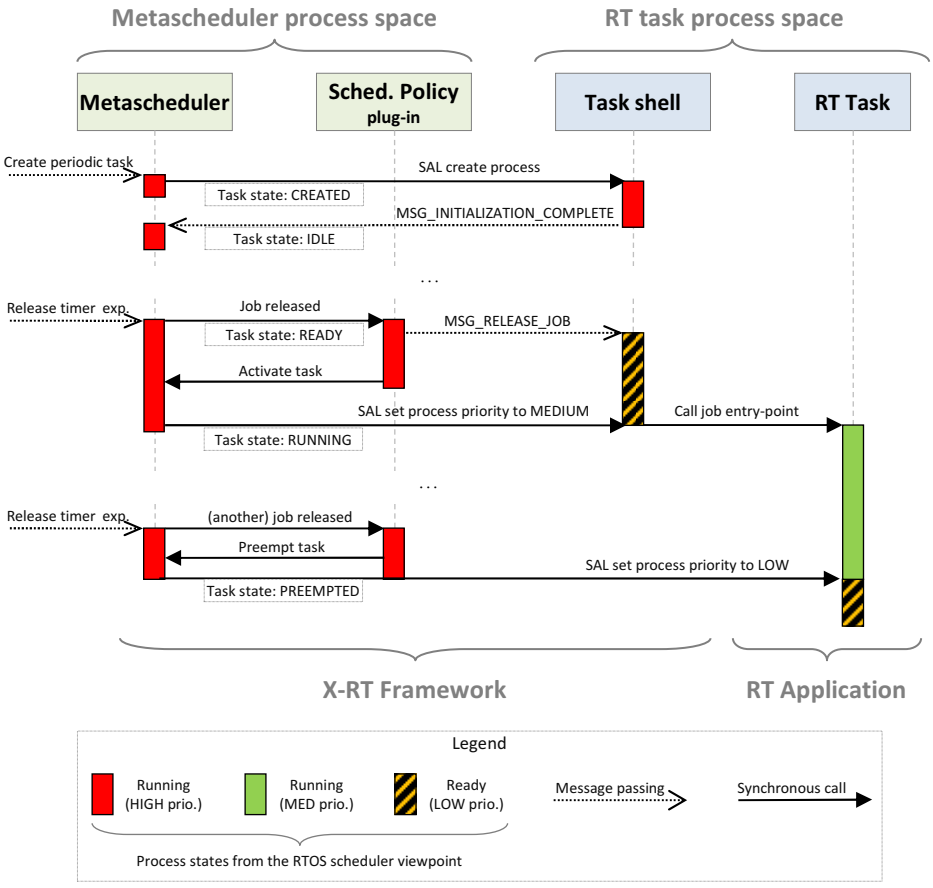


Figure 15: Interaction diagram of the metascheduler components and the RTOS.

Timekeeping

Timekeeping in the X-RT framework is organized in timer queues. Each timer queue (*xrt_timer_queue_t*) is a priority queue of timer objects (*xrt_timer_t*), which use the absolute expiration time as priority key.

Each timer queue is concretely implemented as an addressable binary heap (ABH), a novel tree-based implementation of the binary heap data structure designed ad-hoc for timekeeping in embedded real-time systems, which brings together the performances of binary heaps (all the insert and removal operations of the ABH have logarithmic worst-case complexity), the flexibility of a pointer based tree structure, and the determinism of a embedded-anchor model, which doesn't require any dynamic memory management. (A more in-depth discussion about these topics, together with the presentation of the ABH data structure are deferred to chapter 4.).

All the components of the X-RT framework, which require timers for their operation, take advantage of one or more X-RT timer queues. For instance, the metascheduler employs a timer queue for handling the release of periodic tasks (*release queue*) and one for monitoring their deadlines (when deadlines are not implicit).

The interface exposed for using X-RT timers is the following.

```
1. void TimerQueueCreate
   (
       xrt_timer_queue_t*   timer_queue,
       xrt_timer_callback_t callback
   );

2. void TimerStart
   (
       xrt_timer_queue_t* timer_queue,
       xrt_timer_t*       timer,
       xrt_abs_time_t     expiration
   );

3. void TimerStop
   (
       xrt_timer_queue_t* timer_queue,
       xrt_timer_t* timer
   );
```

It might be questionable why the X-RT timekeeping has been organized in multiple queues, instead of keeping all the timer objects in a single queue,

considering that a single RTOS timer is going to be used at the end. The reasons behind this choice are several. On one side the decoupling in timer queues allows to reduce the timekeeping overhead for different class of timers characterized by different update rates: if a small set of timers that is updated more frequently (e.g., the running queue of a scheduling policy plug-in) and other sets of timers are updated less frequently (e.g., the metascheduler release queue), the overhead for ABH insertion/removal is bounded only to the cardinality of the involved timer queue. Secondly, dividing timers in timer queues allows to handle prioritization, for instance giving more priority to the timer queues of the metascheduler, which are critical for taking system-wide scheduling decisions, and lower priority for timers requested by the end-user for its own application purposes.

From the RTOS interaction viewpoint, timer queues as handled follows. At any time the only RTOS timer, abstracted by the SAL, is triggered to timer expiring soonest, that is the highest priority element among all the timer queues head (Figure 16). When a new timer is started (stopped) its corresponding timer object is inserted into (removed from) the given (corresponding) queue. This operation has a O(log(*n*)) worst-case complexity (with *n* being the number of active timers in the timer queue) due to the ABH implementation. After the insertion (removal), the head of the queue, that is the timer with soonest expiration in that queue, is compared against its old value. If the head has not changed, no further action is required, since, per definition of min-queue, it implies that the RTOS timer is already triggered to the soonest expiration time.

If the queue head has changed, instead, the RTOS timer might need to be retriggered: in the case of an insertion, the new timer might expire sooner than all the other timer present, thus the RTOS timer must be anticipated to match the new (closer) expiration time; in the case of a removal, the removed timer might be the soonest one, thus the RTOS timer must be delayed to match the soonest among the remaining timers, if any.

In order to keep the binding between the timer queue heads and the RTOS timer, another priority queue, the *root timer queue*, is employed in a hierarchical fashion. The nodes of the root timer queue are represented by

the heads of the registered timer queues, and its head corresponds exactly to the expiration time of the RTOS timer.

Thus, whenever the start (stop) of a timer leads to a change of the corresponding queue's head, the priority of corresponding node in the root timer queue must be increased (decreased) accordingly. If such operation, in turn, reflects in a change of the root timer queue's head, the RTOS timer is retriggered. Since a priority increase/decrease operation has still a logarithmic worst-case complexity, the overall worst-case complexity for handling m timer queues in the X-RT framework is $O(\log(m)) + O(\log(n))$ (with n being the length of the largest timer queue, typically larger than the number of timer queues).

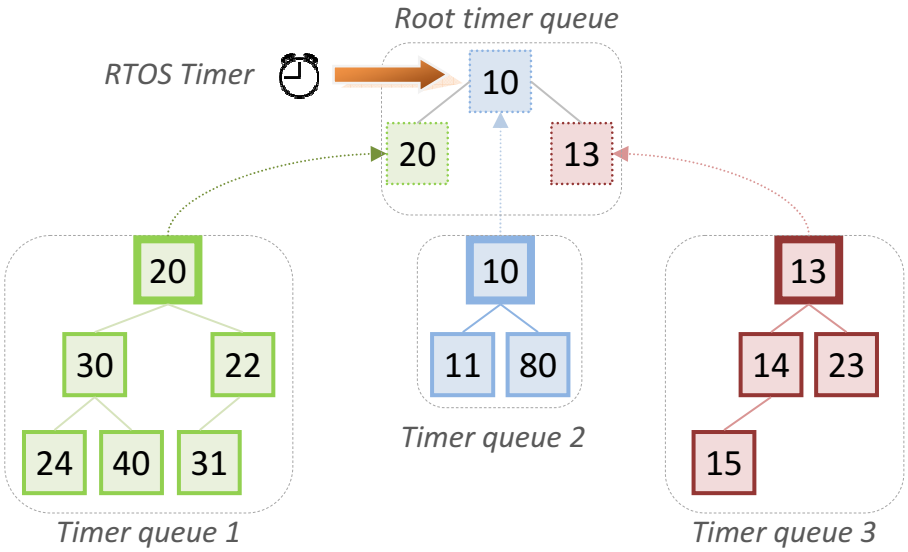


Figure 16: Timekeeping in the X-RT Framework.

3.3. Implementation of the G-EDF scheduling policy

Now that the overall architecture of the X-RT framework and the interaction mechanisms between the metascheduler and the RTOS have been illustrated, the design of the G-EDF scheduling policy plug-in is finally presented.

The G-EDF policy provides that, at any time, the *m* tasks with the closest (absolute) deadline shall be executing on the *m* processors. This apparently straightforward requirement has, however, complex implications as it is applied to an event-driven scenario like the one of the X-RT scheduler, in order to ensure the respect of the policy in every condition.

In the X-RT implementation, the G-EDF plug-in uses two data structures to keep track of running and ready tasks, respectively, a *running queue* (*Rq*) and a *ready queue* (*rq*). Both queues take advantage again of the ABH data structure previously employed for timers.

The running queue is organized as a max-heap (the head represents the task with the furthest deadline), and contains only the (at most) *m* RUNNING tasks currently expected to be running on the processors.

The ready queue is organized, instead, as a min-heap (the head represents the task with the closest deadline) and contains the (at most) $n - m$ released task (either in the READY or PREEMPTED states).

Tasks are inserted using their absolute deadline as the priority key in both queues, respecting the following invariants:

1. $Rq \cup rq \equiv \{t \in \mathrm{T} \mid state(t) \neq \mathrm{IDLE}\}$

   Both the ready and running queues contain only non-IDLE tasks.

2. $Rq \cap rq \equiv \emptyset$

   The ready and running queues are disjoint.

3. $\forall\, t_i \in Rq \;\; \nexists \;\; t_j \in rq \mid d_j < d_i$

   Any task in the running queue has a closer absolute deadline than the tasks in the ready queue.

Release of a new job

When a new job is released (i.e. the metascheduler calls the `OnJobRelease` method on the G-EDF plug-in), the plug-in verifies whether the task should be activated or not, as follows.

The trivial case is represented by the running queue containing less than m tasks. In this case the newly released task just need to be inserted in the running queue (that is a O(log(m)) operation), and activated through the `MetaschedulerActivateTask` call.

Conversely, if the running queue is full, the plug-in must check whether its deadline is closest than at least one of the m other running tasks. With the running queue organized as a max-queue, this translates into a simple O(1) operation, involving just a comparison with the running queue head, which represents the less prioritary running task.

If the new task has a further deadline, it is inserted into the ready queue in O(log(n-m)) and no action is requested to the metascheduler, which will keep the task into the READY state. If the task has a closer deadline than the running queue's head, the corresponding task is pre-empted, issuing a `MetaschedulerPreemptTask`, it is moved to the ready queue (in O(log(n-m))) and the newly released task is inserted in the running queue (O(log (m))) and activated.

Completion of a job

When a job completes its execution (i.e. the metascheduler calls the `OnJobCompletion` method on the G-EDF plug-in), the corresponding task is removed from the running queue. The end of its execution makes room for the execution of another (READY or PREEMPTED) task, which is the task (if any) in the ready queue with the closest deadline.

Since the ready queue is modelled as a min-heap, this operation concretely translates in a `ABHRemoveHighest` operation, which requires O(log(n-m)) time. This is the time required to remove the node and rearrange the ready queue, perform an insertion into the running queue and a `MetaschedulerActivateTask` call, in order to give back MEDIUM

priority to the task's process (and eventually unblock the task-shell if the task was in the READY state) and carry on its execution.

In the case of a job overrun, that is, a job which does not complete by the next job's release, two different overrun reaction strategies are available and can be selected by the user at the moment of task creation: *ASAP* (as soon as possible) and *SKIP*. The former provides that the next job is released (thus is made eligible for execution) as soon as the current overrunning job completes. This allows reducing the impact of short and temporary overloading events (e.g., I/O errors) on the schedule and recovering the nominal execution as soon as possible.

However, this kind of strategy is known for causing avalanche effects on applications catheterized by very high utilization factors close to the schedulability bounds. For such reasons an alternative SKIP policy has been envisaged. In the case of a task overrun, such policy forces the task to skip a number of successive jobs equal to the length of the overrun. This latter policy tends to introduce scheduling fairness, penalizing overrunning tasks by means of job inhibition and giving back CPU time to the other tasks, in order to compensate the scheduling pressure generated by the overrun condition.

Final remarks

As a final remark, it might be worth noting as the plug-in implementation doesn't deal at all with processor assignments. The reason behind this choice is mostly related to the metascheduler design of the X-RT framework. The metascheduler, in fact, does not replace the native RTOS scheduler, rather acts as a frontend for it, letting the RTOS handle the hardware-related context switching and migration operations.

RTOS schedulers, in fact, already implement such logic for handling process-to-CPU assignment and their migrations, since it is a mandatory requirement also for the operation of the simpler priority-driven native scheduler. For instance, the recent releases of the Linux kernel are endowed with fine-grained control logic, which takes into account the multiprocessor topology (for dealing with hyper-threading processors and

NUMA systems) and the cache-affinity of processes when it comes to make decisions about inter-processor task migrations.

Some RTOSs give the possibility to override such behaviour by means of per-process affinity masks, which force the scheduler to execute a given process on a specified processor. However, the decision made in this work is not to take advantage of such mechanisms (i.e. leave the affinity masks filled), enforcing only the priorities of the processes through the X-RT framework and leaving the degree of freedom of the processor assignment to the RTOS.

3.4. SMP experimental evaluations.

In order to evaluate the validity of the X-RT metascheduler approach and the G-EDF plug-in, on SMP platforms, two types of evaluations have been conducted: runtime overheads measurements and schedulability tests.

The system used for the experimentations is an eight-thread Intel Core i7-920 64-bit processor. Each core is endowed with 64k L1 cache, 1MB of L2 cache and 8MB of shared L3 cache. The operating system chosen for the experimentation is Linux x86_64 kernel ver. 3.6.6, in tick-less configuration (CONFIG_NO_HZ = y).

In order to get comparable results with other kernel-space approaches discussed in other cited works, the evaluation methodology illustrated in the next sections is strongly inspired by the one used in [BA2009].

Runtime overheads

The first set of evaluations is represented by overhead measurements and is aimed at identifying which is the overhead introduced by the X-RT framework, in terms of CPU time taken by the metascheduler, the G-EDF plug-in and the underlying RTOS kernel for carrying out all the operations envisaged by the framework. Such overhead depends on three major factors, which have been accounted separately (Figure 17 graphically illustrates them):

1. *Release-queue overhead*: is the time spent to process the release queue, release the expired tasks, reinsert them into the release

queue for their next period and retrigger the RTOS timer, every time the (unique) RTOS timer expires.

2. *Job activation overhead:* is the time spent by the G-EDF plug-in when the release of a new task is notified, plus the time spent by the consequent metascheduler invocations to pre-empt and activate the newly released tasks, plus the time consequently required by the RTOS kernel to alter the priorities of the processes and perform the corresponding context switches.

3. *Job completion overhead:* symmetrically occurs when a job completes and the event is notified to the metascheduler (and in turn to the G-EDF plug-in). This overhead accounts also the time required by the G-EDF plug-in to eventually select and activate the next task, and the corresponding RTOS context-switches, when ready queue is not empty.

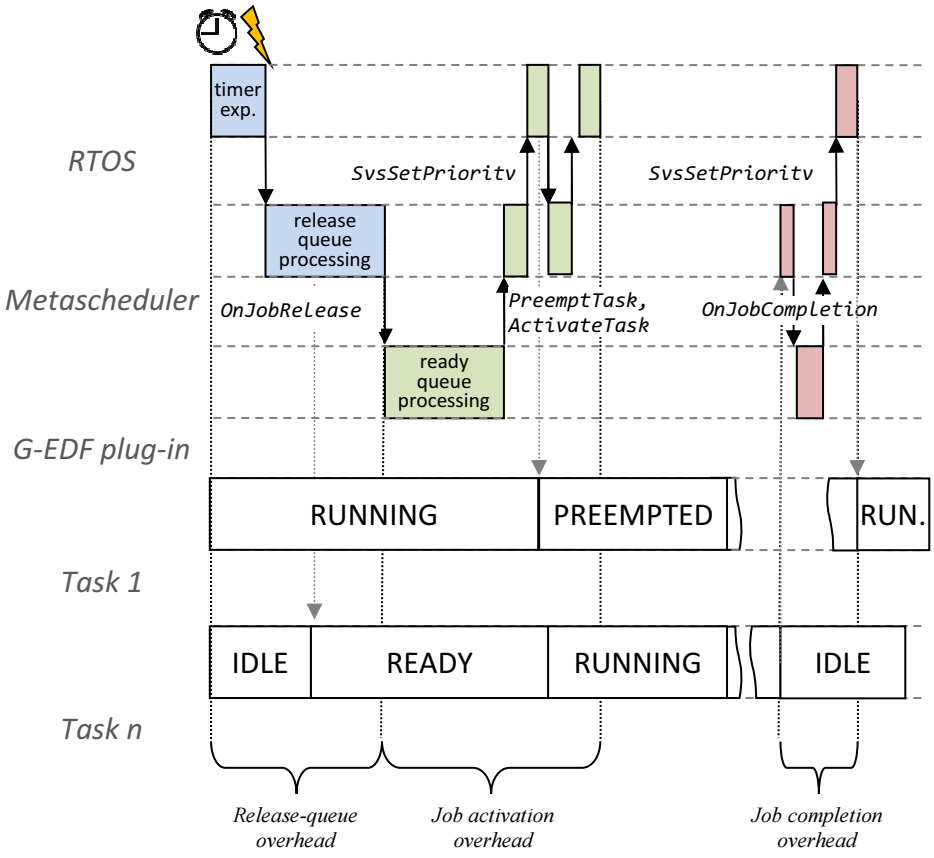


Figure 17: Factors that contribute to the X-RT framework overhead.

The measurements have been conducted using task-sets of variable cardinality, from 50 to 450 in steps of 50. For each step, ten different task-sets have been randomly generated, with periods uniformly distributed in the 10-100 ms. range and keeping an almost constant utilization factor of 3.2 (~ 40% total CPU time). Each of those 90 task-sets have been executed for 30 seconds, and the corresponding average values (outliers filtered out using 98^{th} percentile) are shown in Figure 18-Figure 20.

Figure 18 shows the release-queue overhead. Such overhead is mostly due to the processing of the timer queues and the system call to retrigger the RTOS timer.

Figure 19 shows the job activation overhead. Two main factors contribute to this overhead: the metascheduler + G-EDF plug-in computation and the RTOS system call invocations, for raising the priority of the activated task, send a message to the corresponding task-shell and, in the case a pre-emption is required, lower the priority of the pre-empted task. It might be noted as the major contribution is due to the latter factor, where the Linux system calls impact with an almost fixed cost of 1.5 microseconds. The remaining metascheduler contribution, which gives the logarithmic trend to the overhead curve, is due to the processing of two ABH queues employed by the G-EDF policy plug-in for tracking, respectively, the running and the ready tasks.

Figure 20 finally shows the job completion overhead. Similarly to the previous case, job completion involves up to three RTOS system calls (one for sending the completion message to the metascheduler, one for lowering the priority of the process and an optional third one for increasing the priority of a previously pre-empted process, if any) and the corresponding processing of the two G-EDF queues. In this case, however, the fixed cost due to the RTOS reveals to be slightly higher. A possible explanation for this higher overhead is the concurrent use of the POSIX message queues. In fact, while the job activation message sending is one-to-many (job activation messages are always sent from the metascheduler, which is a single thread, to the task shells), during job completion the messages are sent on the reverse path in a many-to-one fashion. Thus, depending on the implementation details of the POSIX message queues, this is very likely to

cause either bouncing of the cache-lines that hold the message queues' data, and synchronization in the cases in which multiple jobs complete simultaneously on different processors.

In general, the overhead measurements show very encouraging results. In fact, when considering the cumulative effect of this overhead during an entire task-set execution, the corresponding metascheduler overhead ratio, which has been calculated as the ratio between the total CPU time of the metascheduler thread and the total CPU time of the entire process (n + 1 threads) ranges between 0.5% (N=50 tasks) and 1.9% (N=450 tasks).

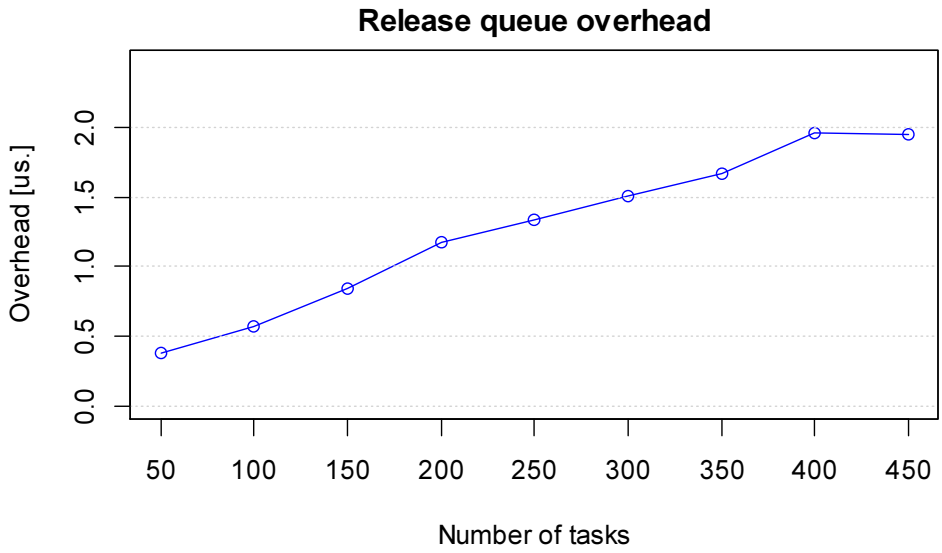


Figure 18: Release queue overhead (average).

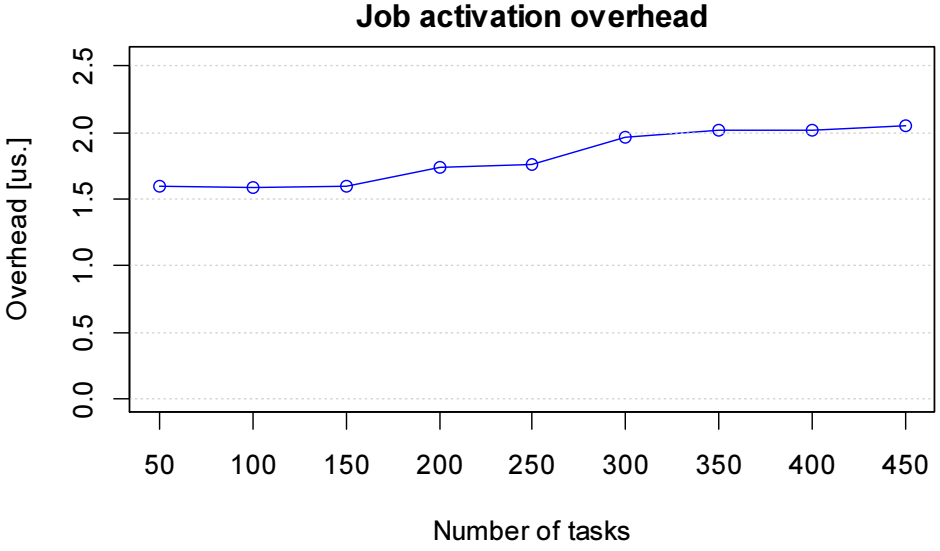


Figure 19: Job activation overhead (average).

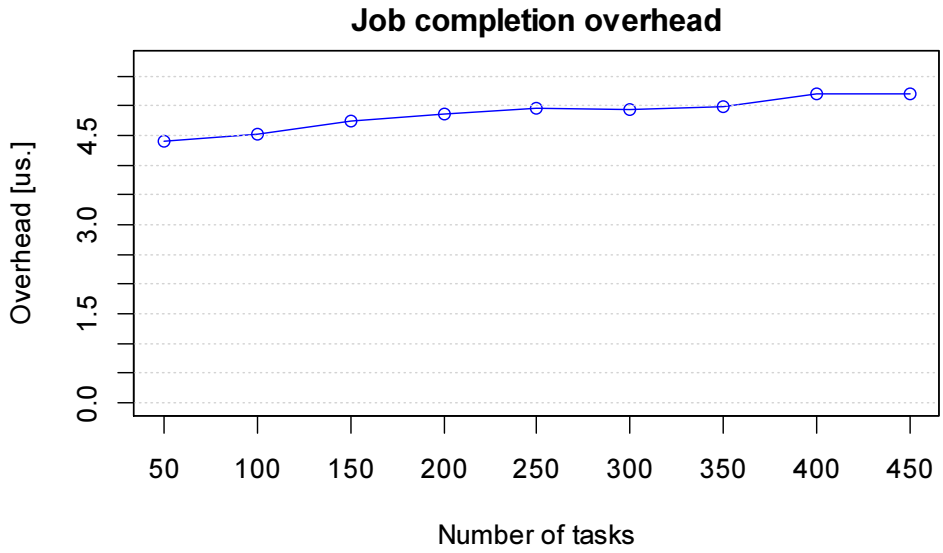


Figure 20: Job completion overhead (average)

Schedulability tests

As regards the overall schedulability, random task-sets have been generated using three period and six utilization distributions, for a total of 18 scenarios. As regards the former, periods were generated according to three uniform distributions in the ranges [3 ms, 33 ms] (*short*), [10 ms, 100 ms] (*moderate*) and [50ms, 250ms] (*long*). As regards the latter, utilization factors were generated using three uniform distributions in the ranges [0.001, 0.1] (light), [0.1, 0.4] (medium) and [0.5, 0.9] (heavy) and three bimodal distributions, of either [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy).

Each task-set was created by generating tasks until a cap on the total utilization factor, which varies between 0.5 and 8 with a step of 0.5, was reached and then discarding the last-added task.

Sampling points were chosen such that sampling density is higher (100 samples) in areas where curves change rapidly, and lower in the other areas (20 samples). Each task-set has been executed for 60 seconds.

For each of the 18 scenarios, three curves have been plotted on the y axis (ratio of schedulable task-sets) as a function of the utilization cap on the x axis (1.0 = 100% CPU time on one processor):

- *HRT schedulability*: each task-set is considered HRT schedulable if, during the experiment, no task misses any deadline. The ratio of HRT schedule task-sets is represented in the plot by a blue dashed line, labelled "*HARD"*.
- *SRT schedulability*: each task-set is considered SRT schedulable during the evaluations if the maximum tardiness of the tasks is less or equal than their period (i.e. if the tasks that miss their deadline complete their execution by the end of their 2nd period). The ratio of SRT schedulable task-sets is represented in the plots by a purple dotted line, labelled "*SOFT*".
- *Theoretical schedulability*: each task-set is tested (offline) for HRT schedulability using four known sufficient (but not necessary) schedulability tests for G-EDF [Bak2003a, BCA2008, BCL2005, GFB2003] and deemed schedulable if it passed at least one of these four tests. The ratio of theoretically schedulable task-sets is represented in the plot by a red straight line, labelled "*SCHED"*.

Results

In the case of uniform light distributions (Figure 21a, Figure 22a, Figure 23a) the frameworks exhibits a very good behaviour as both the HRT and SRT schedulability curves are perfectly overlapped with the theoretical trend of the schedulability tests.

The situation becomes even more interesting when moving to uniform medium distributions (Figure 21b, Figure 22b, Figure 23b). In all the three cases, in fact, the sufficient nature of the schedulability tests emerges in a evident way. While the schedulability tests drop down between the range [5.5; 6.5], the actual HRT schedulability is still held until the utilization factor of 7.3, highlighting a pessimistic behaviour of the four schedulability tests in the case of medium uniform distributions and a still good behaviour of the G-EDF implementation and the overall X-RT framework (a utilization factor of 7.3 on a 8-thread system corresponds to a normalized CPU usage of 91.25%).

A degradation of the scheduling performances, instead, can be noted in the case of uniform heavy distributions (Figure 21c, Figure 22c, Figure 23c) or

bimodal ones (Figure 24, Figure 25, Figure 26) though at a lesser extent. In particular, in the case of short periods, the HRT schedulability is worse than the theoretical lower bounds of the schedulability tests. However, a very good behaviour is still observed for moderate and long periods, where the curve of the HRT schedulability overlaps again with the theoretical bounds of the schedulability tests.

As regards SRT schedulability it is worth noting that the adoption of the SKIP policy, for avoiding system overload in cases of missed deadlines, exhibits the most benefit in the case of heavy utilization and longer periods. In such cases in fact, the penalty interdiction period imposed by the SKIP policy to the heavy overrunning tasks gives back a substantial CPU time to the remaining tasks, which allows them to recover from the overload within a single period. As expected, instead, such effect is less evident when the overrunning tasks are short and highly fragmented, because the contribution to the overload of the system is more distributed.

In general the framework and the G-EDF implementation provided very satisfactory results. The trend for SRT schedulability is very close to what envisaged by other studies on G-EDF, especially due to improvements brought on SRT scenarios by the SKIP policy.

In this regard, Figure 27…Figure 32 show comparative schedulability tests of X-RT (thick blue line) and LITMUS^{RT} (thin purple line) for HRT (solid lines) and SRT (dashed lines) performed on the same machine using six different utilization distributions (uniform/bimodal light, medium and heavy) with periods distributed uniformly in the [10..100] ms. range.

Furthermore, the lockless message-driven multiprocessor architecture proved to cope extremely well with the network-based QPI architecture of the modern Intel processors, as the one used in the experimental evaluations. In particular, the schedulability trends on task-sets with medium uniform utilization are notably higher when compared to the corresponding results obtained in [BA2009], which was employing a crossbar-based SPARC multicore processor (Sun Niagara).

It has to been said that, in the current set of experiments, the generated tasks are simulating pure CPU-bound load, not performing any memory read/write transaction. In this regard, it would be interesting to carry out

more detailed investigations on the behaviour of the overall system, with task performing actual memory access. Such an experimentation would reveal more interesting details about the process-to-processor mapping and migration strategies employed by the underlying RTOS, highlighting the magnitude of the bottlenecks which would unavoidable come in when cache line bouncing effects are involved.

In this work cache-related effects have been avoided, in order to have an evaluation methodology comparable with the other works in the field, assessing the general viability of the novel multiprocessor synchronization approach based on message-exchange.

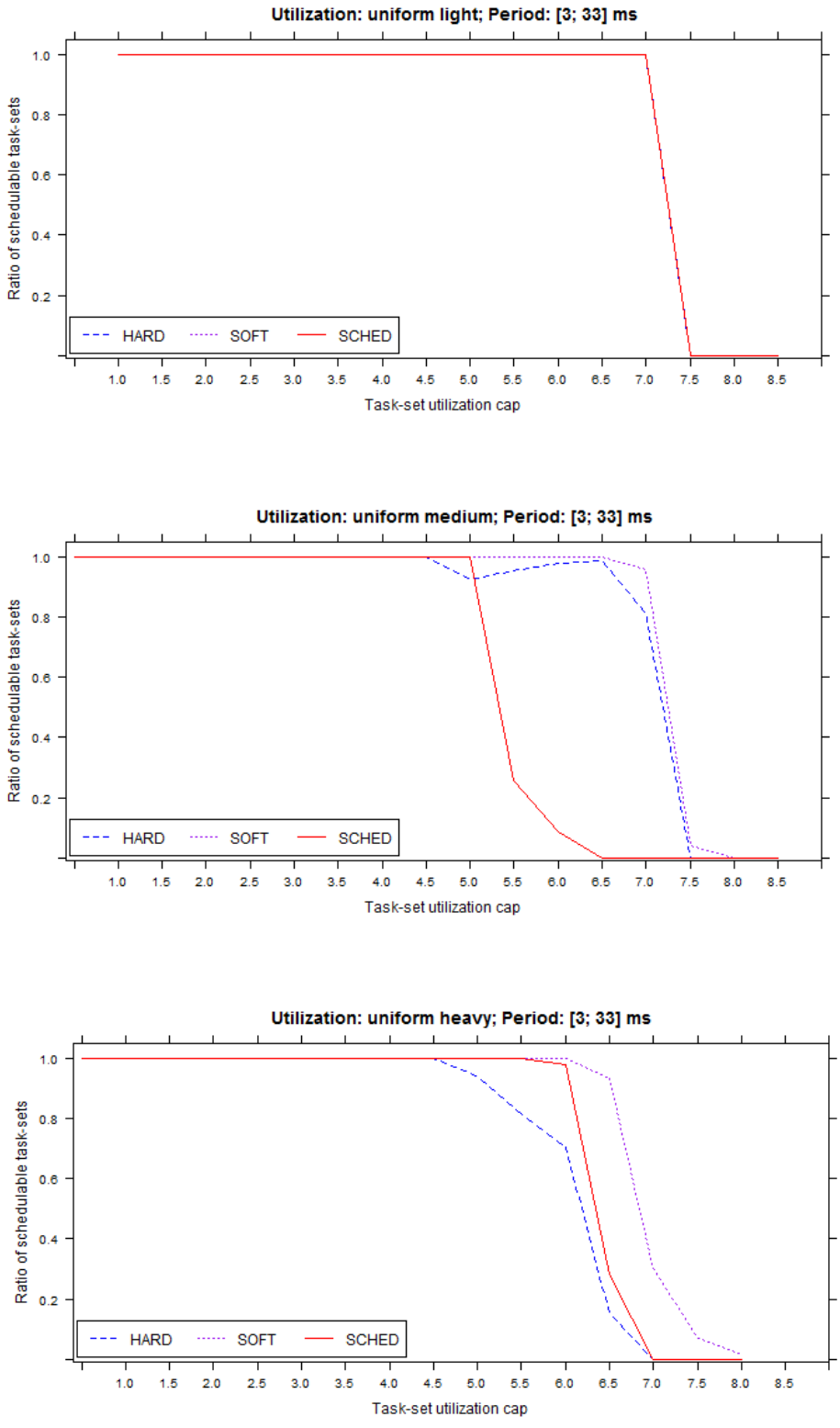


Figure 21:Schedulability test on SMP; period [3,33] ms; distributions: uniform (a), medium (b), heavy (c).

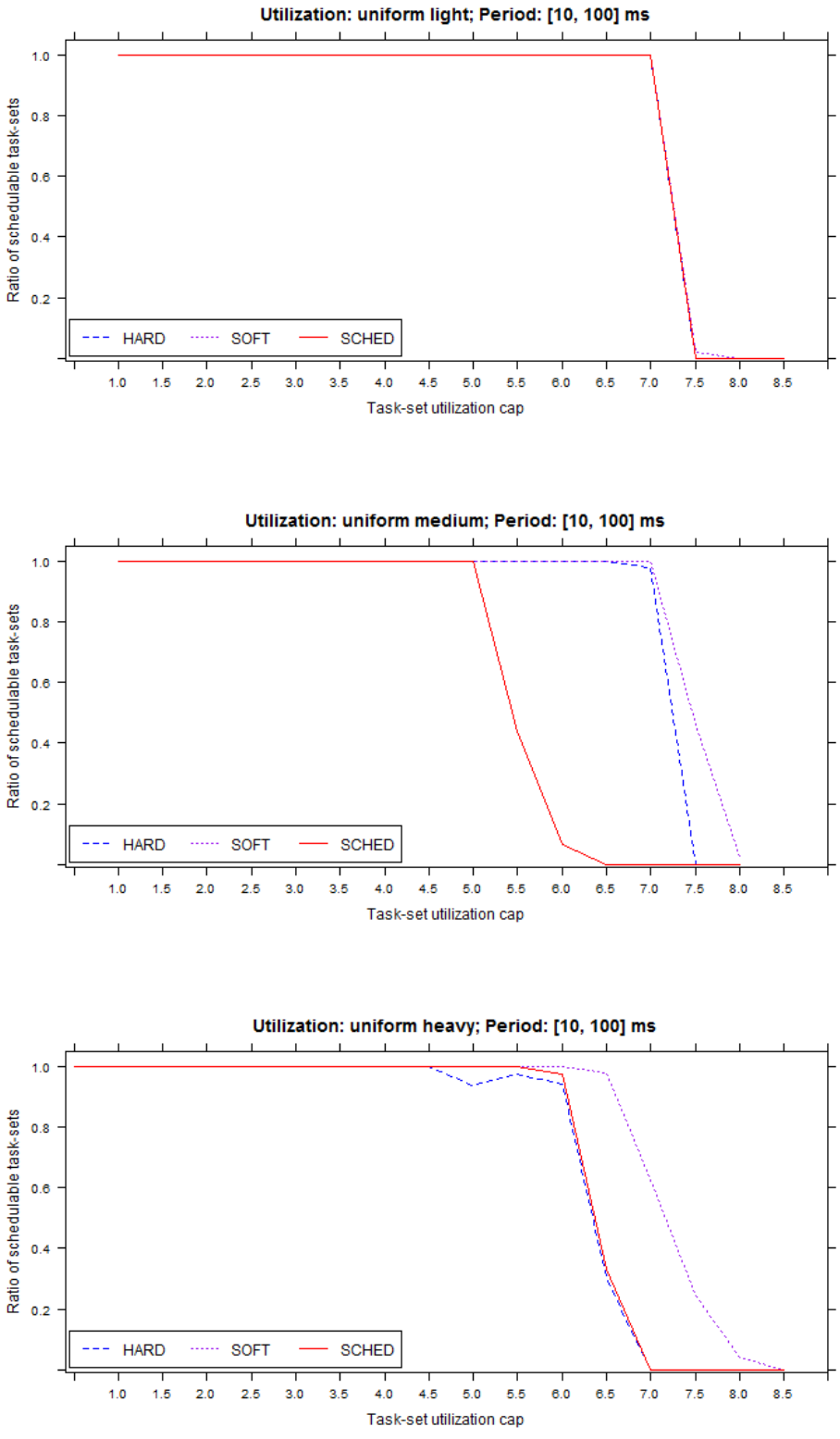


Figure 22:Schedulability test on SMP; period [10,100] ms; distributions: uniform (a), medium (b), heavy (c).

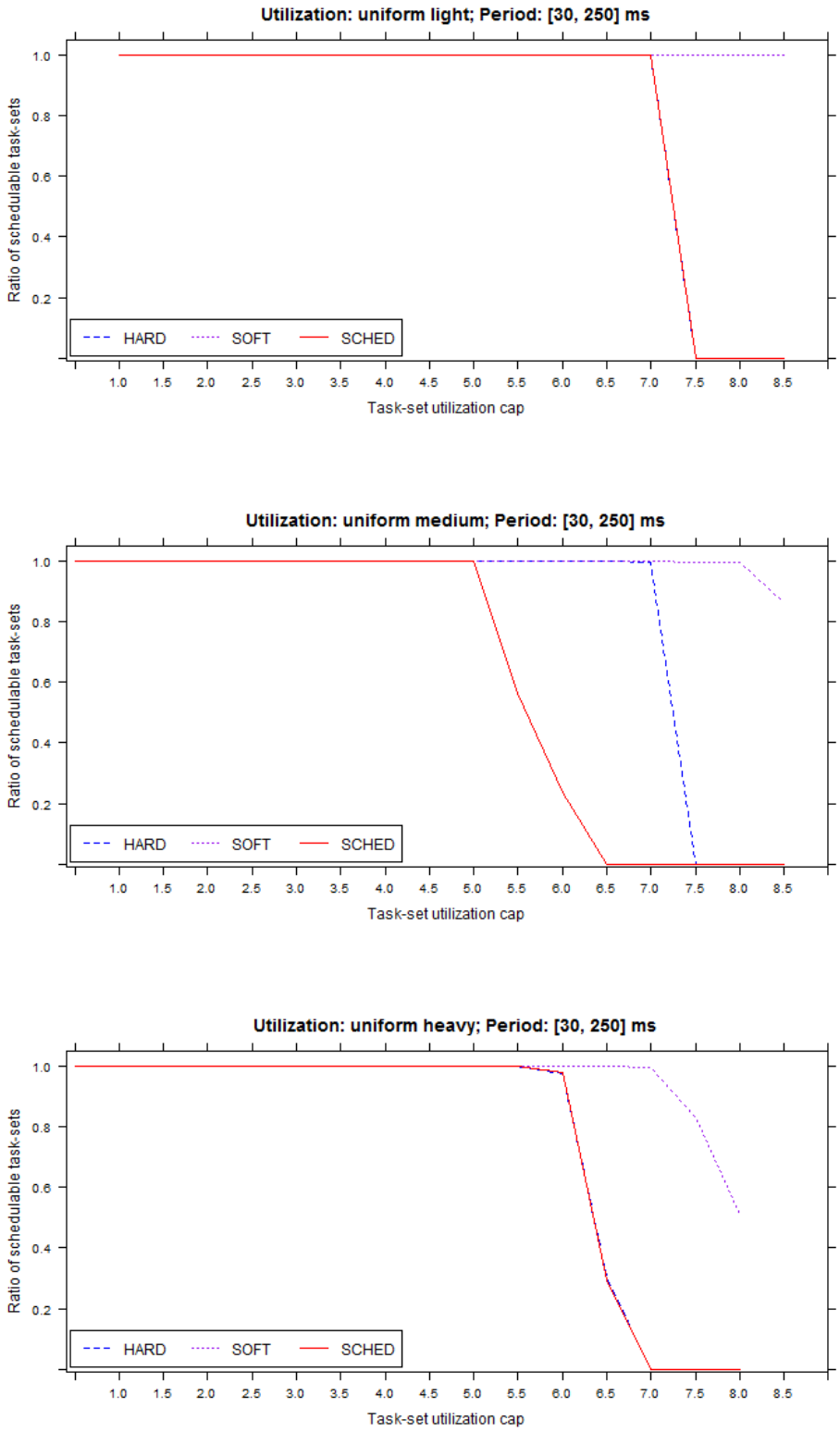


Figure 23:Schedulability test on SMP; period [30, 250] ms; distributions: uniform light (a), medium (b), heavy (c).

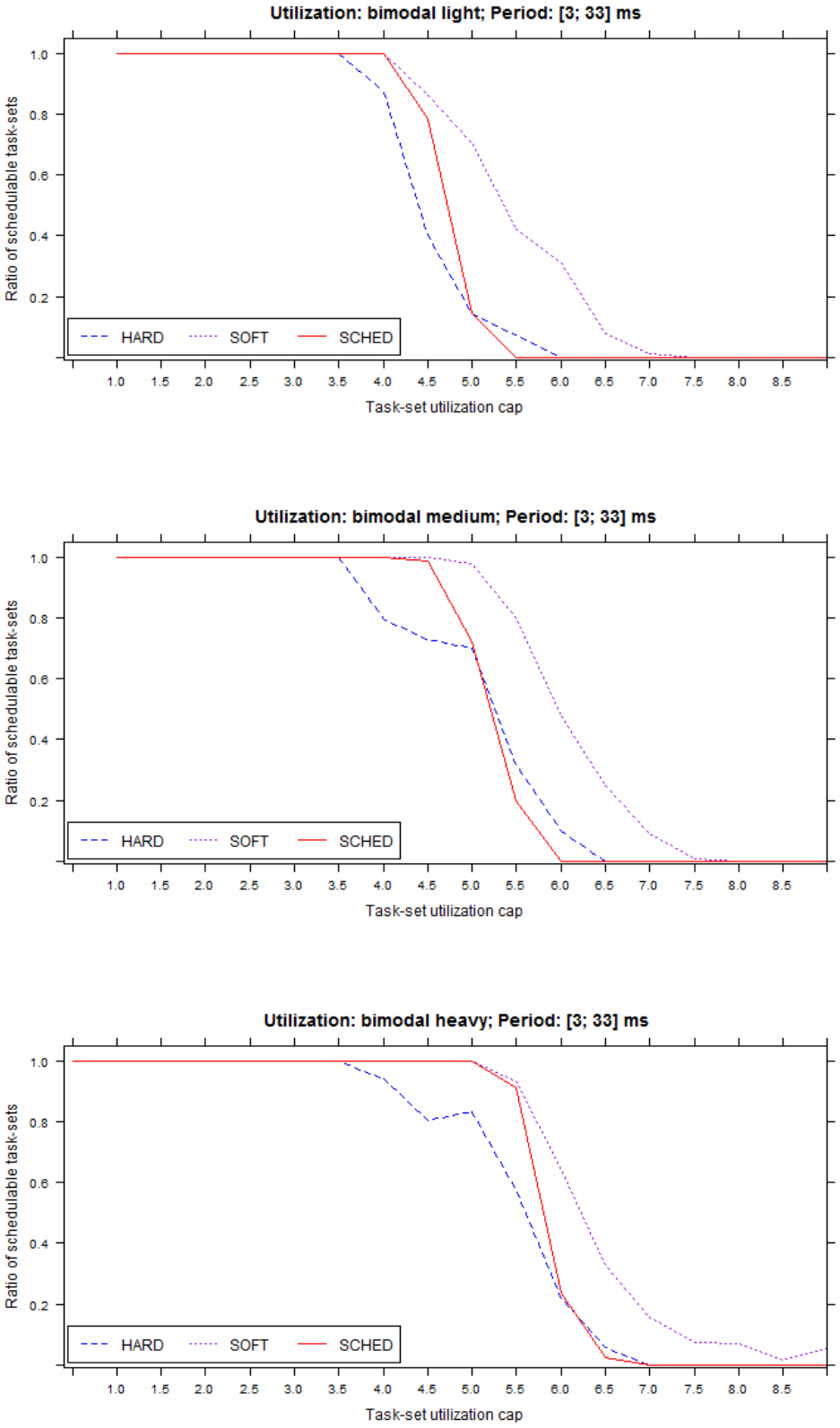
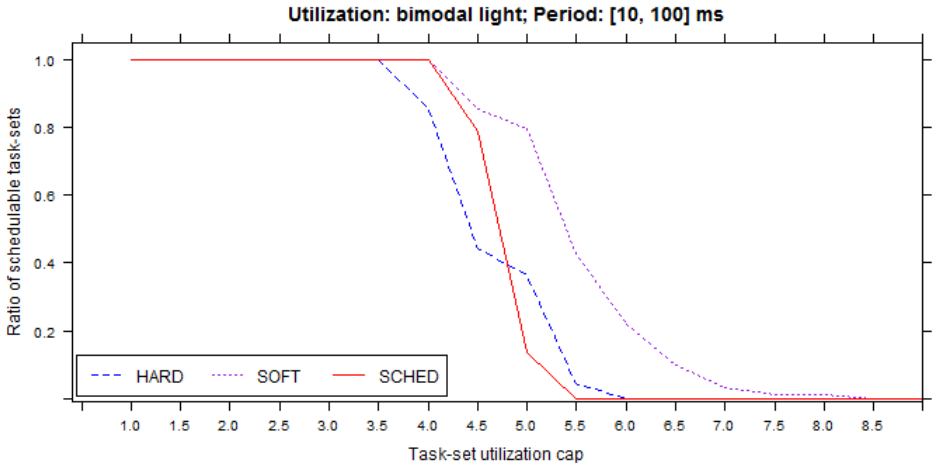
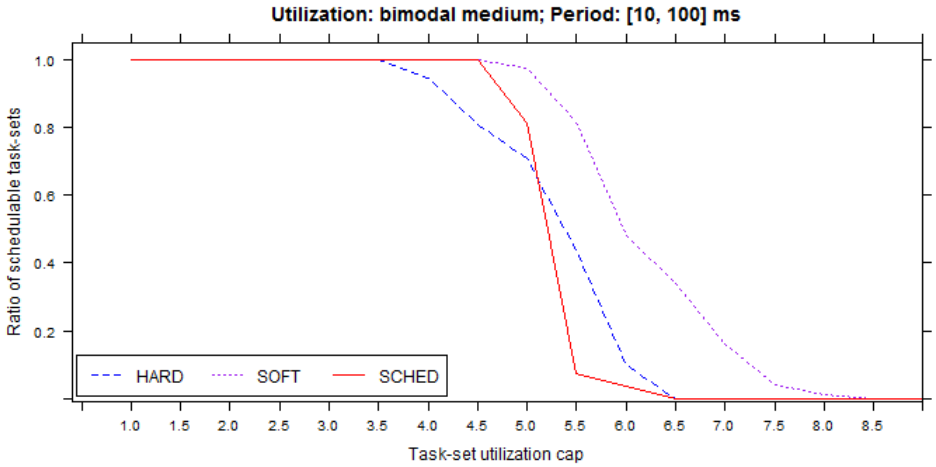


Figure 24: Schedulability test on SMP; period [3, 33] ms; distributions: bimodal light (a), medium (b), heavy (c).





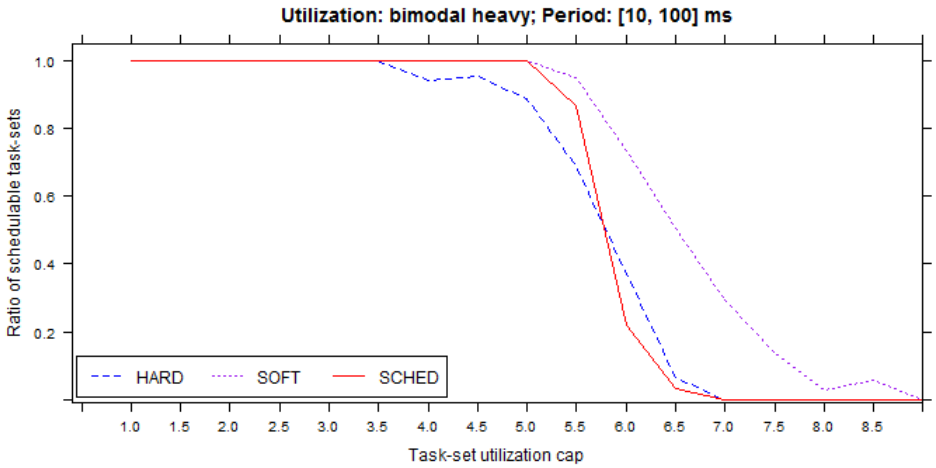


Figure 25: Schedulability test on SMP; period [10, 100] ms; distributions: bimodal light (a), medium (b), heavy (c).

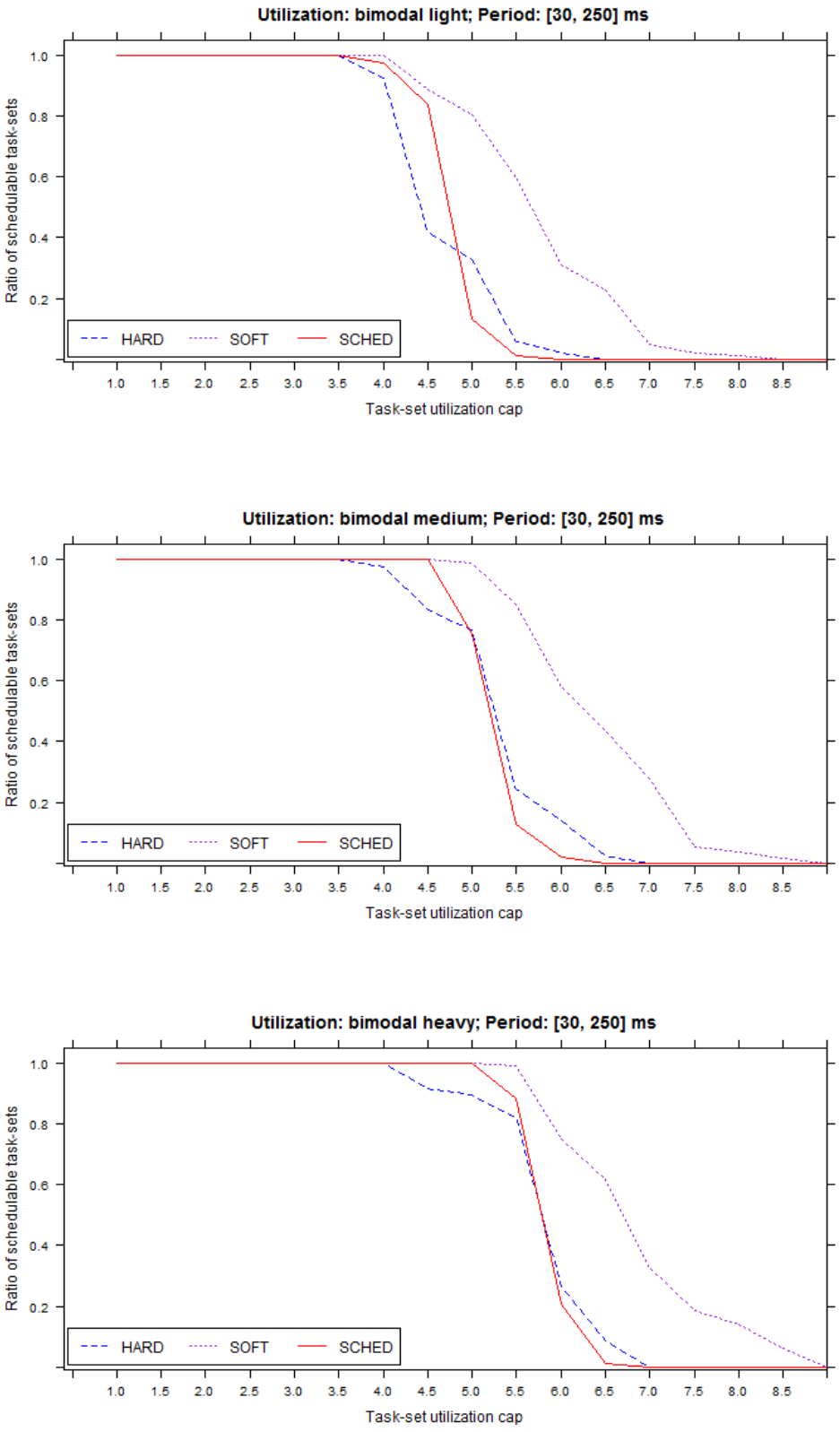


Figure 26: Schedulability test on SMP; period [50, 250] ms; distributions: bimodal light (a), medium (b), heavy (c).

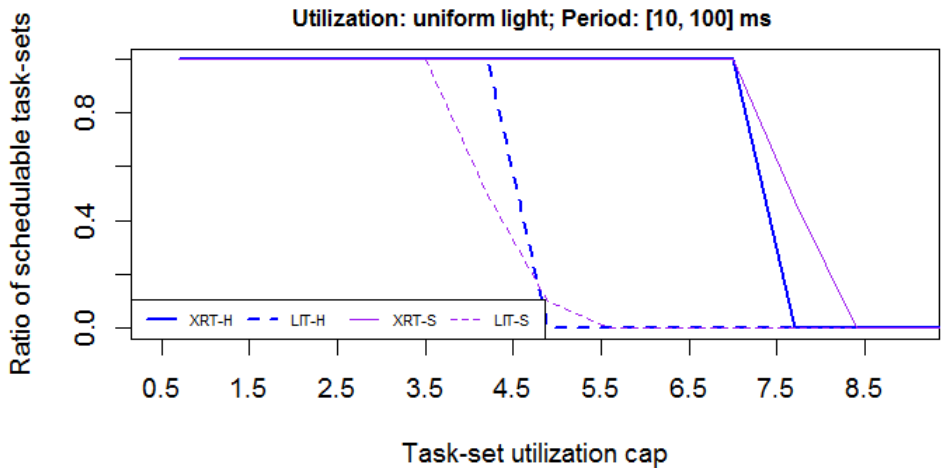


Figure 27: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: uniform light.

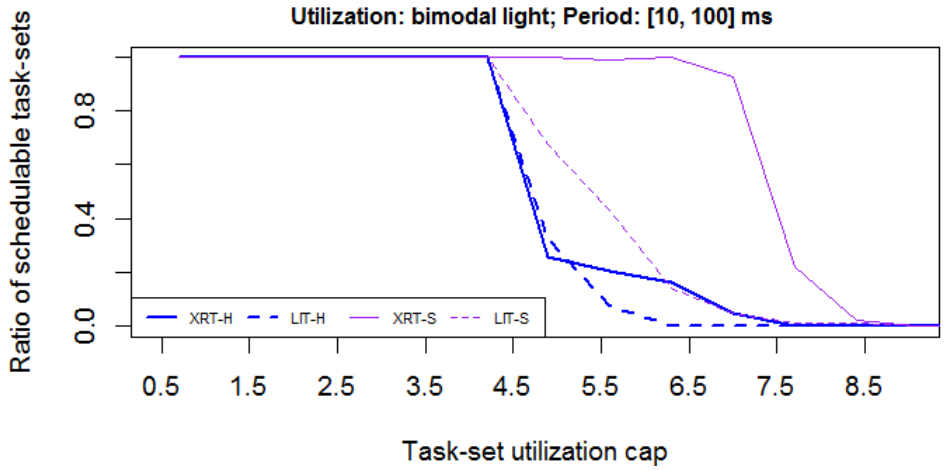


Figure 28: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: bimodal light.

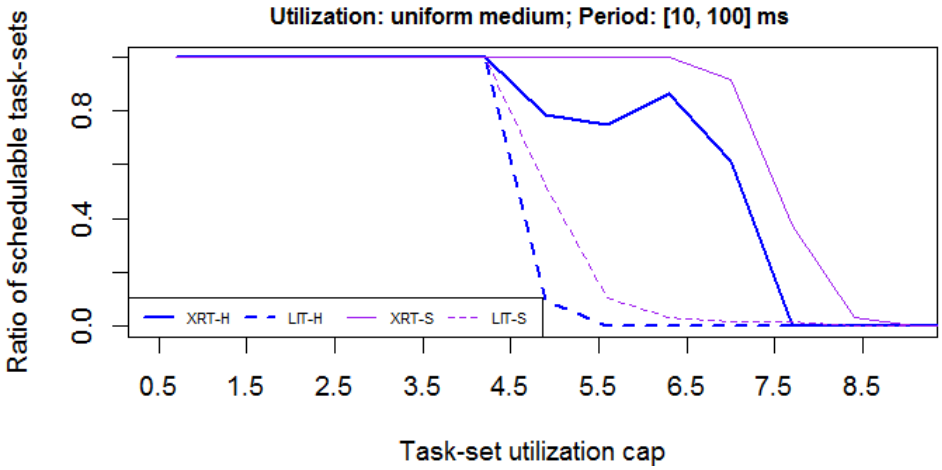


Figure 29: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: uniform medium.

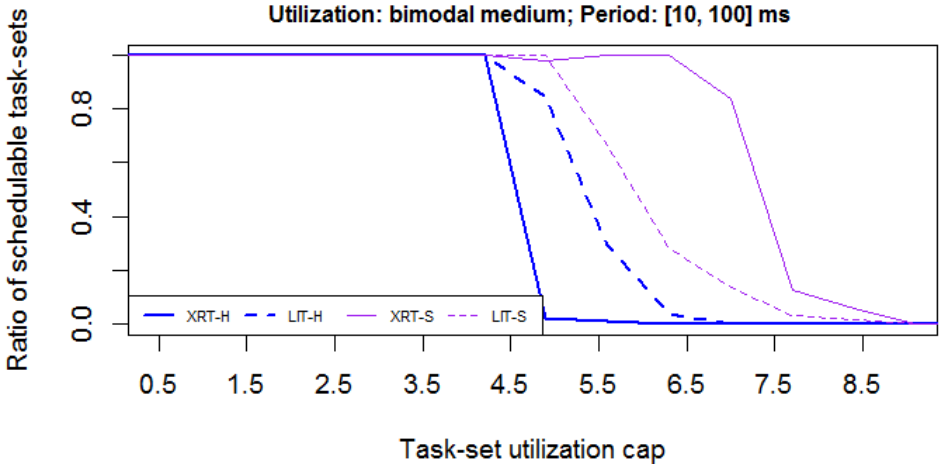


Figure 30: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: bimodal medium.

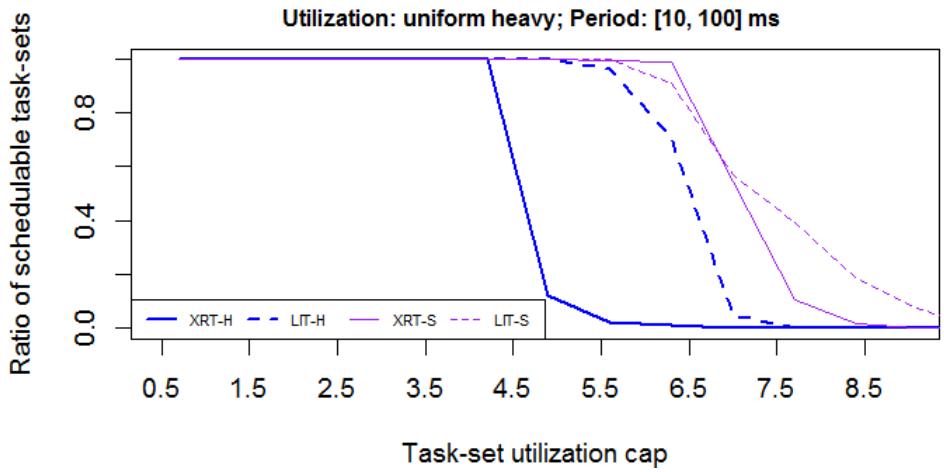


Figure 31: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: uniform heavy.

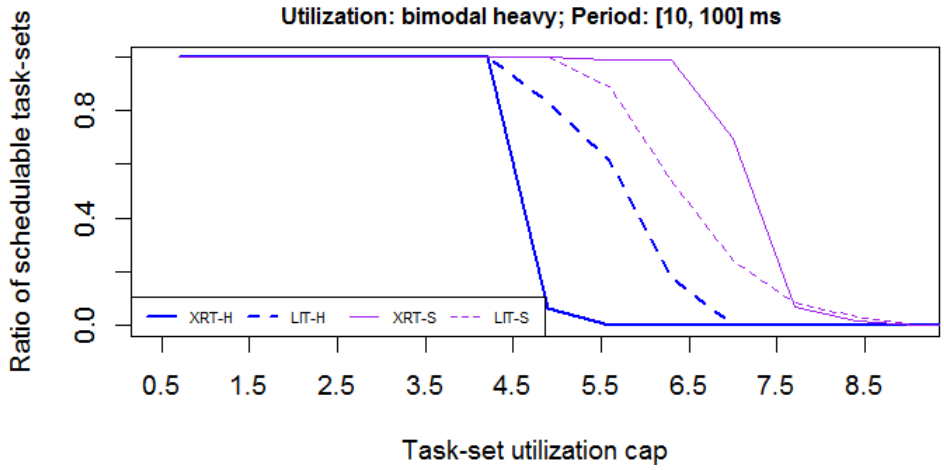


Figure 32: Comparative schedulability tests of LITMUS^{RT} and X-RT on SMP. Period [10, 100] ms.; distribution: bimodal heavy.

3.5. Software architecture for AMP

SMP has become a standard de-facto for many areas in the real-time domain, such as process control and industrial automation, due to the high flexibility offered by its uniform programming model.

However, it has a major drawback in terms of hardware requirements. Some dedicated hardware resources, in fact, are required to support the interlocked operations, cache coherence and other mechanisms typical of SMP systems. Such hardware resources, however, can prove very costly when it comes to small-scale and low-power embedded system due to both area and power requirements.

For such reasons, in many small-scale and highly-integrated embedded real-time systems, AMP reveals to be the leading choice, despite its more complex and less flexible computational model.

Furthermore, it has to be said that, conversely to SMP, where the multiprocessing model is somehow uniform and independent of the specific architecture that implements it (for instance, the software programming model of x86/amd64 multicore processors is not that different from the one of multicore SPARC or PowerPCs), AMP usually doesn't identify a precise model, rather a variety of different and application-specific architectures that often reflect into very different software programming models.

The common principles that typically hold among all the different AMP architectures are mostly related the resulting software organization, which consists in several distinct operating system instances running independently on each processor.

Reference architecture

Before extending the design considerations made so far for SMP to AMP systems, it is mandatory to make some more detailed assumptions about the underlying AMP architecture that is going to be tackled by the X-RT framework.

Considering that the main target of this work has been represented mostly by industrial embedded real-time systems, the choice fell on the area of multiprocessor systems on programmable chip (MP-SoPCs).

Originally exploited as prototyping platforms for later implementation in ASIC, FPGAs have become feasible vehicles for final designs, enabling an agile integration of manifold hardware resources suitably interconnected via a customizable bus, as general-purpose processors (*soft-cores*), memory and peripheral devices. Currently available design tools leave high degrees of freedom to the designer, particularly as regards the inter-processor communication infrastructure and the memory layout. Customization options typically involve not only the choice of the memory technology, which can range from fast on-chip memory to external DRAM solutions, but also the interconnection topology, allowing to tightly couple a memory to a single core, avoiding any contention, or share it, sacrificing access times in favor of resource saving [ANJ+2004, TAK2006].

However, due to the intrinsic nature of FPGAs, the computational performances of soft-cores can often result limited (though still remarkable) if compared to specialized hardware such as modern PC processors, having execution rates ranging within the order of hundreds of MHz. On the other hand, the extreme flexibility of modern SoPC platforms allows many soft-cores to be instantiated on a single FPGA, in order to enhance, accordingly to the user needs, the computational power of the resulting platform by means of multicore hardware parallelism.

Nevertheless, exploiting such multicore platforms in order to concretely take advantage of the hardware parallelism is, in general, a non-trivial task that requires particular care in both the software design and development stages. The complexity level, in fact, is far beyond the traditional uniprocessor scenarios, which are undoubtedly more familiar to the most embedded application developers. In this sense, if we narrow the focus to more specific and constrained scenarios, such as the embedded real-time one, the introduction of an ad-hoc infrastructure which properly masquerades the underlying complexities can make such SoPC multicore platforms feasible and reliable solutions, for instance as in the cases of [BBCG2008, XWB2007].

More technical specifications about the soft-core architecture used for the evaluation of this work are deferred to the *performance evaluations* section. For the moment, the only architectural assumptions made for AMP are: (i) the availability of a shared memory only for holding the working sets of real-time tasks; (ii) a hardware inter-processor communication mechanism that allow the software instances running on the several processors to exchange messages.

The AMP architecture, furthermore, imposes some more restrictions on the scheduling algorithm that can be employed. In particular, conversely to what happens in SMP systems, process migrations cannot be performed. For such reason, for the AMP we are considering the restricted-migration variant of the G-EDF scheduling policy, that is R-EDF [BC2003].

Design of the AMP version of the X-RT framework.

The fundamental contribution brought in by the X-RT framework in the AMP case is represented by the underlying run-time model, herein called *shadow process model*. Its purpose is realize a 1-to-*m* mapping of periodic real-time tasks onto processes of the *m* RTOS instances and manage their execution flow in a centralized manner, according to the global decisions of the scheduling plug-in, realizing a task-level migration abstraction even if RTOS processes cannot be really migrated.

From the software standpoint, each shadow process consists in an instance of the task-shell, which operates in the same way of the SMP version. The main difference is that in this new model, each real-time task is associated to *m* task shells, one for each processor. As in the SMP case, also in AMP the X-RT framework requires only three priority levels (LOW, MEDIUM, HIGH), which keep the same semantic.

At any time, at most one shadow process is ready for execution (from the RTOS scheduler viewpoint) with MEDIUM priority on each of the *m* RTOS instances. This shadow process corresponds to the real-time task that is expected to execute on that processor by the infrastructure. Therefore, keeping the assumption that each RTOS scheduler follows a strict priority-driven policy, no ambiguity can exist as regards the overall set of tasks running on the system at any time.

AMP compliance is ensured since the restricted migration model guarantees that pre-empted tasks cannot be resumed on any processor other than the one where the job execution started, therefore no migration of process context is required. The only state that the infrastructure should care about and keep coherent is the working-set of the tasks, which might be accessed (at different times) by distinct jobs of the same task on different processors. This latter point will be further discussed later in 3.6.

The X-RT framework is deployed in a distributed fashion on AMP: the metascheduler, together with the scheduling policy plug-in, is executed exclusively on one of the *m* processors (which can be used as well for the scheduling of real-time tasks). The scheduling policy plug-in takes global scheduling decisions also in the AMP version of X-RT (with respect of the additional restricted-migration constraint, though). The interface between the metascheduler and the policy plug-in remains unchanged, thus the plug-in remains completely unaware of the underlying process model and SMP/AMP architecture.

The task-shells, conversely, are distributed on the *m* processors. Their operating principle, however, is the same. In this sense, the decoupled architecture of the framework, and in particular the message passing strategy employed to coordinate the metascheduler and the task-shells, shows its best advantages in the AMP scenario, where a coordination based on traditional shared-memory patterns would be completely unfeasible.

An extra component, however, is required to put the shadow process model in operation. Due to the inherent run-time isolation between the RTOS instances, the metascheduler is not capable anymore of directly instantiating the wrapper processes upon instantiation of new real-time tasks. For such reason, a further component, the *dispatcher*(s), is introduced. The framework requires that *m* dispatchers must be pre-loaded on each processor after the initialization of each RTOS instance (Figure 33). From the runtime viewpoint, each dispatcher acts as a local proxy for the centralized metascheduler. The interaction between the metascheduler and the dispatchers is, once again, realized by means of (inter-processor) message passing.

In this regard, a new message (*MSG_CREATE_SHADOW_PROCESS*) is envisaged. Such message is sent by the metascheduler to the dispatchers when a new periodic task is requested to the metascheduler through the *XRT_CreateNewPeriodicTask* method of the X-RT API.

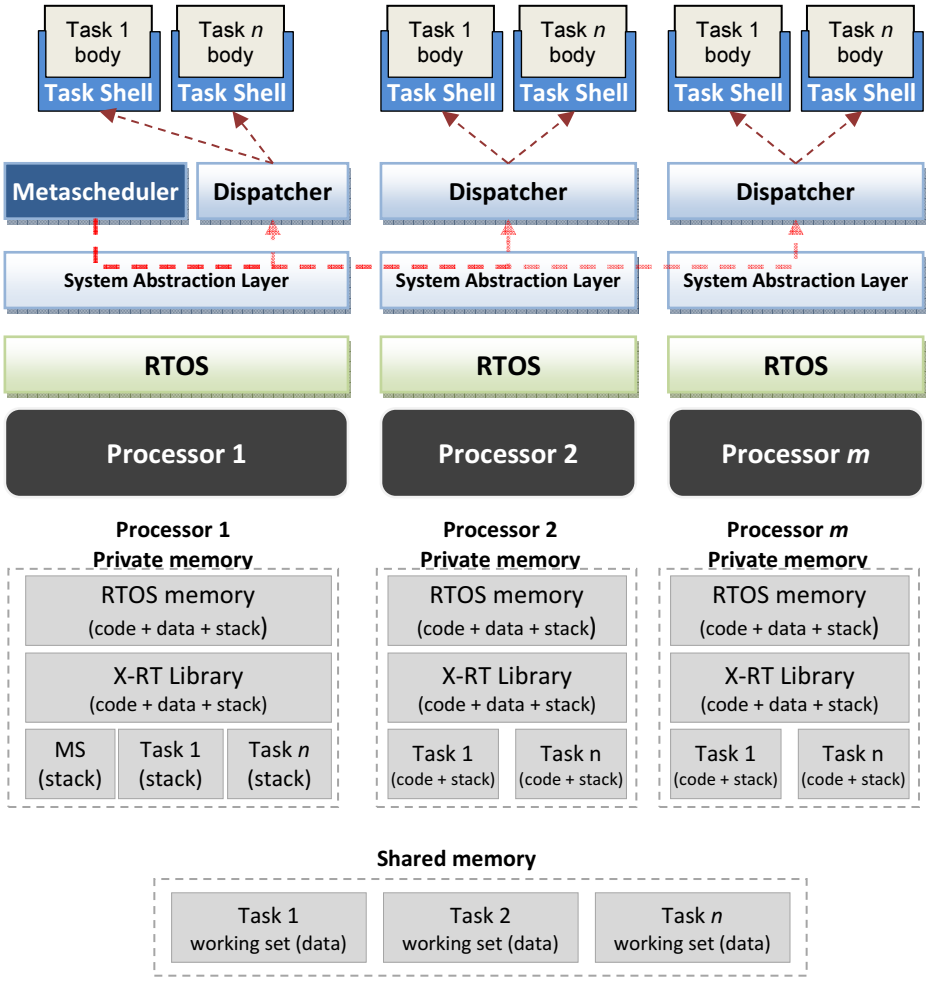


Figure 33: Software and memory organization of the X-RT framework on AMP.

3.6. AMP experimental evaluations

The Altera NIOS-II soft-core has been chosen as reference architecture for the experimental evaluations, due to the flexibility of its integrated development environment that permits easy customization of different hardware templates transparently supported by the bundled µC/OS-II RTOS. The NIOS-II/*f fast* version we employed in our experiments can be further endowed with a write-back directly mapped data cache (D-cache), which permits to reduce bus contentions exploiting spatial and temporal locality of memory accesses. Lacking any hardware coherency support, explicit cache flushes and proper synchronization must be handled by software in order to guarantee coherency of memory shared by different cores. The message-passing infrastructure has been realized using the FIFO core provided by the Altera SoPC, realizing a 1-to-*m* bidirectional channel between soft-cores (Figure 34).

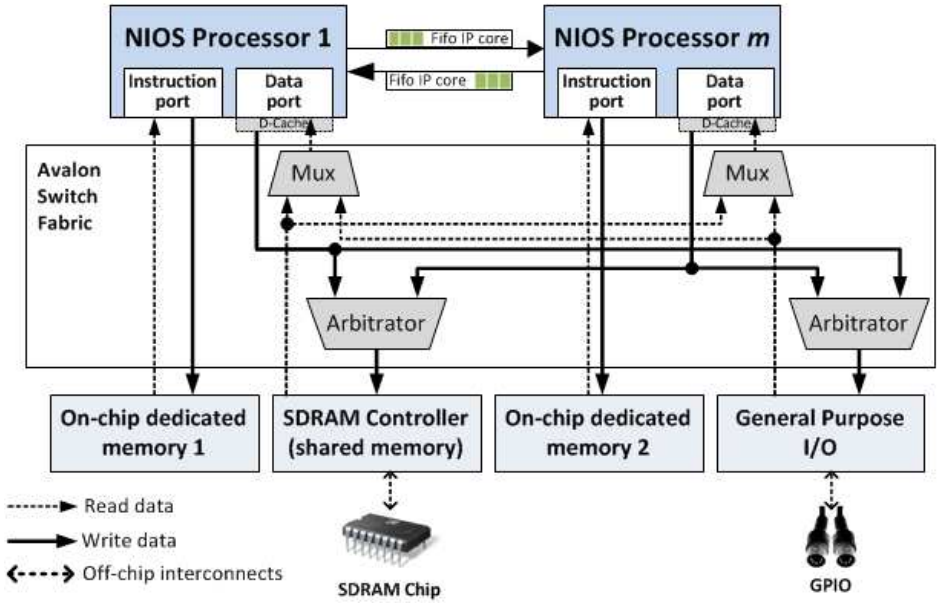


Figure 34: Overview of the Altera SoPC architecture.

Using an Altera Cyclone IV FPGA clocked at 50 MHz and combining different memory and cache layouts as shown in Figure 35, four reference hardware templates based on NIOS-II/f cores have been investigated: *shared memory* (T_S), *shared memory with D-cache* (T_{SC}), *dedicated memory* (T_D), *dedicated memory with D-cache* (T_{DC}). As regards the

memory technology, we used internal MK9 SRAM blocks for the on-chip memory and an external SDRAM module for the shared memory. In order to preserve the memory consistency of the shadow process model in the T_{SC} and T_{DC} templates, explicit cache flushes are performed on job boundaries.

	T_S	T_{SC}	T_D	T_{DC}
Instuctions. cache	2 Kb			
Data cache	No	2 kB	No	2 kB
RTOS memory (Instructions + data)	External memory		On-chip memory	
Tasks memory (Instructions)	External memory		On-chip memory	
Tasks memory (Data)	External memory			

Figure 35: Configuration of the reference hardware templates.

The goals of the experimental evaluation are twofold.

Infrastructure overhead. Two key factors contribute to such overhead: (i) job release overhead, i.e. the interval that elapses between the issue of an *MSG_RELEASE_JOB* message by the metascheduler and the execution of the corresponding shadow process; (ii) job completion overhead, i.e. the interval that elapses between the completion of a job, the update of the working-set and the reception of the corresponding message by the metascheduler. The additional time taken by the scheduling policy plug-in to carry out its scheduling decisions has not been accounted since it strongly depends on the particular policy employed and is extensively discussed by the relative studies herein referred.

Performance slowdown. Apart from the infrastructure overhead itself, the measurements analyze how the run-time execution of application tasks is further biased by the hardware platform. The different hardware templates, in fact, are likely to differently respond to the workload of the real-time tasks, in particular to changes of number of cores simultaneously executing and their working-set size. Furthermore, the more or less frequent context switches and task migrations issued by the scheduling policy can additionally contribute to the run-time duration. In order to account these additional contributes and determine the effective factors which influence them, we set-up an experimental test-bench which combines (Figure 36) the four hardware templates (T) with 4 different number of cores (m), 6 working set sizes (S) , 4 pre-emption rates (P) and 4 migration rates (M, expressed in migrations per period), for a total of 1536 scenarios.

Each scenario involves the scheduling of a fixed number of 16 identical tasks, in which each job executes a CoreMark [Con2009] instance in order to emulate some real workload on the working set. Task periods were chosen to be long enough to compensate duration variance due to the different platforms avoiding overrun conditions. A regular scheduling pattern which relied on a quantum-driven round-robin scheme has been chosen in order to deliver a constant number of preemptions and migrations according to the configuration of each scenario. At each period the 16 tasks are arranged in *m* clusters and each cluster is scheduled on each core in round-robin using a *P* time-quantum ('NO' means that task jobs are sequentially executed). On the next period the pattern repeats shifting the clusters by *M* positions.

$$T\begin{bmatrix} S \\ SC \\ D \\ DC \end{bmatrix} \times m\begin{bmatrix} 1\ core \\ 2\ cores \\ 4\ cores \\ 8\ cores \end{bmatrix} \times S\begin{bmatrix} 512\ B \\ 1\ kB \\ 2\ kB \\ 4\ kB \\ 8\ kB \\ 16\ kB \end{bmatrix} \times P\begin{bmatrix} NO \\ 1\ ms \\ 5\ ms \\ 10\ ms \end{bmatrix} \times M\begin{bmatrix} NO \\ 4\ m/p \\ 8\ m/p \\ 16\ m/p \end{bmatrix}$$

Figure 36: Testbench parameters for the AMP evalutaion.

Experimental results

Figure 37 (a*)* and (b) show the two contributions to the infrastructure overhead. Each column reports the overhead measured for each hardware template in function of *m*, aggregating the average over the variation of *S*, *P* and *M* parameters, as, not surprisingly, they revealed to have a negligible influence on the infrastructure overhead. Job activation measurements show as both the T_D and T_{DC} templates exhibit an almost constant overhead as *m* increases, since the operations performed on the shared memory are minimal. On the other hand, the T_S and T_{SC} templates exhibit a worse scalability, in particular in the case of simultaneous activations on the cores, as both data and instruction ports contribute to the contention of the shared memory module when RTOS scheduling primitives are invoked. Furthermore, it might be also noted that for both the dedicated and shared cases, the relative templates involving data cache exhibit slightly higher overheads. The limited size of the data cache, in fact, is likely to cause a lag due to write-back of the stale cache lines prior to executing the dispatcher code, causing for such a short-length routine an opposite effect

than expected. As regards the completion overheads, both T_S and T_D templates exhibit a very limited, yet expected, contribution. The corresponding templates involving data cache, instead, introduce a more consistent overhead (order of tenths of microseconds) required to invalidate and write-back the data cache in order to preserve the working-sets consistency. In this case, while the T_{DC} template exhibits an almost linear behavior, the T_{SC} template suffers of concurrent data and instruction cache contentions causing increased (≈ 2x) overheads in the 8-cores configuration.

Cumulative infrastructure overhead is shown in Figure 37 (c) as the sum of the two contributions. The dedicated templates exhibit an overall good scalability inducing small and almost constant overhead even in the 8-core configurations, while the shared templates demonstrate to be negatively influenced by the shared memory bottleneck.

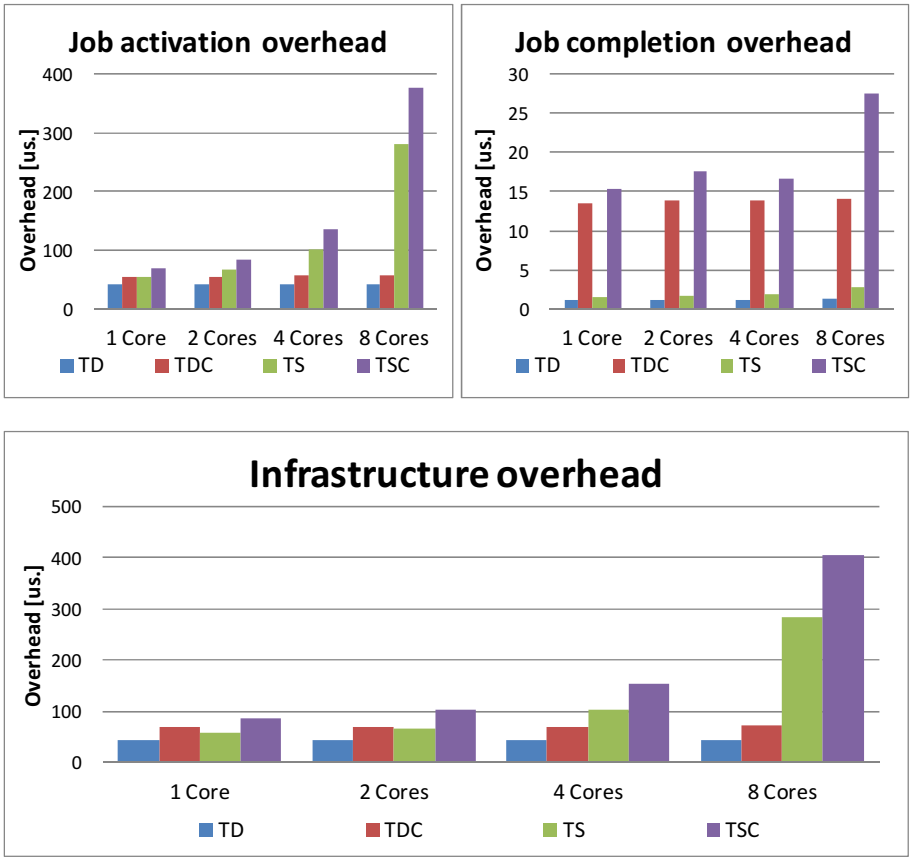


Figure 37: Infrastructure overhead due to job activation (a), completion (b) and cumulative results (c).

In addition to the overhead directly introduced by the scheduling infrastructure, Figure 38 (a-d) show how run-time performance of application tasks is affected by preemptions. Each of the 4 charts reports the average time required to complete a whole job issuing preemptions at different rates (according to the *P* parameter) in function of *m*, under each hardware template. T_D reveals to be the less influenced template incurring, in the {*m*=8 cores; *P*=1 ms} configuration, a slowdown of 1,8% (7 us) compared to the sequential execution case. In the corresponding template involving data cache (T_{DC}), preemptions caused a higher relative increment of 6,9% (5 us.) in the analogous configuration. The shared templates demonstrated to majorly suffer the influence of preemptions, in particular the T_S exhibit a slowdown of 24,5% (98 us) in the {*m*=8 cores; *P*=1 ms} configuration while the introduction of data cache induce in the T_{SC} template a slowdown of 30,8% (25 us). As a broader level consideration it might be noted that the effect of data cache on the preemption overhead has a lesser extent if compared to the speedup provided to tasks run-time.

In order to provide a comparative evaluation of the overall run-time overhead factors, Figure 39 (a-d) show, for each hardware template, the relative slowdowns highlighting, at variations of *W*, the difference between the slowdown due to the hardware architecture and the slowdown due to the scheduling infrastructure. For each column, the lower colored part reports the ratio between the average run-time on the *m*-way multiprocessor configuration performing sequential jobs execution and the corresponding measurement on the uniprocessor configuration. The upper (red) part shows the surplus slowdown, introduced by the infrastructure, using the preemptive round-robin execution with the tightest (*P* = 1 ms) quantum. It may be clearly noted that the slowdown introduced in the infrastructure is definitely marginal in the T_D and T_S templates when compared to the slowdown introduced by the multiprocessor hardware architecture. Such slowdown becomes comparable only in the T_{DC} and T_{SC} templates, highlighting how preemptions suffer a worse exploitation of caches.

As a final remark it might be noted that neither of the considered graphs reports the effect of tasks migrations. In fact, in all of the combinations

considered, the changes of the M parameter did not produce any remarkable effect on the measurements, thus they have been omitted.

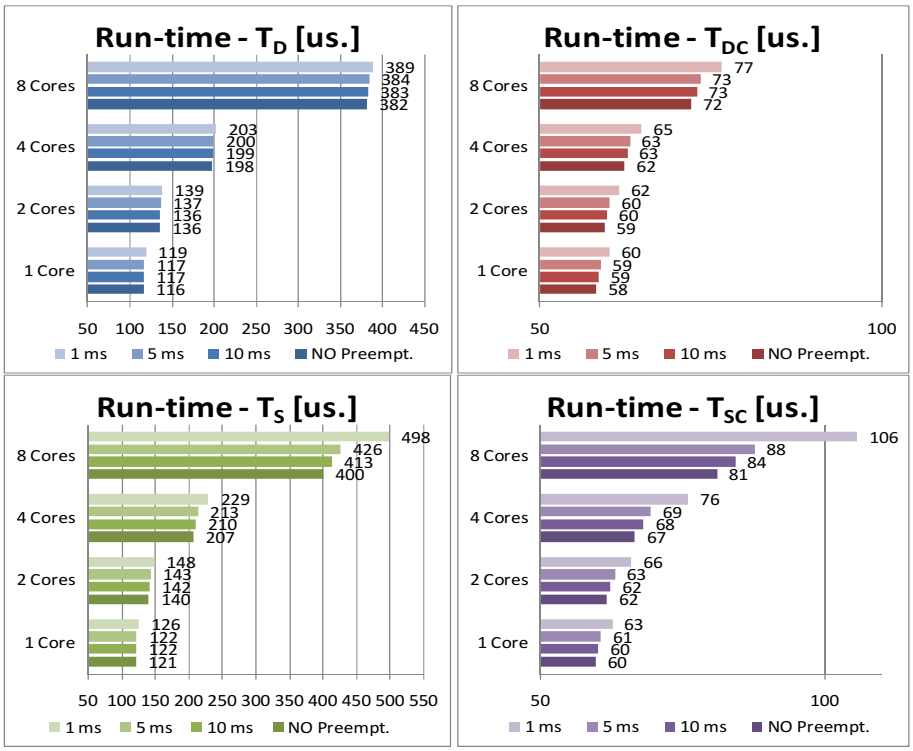


Figure 38: Absolute run-time performances of TD (a), TDC (b), TS (c) and TSC (d) templates varying m and P parameters with W: 16 kB.

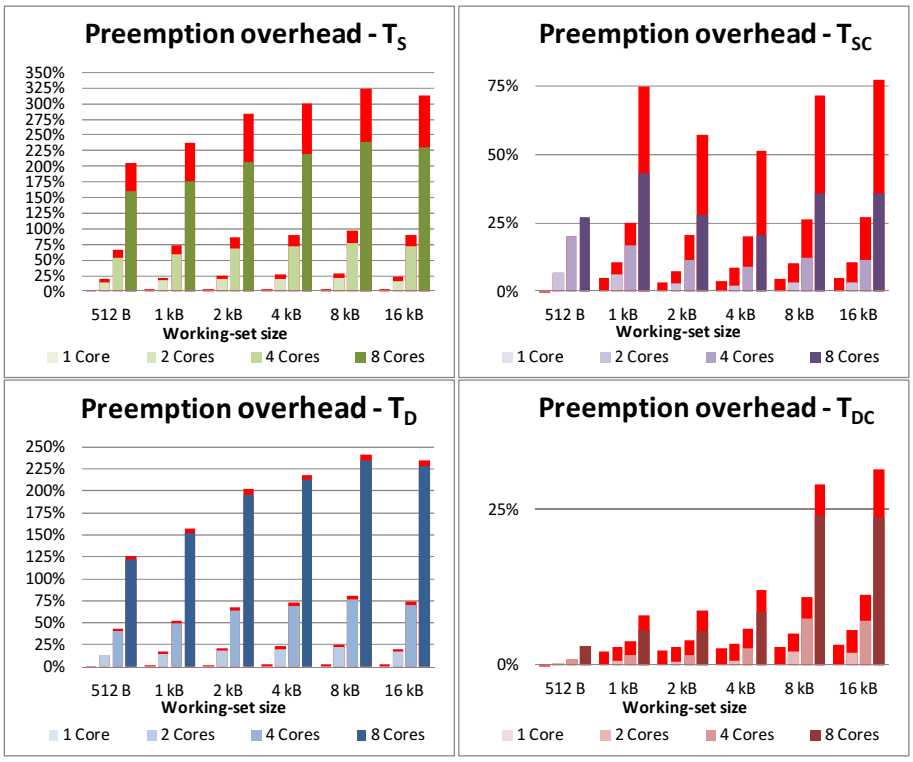


Figure 39: Relative slow-down of TD (a), TDC (b), TS (c) and TSC (d) templates

3.7. Concluding remarks

In this chapter the design considerations and the essential implementation details of a real-time scheduling framework called X-RT have been presented. Such framework enables scheduling of real-time tasks on symmetric and some asymmetric multi-processor platforms, according to global (restricted-migration in the case of AMP) scheduling policies. The focus has been put on the mechanisms that, regardless the particular policy employed, allow to arbitrarily perform job preemptions and task migrations on the mainstream embedded SMP and AMP platforms employing only elementary scheduling primitives offered by almost every RTOS. In order to decouple these low-level scheduling mechanisms from user-definable high-level scheduling policies, a metascheduler approach has been introduced.

The operating principle of this metascheduler stands on a dedicated high-priority process that coordinates the execution of the other processes by means of message passing interaction and dynamically mangles their priorities, using only conventional system calls provided by the RTOS, in order to emulate the operation of more complex global scheduling policies.

Experimental evaluations have been carried out to assess the viability of the approach, employing an Intel eight-thread processor running Linux 3.6 kernel, for the SMP version, and four reference FPGA-based multiprocessor templates combining different memory models and cache layouts for the AMP version. The experimental evaluations analyzed both the overhead directly introduced by the scheduling infrastructure and the further consequences yielded on run-time performances, putting particular attention to the effect of scheduling decisions, i.e. preemptions and migrations, on the tasks run-time.

In this regard the overhead introduced by the proposed framework shows to have a limited extent, both in SMP and in AMP platforms which involve dedicated memory for the RTOS. Furthermore, in the case of AMP platforms, job preemptions induce a slowdown which is smaller than the slowdown caused by the multiprocessor parallelism. Task migrations, furthermore, showed to not cause any remarkable effect on AMP, as the

approach employed does not actually migrate processes, rather it activates different shadow instances on different processors.

As future research directions, as regards SMP, the work herein presented should be extended to take in account cache-related effects, simulating real-world workloads on memory working-sets of various size and using different access patterns, as for instance is done in [Bas2011].

As regards AMP, the experimental evaluations herein presented should be extended in order to contemplate more complex MPSoC architectures involving other communication and interaction paradigms such as network-on-chips, and studying the viability of the approach (or alternative ones) on those hardware platforms which do not assume any shared memory at all.

4. Data structures for timekeeping in real-time systems

4.1. Introduction

In the area of real-time systems, one of the most critical functions typically handled by the operating system is represented by timekeeping. Timers, in fact, represent a key building block for both the operating system itself, for carrying out its internal operations, and for user-space applications, for instance when they take advantage of services like POSIX's timers or `sleep` system calls.

Timekeeping is mainly a software problem, which has, however, tight dependencies on the underlying hardware. It is quite typical, for an operating system, to handle at any time hundreds or thousands outstanding timers, going to expire in sooner or later future intervals. On the other side, the hardware platform typically offers only a few (sometimes just one) programmable hardware timers to carry out the timekeeping activities. Thus, the operating system has to properly multiplex such large queues of software timers using the few hardware timers available. Such multiplexing requires proper data structures.

In traditional systems, most of the timers are required to have just coarse granularities in the order hundreds of milliseconds, for instance in the cases of device drivers dealing with I/O timeouts or user-space applications interacting with the user. However, this trend is changing over time and, nowadays, the number of drivers and applications which require finer grained timers is constantly increasing.

For instance, timers with fine granularity are required by many modern networking protocols to measure accurately small intervals of time. Accurate estimates of roundtrip delay are fundamental for TCP congestion control algorithms on wireless networks [Chi2005] or for distributed protocols like the scalable reliable multicast framework [FJM+1995]. Furthermore, many modern multimedia applications [DTH1992] use high frequency timers, and the number of such applications is nowadays increasing. If then we move to the area of industrial automation, signal processing and embedded real-time systems in general, the number of fine

grained timers, with resolutions down to the nanosecond range, becomes substantial.

The performance of timekeeping operations becomes an issue when fine granularity timers are involved and when the average number of outstanding timers is large. Furthermore, if the timekeeping is performed inside an interrupt service routine, as it actually happens in the most operating systems, such overhead becomes critical for the reliability and responsiveness of the entire system.

Above all, in the case of real-time systems, the RTOS scheduler has a compelling need of timers, since it must deal with periodic release of tasks and monitoring of their deadlines with extremely high accuracy. In this scenario, timekeeping represents the most crucial activity in the performance path of the most RTOS operations.

In order to have a qualitative idea of the impact that timekeeping overhead has on the runtime performances of a real-time system, just consider a very modest real-time application, involving, for instance, a dozen periodic tasks with periods and deadlines in the millisecond range: the RTOS scheduler will need to intervene several thousand times in each second. Thus, if the timekeeping routine takes even just a few microseconds for its execution, it would introduce an overage overhead of about 5% of CPU time.

However, more than the average case, the worst case overhead is the most crucial aspect to account for in the runtime behavior of a RTOS. If the scheduling overhead is negligible in most of its interventions (so that even the average overhead is negligible), but occasionally takes larger amounts of time, its effect may be catastrophic if interleaving with the execution of a hard real-time task with a small slack.

For these reasons, in real-time systems ensuring that the overhead introduced by timekeeping operations is bounded, for instance by means of exploiting appropriate data structures which can guarantee that by design, is generally preferred than keeping an extremely low average-case behavior with longer worst cases, as usually happens in the design of general purpose operating systems.

In the following, the topic of timekeeping is explored from the software implementation viewpoint, first analyzing some traditional approaches already known in technical and scientific literature, and then discussing a novel approach, introduced by this thesis, designed for time-critical and memory-constrained embedded real-time systems.

4.2. Problem statement

We consider, in the following, the problem of handling a set of an arbitrary number of timers, by means of the following primitives:

`StartTimer(timer_handle, interval, expiry_callback)`

Invoked by the application to request the start of a timer, which will expire after `interval` time units. The caller supplies a reference (`timer_handle`) to the timer object, which in most real-world implementations is simply an opaque pointer, used to distinguish requests for this timer from other timers in the system. Upon expiration, the `expiry_callback` function will be called back, if the timer has not been stopped in the meantime.

`StopTimer(timer_handle)`

Invoked to stop the timer referenced by `timer_handle`.

`Timekeeping()`

Software routine, typically invoked upon a hardware timer interrupt, responsible for updating the state of the registered timers, according to the chosen *timekeeping methodology* (discussed soon), and triggering the execution of the `callback` for the expired timers.

There are two main timekeeping methodologies which can be used to interact with a hardware timer: *tick-driven* and *tick-less* handling.

Tick-driven handling

Tick-driven handling is the most straightforward way to realize timekeeping. It requires the hardware timer to simply deliver its interrupts at fixed rate (*tick rate*), triggering the execution of the `Timekeeping` routine at equidistant intervals of time. The tick rate is typically decided upon system initialization and never changed at runtime.

While such a way of handling timers is evidently simple, it has a major issue: the periodic interrupt handler introduce a constant source of overhead, even when there is no compelling need. In other words, the hardware timer interrupt handler will execute at its usual rate even if there are no timers registered. Secondly, the maximum resolution for all the software timers handled is dependent on the `tick rate`. For instance, if the hardware timer is programmed to a rate of 1000 Hz, the maximum resolution allowed for software timers is 1 ms.

A minor advantage of tick-driven handling is that the expiration time can be stored as a relative interval, rather than an absolute time. This can save some memory and some arithmetic computation time on very small microcontrollers with few memory and very low (i.e. 8 or 16 bits) data parallelism. Relative timekeeping, however, is not considered in this thesis as practically irrelevant for most of the modern platforms.

Tick-less handling

Tick-less handling is an alternative and more advanced methodology to deal with timekeeping [SPV2007]. It requires a high-resolution programmable interval timer (PIT), which nowadays is available on most hardware platforms, and is often directly embedded in the processor. A PIT is a free-running monotonic counter driven at a fixed rate (usually in the order of nanoseconds) that ideally never wraps.

The advantage of a PIT is that it doesn't deliver interrupts at a predetermined rate; rather it is further endowed with a register, freely re-programmable by the software, which triggers an interrupt only when the internal counter reaches that value. In other words, a write to the PIT register marks a decisional instant in a precise moment in the future, which triggers a single interrupt.

The basic idea behind tick-less handling, therefore, is that the operating system keeps the hardware PIT always triggered to match the software timer expiring soonest. Thus, every time an interrupt is triggered, the corresponding software timer callback is invoked and the hardware PIT is reprogrammed with the value of next software timer, if any.

This synchronization between the PIT and the software timer queues, however, is not that straightforward as it might seem at a first glance. In fact software timers can be stopped, as well as new timers can be added at any time. In all these situations, the synchronization with the PIT must be ensured in order to never miss a timer event.

The two aforementioned timekeeping methodologies reflect in a very realistic way what happens in the majority of real-world RTOS. In some cases, whereas the underlying hardware platform is endowed with both the hardware timers, the operating system can provide both forms of timekeeping. For instance the Linux kernel, in some configurations, is able to provide lower resolution timers, handled in a tick-driven fashion by the tick handler, and high-resolution timers (namely `hrtimers`) handled in a tick-less fashion with a nanosecond resolution [GN2006].

4.3. Traditional data structures for timekeeping

Dense array of timers

One of the most straightforward ways of realizing software timekeeping is modeling the timers queue in memory as contiguous arrays of timer entries. In this model the entries are compacted within the array, reflecting their creation order (but not their `interval`), as in Figure 40.

Timer 1	Timer 2	Timer 3	Free entry	Free entry
56	51	62		
*expiry_callback	*expiry_callback	*expiry_callback		

Figure 40: Dense array data structure for timekeeping problems.

Computational worst-case complexity

In such a structure the implementation of the `StartTimer` routine is very straightforward. Supposing to know the number of active timers in the queue, in order to start a timer, the routine can directly index the next free entry and store the timer entry in O(1). The `timer_handle` can be just a unique identifier assigned to each timer upon creation.

On the other side, however, the `StopTimer` routine requires a full scan of the array, in order to search for the given `timer_handle`, and shift all its following entries to re-compact the array, thus with O(n) complexity, with n being the number of active timers.

As regards the `Timekeeping` routine:

- In case of *tick-driven* handling, it has to increment the system time and check whether it has reached the value of one or more active timers. Once an expired timer is detected, the corresponding `expiry_callback` is invoked and the expired timers are shifted out similarly as in the `StopTimer`, thus requiring again O(n) time in the worst-case.
- In case of *tick-less* handling, the `Timekeeping` routine has to scan the array to find the expired timer, remove it and then retrigger the PIT with the value of the next timer expiring soonest, requiring O(n) time.

Memory complexity

Since the memory usage of this model is basically O(N), with N being the maximum number of active timers allowed, it works fine in the cases where a bound on the maximum number of timer can be determined a priori. The static nature of the array, instead, turns out to be particularly inefficient when the number of active timers is extremely variable, thus forcing to either over-allocating a huge array or to rely on memory reallocation techniques. Memory relocation, however, can be very time-consuming since it can involve a deep copy of the previous array in a larger one. For these reasons this model usually fits only the case of “home-brewed” timekeeping for very modest applications, where the number of timers is small and known a priori.

Sparse array of timers

A slightly variation of the latter model can be obtained relaxing the density constraint and allowing free entries to interleave active timers entries, as depicted in Figure 41.

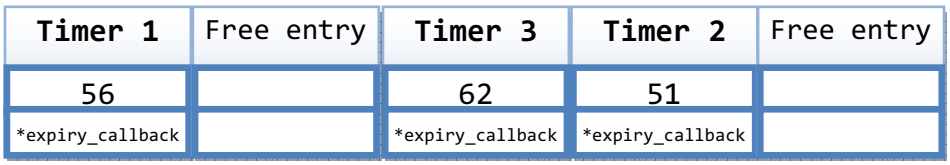


Figure 41: Sparse array data structure for timekeeping problems.

Computational worst-case complexity

The behavior of this model is almost the same of the dense array model, with the difference that the opaque `timer_handle` in this case can directly reflect the index within the array of its entry, since entries are never shifted until they expire or are stopped. The consequence of that is that now the `StartTimer` has to find a free entry before writing the new timer descriptor, thus requiring O(N) time, with N being the length of the array. `StopTimer`, instead, can use the `timer_handle` to directly index the entry to be removed and just wipe it, in constant O(1) time (only in tick-driven handling, though).

The `Timekeeping` is almost unaltered, with the only difference that, in all cases, its operation requires, in the worst case, O(N) rather than O(n) time, since no assumptions can be made on the density of the array, thus on the position of a timer entry.

Memory complexity

The same considerations of the dense array model apply.

Sorted linked list

In the sorted linked list model (Figure 42) active timer entries are kept sorted, with respect of their expiration time, in a doubly linked list. Conversely to the array-based variants, this model provides dynamic expandability, not requiring to pre-allocate a-priori any storage for the

timer entries (with the only exception of head and tail pointers, which, however, are fixed regardless the number of timers in the queue).

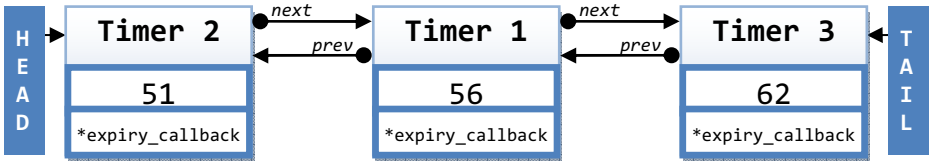


Figure 42: Linked list data structure for timekeeping problems.

Computational worst-case complexity

In such a structure the `StartTimer` routine needs to walk the linked list, find an entry with a later expiration, and insert the new one before that, thus requiring O(n) time in the worst case.

Since the knowledge of a single entry is sufficient to walk the queue in both directions, the `timer_handle` can directly reflect the memory address of the corresponding timer entry. For such reason, the `StopTimer` routine can directly remove any entry in constant time, thus with O(1) complexity, just linking together its previous and next entries.

As regards the `Timekeeping` routine:

- In case of *tick-driven* handling, it just needs to increment the system time and check if it reached the value of the first active timer entry (the queue's head). If the head is expired, the corresponding `expiry_callback` is triggered and the entry is removed, thus requiring constant O(1) time (under the assumption that expiration times are unique, otherwise it would requires O(t) with t being the number of timers concurrently expiring, which is still optimal).
- In case of *tick-less* handling, the `Timekeeping` routine needs just to remove the head of the queue, invoke the corresponding `expiry_callback` and retrigger the hardware clock with the absolute time of the next entry, thus requiring constant O(1) time. Furthermore, whenever the first timer is stopped (thus the list's head is removed), the PIT just needs to be retriggered with the value of the new list's head.

Memory complexity

It might be immediately noted that this model has an optimal memory utilization, requiring O(n) memory to keep only the active timer only.

Timing wheel

Timing wheel is a more complex data structure, mostly intended for tick-driven timekeeping, presented by Varghese et al. in [VL1987], and further refined in [VL1997]. It basically consists in a fixed-length array of linked lists of timers (Figure 43). Time is divided into cycles, with each cycle consisting in N time units. The information about the current time is kept through a combination of a single array of length N, which keeps the state for the current cycle in a modular fashion, and a cycle counter c. The current time t, therefore, is represented by the tuple $\{c, i\}$, that is, the current number of cycles and the index within the cycle array, such that $t = c \cdot N + i$, and in every moment the N entries of the array correspond to the time interval $[c \cdot N \; ; \; 2 \cdot c \cdot N - 1]$. Each of the N linked lists contain the timer entries that expire in the corresponding time identified by the index of the list. The i index is incremented modulo N and, when it wraps, c is consequently incremented by one in order to reflect the new cycle. Timers whose expiration exceed such interval are placed in a so called *overflow list*, which is checked upon each new cycle boundary (discussed later).

Computational worst-case complexity

In a timing wheel, the `StartTimer` routine proceeds as follows.

First of all, it has to check if the expiration time e of the new timer falls within the current cycle, i.e. if $c \cdot N \leq e < c \cdot (N+1)$, or beyond it, i.e. $e \geq c \cdot (N + 1)$. In the former case, the new timer entry is appended to the linked list of index $i = (e \bmod N)$. In the latter, it is appended to the overflow list. In both cases the `StartTimer` require constant time, since it involves only an array lookup and a unsorted list enqueue operation, thus its worst-case complexity is O(1).

Since timer entries are always part of a linked list (either a list of the array or the overflow list), the `timer_handle` can be directly implemented as the memory address of the corresponding timer entry, similarly to the case of

the sorted linked list model. Thus, also in this case the `StopTimer` has O(1) complexity, at least as regards tick-driven handling.

The `Timekeeping` routine, instead, requires some more careful analysis.

- In case of *tick-driven* handling, the `Timekeeping` routine updates the system time incrementing the *i* index by one modulo *N*. Two scenarios are possible: (i) $i < N$ and (ii) $i = 0$, thus it wraps. In (i), the only operation to be performed is checking if the linked list addressed by the incremented *i* index contains any timer, and if so call the `expiry_callback` for those and remove them, thus requiring constant O(1) time (or still optimal O(*t*) relaxing the assumption on expiration time uniqueness). The latter (ii) case, instead, is definitely more complex, since upon each new cycle, the overflow list must be processed looking for timers which expiration time falls in the new cycle, and if so they need to be moved from the overflow list to their corresponding list in the N-length array, thus requiring O(*n*) time in the worst case.

  As a final consideration, it might be noted as adding a sorting restriction to the overflow list can favor the `Timekeeping` complexity, which becomes O(*N*), in favor of the `StartTimer` routine, which consequently becomes O(*n*).

- In case of *tick-less* handling, instead, the situation gets worse. In fact, once a timer expires (or equivalently when first timer is stopped), the next timer expiring soonest must be looked-up, in order to retrigger the hardware PIT. This operation, however, requires to iterate over the array until a non-empty list is found, thus requiring O(*N*) time. Even worse, if no more timers are present in the current cycle, the cycle corresponding to the next timer must be loaded from the overflow list, requiring additional O(*n*) time. Thus the worst case complexity in the case of tick-less handling is O($N + n$), in practice making timing wheels an unfeasible choice for this scenario.

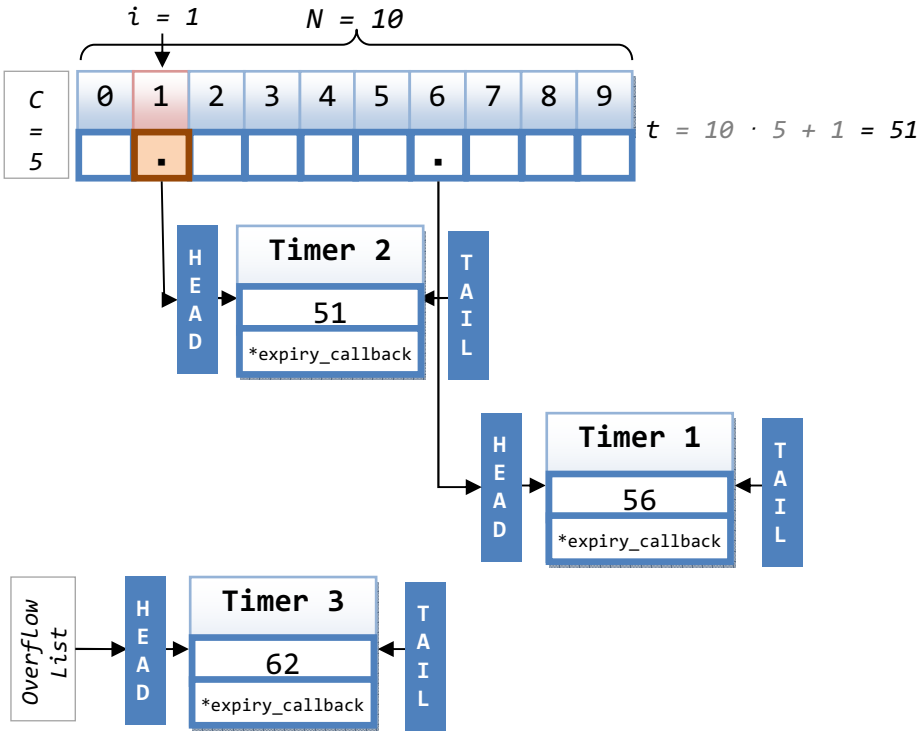


Figure 43: The timing wheel data structure for timekeeping problems.

It might be noted as the average computational complexity of the timing wheel model, in case of tick-driven handling, is extremely good, requiring constant O(1) time in *N*-1 cases out of *N*. For such reasons, it turns out to be a particularly suitable solution for handling timers in general purpose operating systems. The Linux 2.6 kernel, for instance, employs a variant of the timing wheel data structure presented for handling coarse granularity timers.

The remaining O(*n*) case (wrapping), however, still represents a non negligible worst-case, making this data structure not suitable for real-time systems. Taking again the case of the Linux kernel, high resolution timers are handled by means of a completely different and more deterministic data structure [GN2006], a self-balancing binary tree, which is going to be discussed in the next section.

A sliding-window variant of the original timing wheel aims at ensuring a constant O(1) complexity for the `Timekeeping` routine when all the timers are registered with an expiration interval of at most *N* time units. If any

timers are registered beyond that bound, however, the overflow list comes up again, bringing back the original O(n) behavior. Thus the approach remains impracticable for real-time systems which require nanoseconds resolutions, unless a very large N is employed to cover the horizon of possible timer intervals, though yielding a significant memory usage.

Additionally, in [VL1997] another variant is presented, aimed at distributing the overflow list over the N arrays by means of hashing, thus reducing the average cost of the overflow list processing to O(n/N). However, no particularly assumptions can be made on the worst case complexity, unless introducing some strong assumptions on the distribution of the timer intervals.

Memory Complexity

The main strength of the timing wheel model, and most of its variants, lays on a memory vs. computational complexity tradeoff. In general the largest the array is, the lowest probability of processing entries in the overflow list it gets, though its worst case complexity remains linear. For such reasons, its memory complexity of O(N + n) it is far away from being optimal, and can become an issue when a large number of queues, requiring a timing wheel each, is required

Self-balancing binary search tree

Another approach for organizing timers in memory is represented by exploiting binary search tree (BST). A BST is a tree-based data structure (Figure 44) in which each node η has (at most) two children η_L, η_R, which respect the following ordering relation: $\eta_L < \eta \leq \eta_R$. The point of BST is to keep its nodes sorted by their key, in order to allow fast (i.e. O(*log(n)*)) insertion, retrieval and removal operations.

The main issue of conventional BST, however, is that, depending on the insertion/removal pattern, the tree can easily degenerate in a linked list (for instance, simply inserting new nodes in increasing order of their keys). As the tree degenerates in a flat list, the run-time behavior of its operations degenerates into linear complexity.

For such reasons, self-balancing BST (SB-BST) are typically preferred when worst-case behavior is a concern [ST1985]. Most SB-BST implementations (e.g., Red-black trees [Knu2006] and AVL trees [AL1963]) have the same organization of a conventional BST, differing only in the behavior of the insertion and removal operations. Qualitatively, their operating principle is based on spending little more effort upon each modification of the tree (though still keeping a logarithmic complexity) in order to keep the height of the tree small, thus guarantee a logarithmic behavior to subsequent operations.

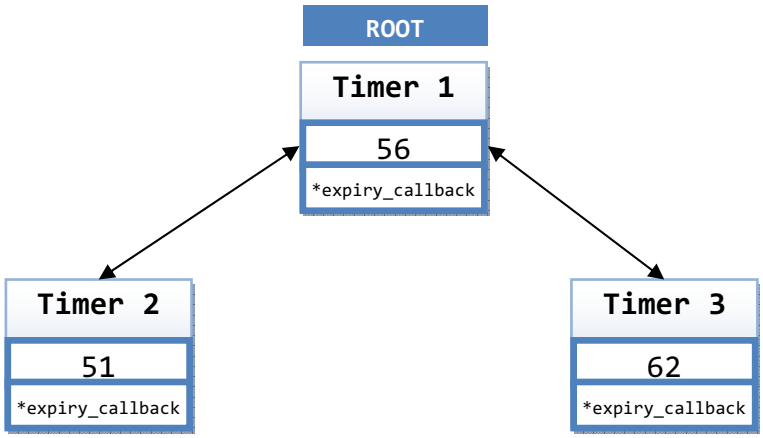


Figure 44: Binary search tree data structures for timekeeping problems.

Computational worst-case complexity

In the timekeeping scenario, SB-BSTs are employed to keep timer entries sorted by their expiration time, i.e. each timer entry is modeled by a BST node, operating as follows: the `StartTimer` routine inserts the timer entry by means of a conventional self-balancing BST insertion, which requires O(log(*n*)) time in the worst case.

Like a linked list, BST nodes have the additional benefit of being directly addressable, i.e. the knowledge of a single entry is sufficient to traverse the tree in any direction. Thus the opaque `timer_handle`, in the BST model, can be directly implemented as the memory address of the corresponding timer entry. For such reason, the `StopTimer` routine can remove any entry, by means of a self-balancing tree removal operation, in O(log(*n*)) time.

As regards the `Timekeeping` routine:

- In case of *tick-driven* handling, the `Timekeeping` routine just needs to increment the system time, check if it reached the value of the active timer entry expiring soonest, that is the leftmost leaf of the BST and, if so, trigger the corresponding `expiry_callback` and remove it. Both the lookup and removal operations require O(log(*n*)) time in the worst case when the BST is balanced.
- In case of *tick-less* handling, the `Timekeeping` routine needs just to remove the leftmost leaf of the BST, invoke the corresponding `expiry_callback` and retrigger the hardware clock with the absolute time of the next entry (its parent node), thus requiring O(log(*n*)) time for keeping the BST balanced after removal.

Memory complexity

SB-BST have an optimal memory utilization, requiring O(*n*) memory to keep only the active timer entries. In order to be able to traverse the tree in either direction, starting from an arbitrary node, each node need to keep three pointers (left child, right child and parent) in its payload, in addition to the expiration time and the `expiry_callback` pointer.

Array-backed binary heap

In the following, another model based on an array-backed data structure called binary heap is presented. A binary heap is typically employed to implement a priority queue, that is, an abstract data type, similar to a queue, where each element has a *priority* associated with it.

A priority queue supports the following two operations: `insert_with_priority`, that inserts an element into the queue with a given priority and `remove_highest_priority_element`, which removes from the queue and returns the element that has the highest priority. Priority queues are typically employed in a wide variety of applications such as graph problems, discrete event simulation, network routing and, of course, timekeeping. In the specific case of timekeeping, the priority queue elements are represented by timer entries which priority is inversely proportional to their expiration time, such that the element with the maximum priority represents the timer expiring soonest.

From a logical viewpoint, a binary heap is a tree-like data structure, which nodes respect two properties: (i) *shape property*: a binary heap is always a complete binary tree, i.e. all levels of the tree, except possibly the last one are completely filled, and, if the last level is not complete, the nodes of that level are filled from left to right. (ii) *heap property*: each node has a higher or equal priority than its children. In this regard, it might be worth nothing that no relationship exists between the priorities of nodes on the same level. Compared to the BST, in fact, a binary heap induces a more relaxed ordering among its nodes.

From a memory layout viewpoint all implementations of the binary heap known so far are based on arrays (*array-backed heap*). Since the logical structure of a binary heap is a complete binary tree, its physical structure can be stored in memory through an array, according to the breadth-first binary tree implicit representation.

In this arrangement, no pointers are required to address children or parent nodes, as they can be directly indexed in the array as follows: if a node has an index i, its left and right children are found, respectively, at indices $2 \cdot i+1$ and $2 \cdot i+2$, while its parent (if any) is found at index $\lfloor (i-1)/2 \rfloor$, assuming the root has index 0 (Figure 45).

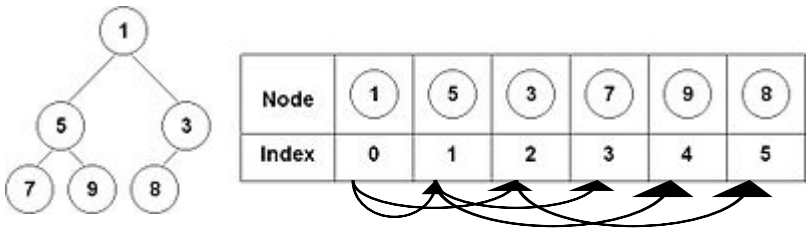


Figure 45: Array-backed binary heap data structure for timekeeping problems.

Computational worst-case complexity

We consider in the following a binary heap backed by an array of size N, containing, at the time the routines are invoked, $n < N$ active timer entries.

The `StartTimer` routine proceeds as follows. The new timer entry is placed in the array at index n (the first non-occupied slot). While this placement preserves the *shape property*, the new timer entry may violate the *heap property* since its expiration time might be closer than its parent

entry. In such case, the new entry must be recursively swapped with its parent, until the *heap property* is restored. This operation, typically called *bubble-up* requires O(log(*n*)) time in the worst case, that is, the case in which the new timer expires sooner than all other *n* timers (has the highest priority), thus it must be percolated up through all the height of the tree (which is $\lfloor \log_2(n) \rfloor$) up to the root.

The main issue that arises with array-backed data structures in general, and in particular with a binary heap, is the addressability of the single entries. In fact, since the position of an entry within the array is not fixed, in the binary heap model the `timer_handle` cannot directly use the memory address of the entry. Typically, items are addressed by means of a unique id (e.g., a counter which is monotonically incremented upon each `StartTimer invocation`).

Therefore, in the case of a `StopTimer`, unless the timer to be removed is exactly the root of the binary heap, its id must be looked-up requiring a full visit of the entire heap, giving the `StopTimer` a worst-case runtime complexity of O(*n*). Eventually, the complexity of the lookup operation can be improved (at the expense of the other operations) and become O(log(*n*)) by using an alternative model for the `timer_handle` based on indirect addressing. Indirect addressing basically consists in using handle objects to establish the node-id to array-position mapping. Such handles, however, need to be updated every time a node's position is altered (e.g. by bubble-up or percolate-down operations), thus adding overhead to most of the heap mangling operations.

As regards the `Timekeeping` routine:

- In case of *tick-driven* handling, it just needs to increment the system time, check if it reached the value of the root node of the binary heap and, if so, trigger the corresponding `expiry_callback` and remove it by means of a `remove_highest_priority_element` operation. The latter involves two stages: (i) replacing the root node, by definition placed at index 0 in the array, with the last element of the array (that is the downmost and rightmost node of the tree) in order to preserve the shape property of the heap. (ii) Then, the new root is percolated down, re-iteratively swapping it with its highest priority children,

symmetrically to what happens in the insertion case, until the heap property is restored. Since, in the worst case, the binary heap must be fully traversed in its height, the `Timekeeping` routine has O(log(*n*)) complexity.

- In case of *tick-less* handling, the operations to be performed by the `Timekeeping` routine are almost unchanged: it has to remove the root, invoke the corresponding `expiry_callback` and retrigger the hardware clock with the absolute time of the next entry, that is the new root of the binary heap resulting after the `remove_highest priority_element` operation on the expired one.

Memory complexity

It might be immediately noted that, as in the array-based models previously presented in beginning of this chapter, the memory usage of this model is also O(*N*), requiring an upper bound estimation on the maximum number of active timers or runtime memory relocation techniques.

As a final remark, it can be noted that, the worst-case complexity of this data structure is never better than the BST model. Its analysis in this thesis might seem arguable at a first glance. However, two further considerations must be done in this regard: first, the worst-case runtime complexity herein analyzed gives an indication of the asymptotic behavior of the models, but doesn't give any information about their actual performances. Such performance analysis will be carried out in the end of this chapter. Secondly, the logic structure of the binary heap model underpins the architecture of a novel data structure called *addressable binary heaps*, discussed in the next section, which shares with this one its logical structure.

As a summary of this section Figure 46 gives an overall overview of the worst-case run-time and memory complexities of the models analyzed.

Model	StartTimer w.c. complexity	StopTimer w.c. complexity	Timekeeping TD w.c. complexity	Timekeeping TL w.c. complexity	Memory complexity
Dense array	O(1)	O(*n*)	O(*n*)	O(*n*)	O(*N*)
Sparse array	O(*n*)	O(1)	O(*N*)	O(*N*)	O(*N*)
Sorted list	O(*n*)	O(1)	O(1)*	O(1)*	O(*n*)
Timing wheel	O(1)	O(1)	O(*n*)	O(*N* + *n*),	O(*N* + *n*)
SB-BST	O(*log(n)*)	O(*log(n)*)	O(*log(n)*)	O(*log(n)*)	O(*n*)
Binary Heap	O(*log(n)*)	O(*n*) / O(*log(n)*)	O(*log(n)*)	O(*log(n)*)	O(*N*)

Figure 46: Overview of worst-case computational complexity and memory complexity of the analyzed timekeeping models

4.4. The addressable binary heap

In order to overcome the limitations of the array-backed binary heap, a novel approach, called addressable binary heap (ABH), is presented. The aim of ABH is to implement a binary heap by means of a pointer-based tree-like physical structure. The design of the ABH lays on the conventional layout of a binary tree, in which each node is linked to its two children by means of pointers. No pointer to parent is required.

As in the array-backed binary heap, the nodes of the tree respect both the *shape property* and the *heap property*. In the following we present the details of the `insert_with_priority` and `remove_highest_priority_element`, showing as they can be implemented with a logarithmic complexity. The removal routine is further extended to the general case of removing an arbitrary node from the ABH, still keeping a logarithmic worst-case complexity.

Insertion of a new node, with arbitrary priority, in the ABH.

The first issue that arises when inserting a new node in an ABH, is finding its proper location, in order to keep the tree complete and respect the shape property. In the array-backed case this is immediate, since using the

implicit tree representation the right position (downmost and leftmost) corresponds directly to the index n of the array, where n is number of nodes already present. In a pointer-based scenario it requires a little more analysis.

Definitions

- $L(\eta)$: *level of a node*

  Given an arbitrary node η of the ABH, we denote with $L(\eta)$ the level of such node, that is, the number of parent nodes of η. There is only one node η_1 in the ABH such that $L(\eta_1) = 0$ and that node is the root.

- η^{-N} : N^{th} *parent of a node*

  Given a node of the ABH $\eta \neq \eta_1$, and a natural number N, such that $N \leq L(\eta)$, we denote with η^{-N} the N^{th} parent of the node η, such that $L(\eta) - L(\eta^{-N}) = N$, with N=0 being the identity $\eta^0 \equiv \eta$.

- η_L η_R: *left and right children of a node*

  We denote with η_L η_R, respectively, the left and right children of a node η.

- $P(\eta)$ *: path of a node*

  Given a node of the ABH $\eta \neq \eta_1$, we denote with $P(\eta)$ the binary sequence $[0 \mid 1]^{L(\eta)}$, such that each i^{th} $(0 \leq i < L(\eta))$ element of the sequence, with i=0 being the rightmost element, is

  $$P(\eta)_i = \begin{cases} 0 \text{ if } \eta^{-i} = (\eta^{-i-1})_L \\ 1 \text{ if } \eta^{-i} = (\eta^{-i-1})_R \end{cases}$$

  More informally, the path of a node is a binary string that describes the sequence of branches that must be taken to reach the node from the root. The length of the binary string is equal to the level of the node and, for each level, each zero (one) bit means that the path for that level follows the left (right) child.

- $I(\eta)$ *: index of a node*

Given an arbitrary node η of the ABH, we denote with $I(\eta)$ its index within the tree, counting the nodes left to right, from the root level $L(\eta_1)$ up to the upper level $L(\eta_N)$, starting from $I(\eta_1) = 1$.

Property 4.4.1

Given an arbitrary node η of index $I(\eta)$, the indexes of its left and right children are, respectively: $I(\eta_L) = 2 \cdot I(\eta)$ and $I(\eta_R) = 2 \cdot I(\eta) + 1$.

Proof

Let us consider an arbitrary node η of the ABH. From the definition of index, $I(\eta)$ corresponds the number of nodes that precede it (top to bottom, left to right) plus one.

Therefore, the index can be alternatively be expressed as:

(1) $I(\eta) = A(\eta) + S_L(\eta) + 1$

Where $A(\eta)$ is the number of ancestors of η (upper-level nodes) and $S_L(\eta)$ the number of its left siblings (preceding nodes on the same level).

Furthermore, from the definition of complete binary tree:

(2) $A(\eta) = 2^{L(\eta)} - 1$

(3) $S(\eta) = S_L(\eta) + S_R(\eta) = A(\eta) = 2^{L(\eta)} - 1$

More informally, the number of ancestors of a node η is equal to the number of its left and right siblings.

Thus

$S_L(\eta) = I(\eta) - A(\eta) - 1$ *From* 1

$= I(\eta) - (2^{L(\eta)} - 1) - 1$ *From* 2

(4) $= I(\eta) - 2^{L(\eta)}$

By construction, the number of right siblings of η is:

$S_R(\eta) = S(\eta) - S_L(\eta)$

$= (2^{L(\eta)} - 1) - (I(\eta) - 2^{L(\eta)})$ *From* 3,4

(5) $= 2^{L(\eta)+1} - I(\eta) - 1$

We can now determine I(η_L), that is, the index of the left child of η. Per definition of complete tree, the number of nodes that lay between η and η_L is exactly the number of right siblings of η plus the number of children of the left siblings of η, that is $2 \cdot S_L(\eta)$. Thus the index *of* η_L is:

$$I(\eta_L) = I(\eta) + S_R(\eta) + 2 \cdot S_L(\eta) + 1$$

$$= I(\eta) + (2^{L(\eta)+1} - I(\eta) - 1) + 2 \cdot (I(\eta) - 2^{L(\eta)}) + 1 \quad \textit{From } 5,4$$

$$(6) \quad = 2 \cdot I(\eta)$$

Thus, the index of the right child:

$$I(\eta_R) = I(\eta_L) + 1$$

$$= 2 \cdot I(\eta) + 1 \quad \textit{From } 6$$

□

Property 4.4.2

$P(\eta) = (I(\eta) - 2^{L(\eta)})_b$ for each node of the ABH $\eta \neq \eta_1$.

More informally: *The path to any non-root node of the ABH corresponds to the binary representation of its index minus its most significant bit.*

Proof

The property can be verified by induction, as follows.

It is immediate to verify that the property is valid for the first two non-root nodes at level 1, that are, the nodes η_{1L} and η_{1R} of index 2 and 3:

$$P(\eta_{1L}) = 0_b = 10_b - 10_b = 2 - 2^1 = I(\eta_{1L}) - 2^{L(\eta 1L)}$$

$$P(\eta_{1R}) = 1_b = 11_b - 10_b = 3 - 2^1 = I(\eta_{1R}) - 2^{L(\eta 1R)}$$

Furthermore, by definition of path, given an arbitrary node η of the ABH, the paths of its left and right child are, respectively:

(1a) $P(\eta_L) = P(\eta) \# 0_b = 2 \cdot P(\eta)$ (# → binary concatenation operator)

(1b) $P(\eta_L) = P(\eta) \# 1_b = 2 \cdot P(\eta) + 1$

More informally, the path of the left (right) child of a node is equal to the path of that node concatenated with 0_b (1_b) or, equivalently, multiplied by two (plus one).

Let us now consider a generic node η with $L(\eta) > 1$ and suppose that the property is valid for the level $L(\eta)$

(2) $P(\eta) = I(\eta) - 2^{L(\eta)}$

From property 4.4.1, the indexes of the two children of the node η are:

(3a) $I(\eta_L) = 2 \cdot I(\eta)$

(3b) $I(\eta_R) = 2 \cdot I(\eta) + 1$

Hence, at level l+1, the path of the left child of η will be:

$P(\eta_L) = 2 \cdot P(\eta)$ *From* 1a

$= 2 \cdot (I(\eta) - 2^{L(\eta)}) = 2 \cdot I(\eta) - 2^{L(\eta)+1}$ *From* 2

$= I(\eta_L) - 2^{L(\eta)+1}$ *From* 3a

$= I(\eta_L) - 2^{L(\eta L)}$ *Per definition of child*

Similarly, for the right child

$P(\eta_R) = 2 \cdot P(\eta) + 1$ *From* 1b

$= 2 \cdot (I(\eta) - 2^{L(\eta)}) + 1 = 2 \cdot I(\eta) + 1 - 2^{L(\eta)+1}$ *From* 2

$= I(\eta_R) - 2^{L(\eta)+1}$ *From* 3b

$= I(\eta_R) - 2^{L(\eta R)}$ *Per definition of child*

Thus, the general property 4.4.2 $P(\eta) = (I(\eta) - 2^{L(\eta)})$ holds.

□

Figure 47 graphically illustrates the property 4.4.2, highlighting the match between the indexes of the nodes and their paths.

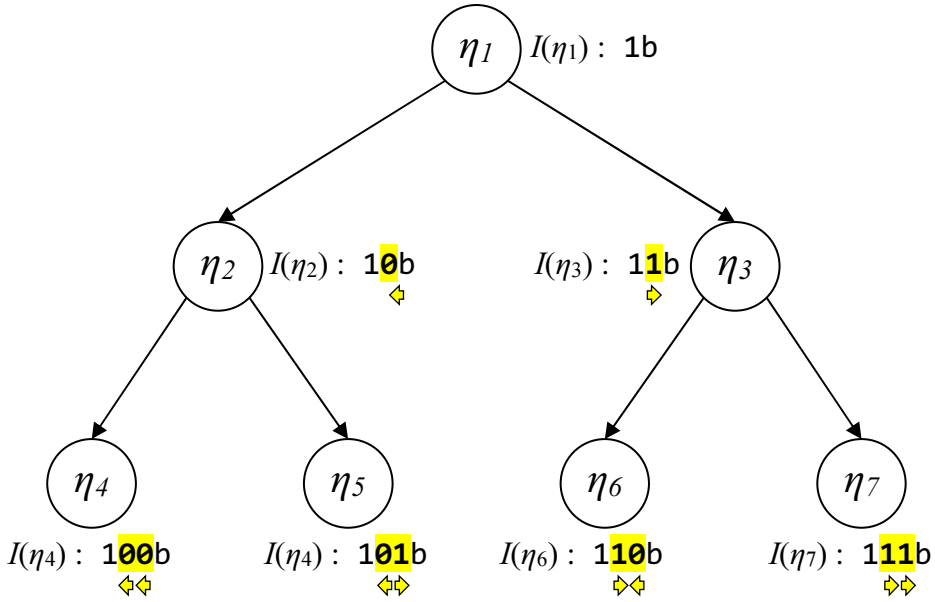


Figure 47: Graphical overview of the fundamental path-finding property that underpins the operations of the addressable binary heap.

Data structure definition

Once that the theoretical process that underlies the look-up of nodes from their index has been presented, the concrete data structure and the insertion/removal algorithms can be discussed in their details.

The concrete data structure of an ABH is defined as follows:

```
1. typedef struct
2. {
3.     abheap_node_t*     head;
4.     abheap_node_t      sentinel;
5.     size_t             count;
6. } abheap_t;
```

where *head* is a pointer to the root node descriptor (the one with highest priority) and *count* is an integer variable which keeps track of the number of nodes currently present in the ABH.

The data type associated to each node of the ABH is defined as follows:

```
1. typedef struct abheap_node
2. {
3.     abheap_prio_t       priority;
4.     struct abheap*      owner;
5.     unsigned long       position;
6.     struct abheap_node* left_child;
7.     struct abheap_node* right_child;
8. } abheap_node_t;
```

where *priority* represents the numeric priority of that node (*abheap_prio_t* is just a redefinition of the value type chosen to model the priority), *owner* is a back pointer to the owner ABH (only for diagnostic purposes, .e.g., breaking the debugger in case of double insertions or double removals), *position* represents the index $I(\eta)$ of that node (as defined in 4.4) and *left_child/right_child* the pointers to the node's left and right children.

Path extraction and navigation

As regards the navigation in the ABH, in the following the methods for extracting the path of a node from its index and navigating through it are presented, basically concretizing in C code the theoretical considerations previously made.

An ad-hoc data type has been defined for representing a path and allowing efficient navigation through it, as follows.

```
1. typedef struct
2. {
3.   unsigned long  bitmap;
4.   int            steps;
5. } path_t;
```

In such data type, *bitmap* is a machine word, in which each bit represents, MSB to LSB, the direction (0: left, 1: right) that must be taken when branching in each level of the ABH. The length of the *bitmap* word depends on the underlying architecture. In the most common cases it is a 32 or 64 bit word, thus allowing to keep the path for trees of up to 2^{32}-1 or 2^{64}-1 nodes.

However, depending on the actual number of nodes present in the ABH and the number of branches already taken when navigating along that path, the number of meaningful bits will vary. Per definition the *bitmap* is left aligned, i.e. its MSB represents the branch direction for the next level. In order to identify the number of meaningful most significant bits, a separate integer variable is introduced, here called *steps*. In practice this variable represents the number of levels of the ABH that still need to be traversed to reach the destination.

In the previous section it has been shown that the binary string representing the path of a node corresponds to the binary representation of the index, after its most significant high bit is stripped off. In order to identify and strip such bit we take advantage of a specific machine instruction which most computer architectures offer [5], herein referred to as

[5] In lack of that, the `CountLeadingZeros` function can be easily emulated in software.

CountLeadingZeros, that returns the position of the most significant high bit of a machine word (*bsr* on Intel x86, *lzcnt* on AMD, *clz* in ARM).

The corresponding C code is shown in the following.

```
1. path_t GetPathToNode(unsigned long node_index)
2. {
3.   path_t path;
4.   int leading_zeros    = CountLeadingZeros(node_index);
5.   const int ULONG_BITS = sizeof(node_index) * 8;
6.   /* The maximum operand allowed by C standards for the
7.    * << operator is the word size (64). Therefore the shift
      * operation must be split in two steps */
8.   path.bitmap = (node_index << (leading_zeros)) << 1;
9.   path.steps  = ULONG_BITS - leading_zeros - 1;
10.  return path;
11.}
```

The navigation of the path is absolutely straightforward. Basically it consists in a check of the *bitmap*'s MSB, a left-shift operating and a decrement of the number of remaining steps.

```
1. bool PathHasNext(const path_t* path)
2. {
3.   return path->steps > 0;
4. }
5.
6. path_dir_t PathMoveNext(path_t* path)
7. {
8.   path_dir_t dir = (MSB(path->bitmap)) ? RIGHT : LEFT;
9.   path->bitmap <<= 1;
10.  path->steps   -= 1;
11.  return dir;
12.}
```

Insertion algorithm

It is immediate to note that when a new node is being inserted in a ABH containing *count* nodes, in order to ensure the shape property a node (possibly the new node itself, if its priority is sufficiently low) will end up occupying the position at index *count* + 1.

Furthermore, in the light of the theoretical considerations made in 4.4, it is also evident that the knowledge of the final destination index gives also a precious information on the path that leads to its position.

A naïve insertion approach could consist in emulating the behavior of an array-backed binary heap, which is, first inserting the new node in the last position, following fully the path, and then percolating it up to restore heap property. While this approach is theoretically correct, it would require both an extra memory overhead, in order to keep track of the node parents, either storing them in the `abheap_node_t` structure or in a temporary stack, and a run-time overhead, since the tree should be traversed in its height two times, in both directions. Instead, the insertion algorithm envisaged traverses the ABH only once (top-down). In order to do so we need to introduce the concept of the *dangling node*.

Conceptually, the *dangling node* represents the potential competitor of each node that is encountered traversing the tree top-down along the path identified by the target position. At each level, such dangling node can turn out to have either a lower or equal priority than its current competitor, or a greater priority. In the former case, the traversal just continues to the next level, following the path without performing any modification to the ABH. Conversely, in the case that the dangling node has a higher priority than its current competitor the two nodes are swapped. It might be worth noting that swapping an ABH node η with another node η' (the dangling node) that has a higher priority than η but a lower priority than η^{-1}, preserves the heap property.

After a swap, the former dangling node assumes its final position at that level (thus the parent's pointer and its pointers are updated accordingly), while the former competitor is detached from the ABH and becomes the new dangling node.

This process continues until the last level of the tree is reached. There, the final dangling node, which in the meanwhile might have been swapped with lower priority nodes encountered along the path, is placed.

It can be noted as this algorithm traverses the tree only once in its height, and since, per definition, an ABH is always complete, the worst-case complexity of the algorithm is O(log(n)).

In the following, the C code that implements the insertion algorithm just described is shown.

```
1. void ABHeapInsert
2. (
3.     abheap_t*         heap,
4.     abheap_node_t*    node,
5.     abheap_prio_t     priority
6. )
7. {
8.   abheap_node_t*  dangling_node   = node;
9.   abheap_node_t** parent_ptr      = & heap->head;
10.  unsigned long   target_position = (heap->count + 1);
11.  path_t path    = GetPathToNode(target_position);
12.  node->owner    = heap;
13.  node->priority = priority;
14.
15.  while (PathHasNext(&path))
16.  {
17.    abheap_node_t* current = *parent_ptr;
18.    if (HasHigherPriority(dangling_node, current))
19.    {
20.      ReplaceNode(current, dangling_node, parent_ptr);
21.      SwapPointers(dangling_node, current);
22.    }
23.
24.    if (PathMoveNext(&path) == LEFT)
25.    {
26.      parent_ptr = & current->left_child;
27.    }
28.    else
29.    {
30.      parent_ptr  = & current->right_child;
31.    }
32.  }
33.
34.  dangling_node->position    = target_position;
35.  dangling_node->left_child  = & heap->sentinel;
36.  dangling_node->right_child = & heap->sentinel;
37.  *parent_ptr                = dangling_node;
38.  heap->count += 1;
39.}
```

Highest priority element removal algorithm

Let us consider first the case of removing the highest priority element (the root) from the ABH. Trivial cases, i.e. $count \leq 1$, are omitted.

Step 1: identification and removal of the last node

As in the case of the array-backed binary heap, a good approach for removing the root node and, at the same time, preserving the shape property is represented by replacing the root with the last node of the heap (the one with highest index). Since the ABH data structure is based on pointers, however, such replacement requires that the last node must be first identified and unlinked from the heap.

The identification process is, at this point, absolute straightforward and analogous to what discussed in the insertion case. The knowledge of the `count` of nodes directly reflects the index of the last node, thus its path. Furthermore, since the path to reach the last node must necessarily pass through its parent, the unlink process can be performed traversing the ABH in its height just once, keeping track in a temporary variable of the left/right child pointer of the last parent seen and invalidating it at the last iteration, as follows:

```
1. abheap_node_t* UnlinkLastNode(abheap_t* heap)
2. {
3.  abheap_node_t** parent_ptr = & heap->head;
4.  path_t          path       = GetPathToNode(heap->count);
5.  abheap_node_t*  last_node;
6.  while (PathHasNext(&path))
7.  {
8.    abheap_node_t* current = *parent_ptr;
9.    if (PathMoveNext(&path) == LEFT)
10.   {
11.     parent_ptr = & current->left_child;
12.   }
13.   else
14.   {
15.     parent_ptr = & current->right_child;
16.   }
17. }
18. last_node    = *parent_ptr;
19. *parent_ptr  = & heap->sentinel;
20. return last_node;
21.}
```

Step 2: restoration of the heap property

The pointer based physical structure of the ABH paves the way towards the restoration of the heap property in a similar way to what happens, at least from a logical viewpoint, in the array-backed binary heap during a standard percolate-down operation.

In the case of ABH, however, this operation can be performed in a lighter-weight fashion by means of just mangling link pointers, without actually swapping or physically moving the nodes themselves.

This *hole-propagation* approach, herein called *bubble-down*, consists in the following: the removal of the root node conceptually creates a hole in the tree that must properly filled, ensuring to preserve both the heap property and the shape property of the ABH.

From an algorithmic viewpoint, however, the node which causes the *hole* is not concretely removed. Instead, it is directly replaced by choosing among the highest of its two children and the last node of the ABH (which has been unlinked in the previous step and is temporary not part of the ABH). At this point two cases are possible: (i) one of the two children has the highest priority, so its replacement preserves the heap property but causes the hole to move downwards in the place of the swapped child. In this case the algorithm must be reiterated, in the worst case until the hole is pushed over the last level of the ABH; (ii) the last node is the one with the highest priority: this causes the termination of the algorithm, since its replacement gives back the complete shape to the ABH and restores the heap property.

It might be finally worth noting that the algorithm, as described so far, can be theoretically performed also on a portion of the ABH, i.e. starting from a non-root node. For such reason, in the following a generalized version, which takes as input parameters a generic start node (and the link of its parent), is presented:

```
1. void BubbleDown
2. (
3.   abheap_node_t*  hole,
4.   abheap_node_t** hole_parent_ptr,
5.   abheap_node_t*  replacement
```

```
6. )
7. {
8.   abheap_t*       heap          = hole->owner;
9.   abheap_node_t** parent_ptr    = hole_parent_ptr;
10.  unsigned long   hole_position = hole->position;
11.  abheap_node_t*  hole_left     = hole->left_child;
12.  abheap_node_t*  hole_right    = hole->right_child;
13.  bool            completed     = false;
14.  while (not completed)
15.  {
16.    if (HasHigherPriority(hole_left, hole_right))
17.    {
18.      if (HasHigherPriority(hole_left, replacement))
19.      { /* Pull-up hole's left child */
20.        *parent_ptr          = hole_left;
21.        parent_ptr           = & hole_left->left_child;
22.        hole_left->position /= 2;
23.        SWAP(hole_right, hole_left->right_child);
24.        hole_left            = hole_left->left_child;
25.        hole_position        = hole_position * 2;
26.      }
27.      else
28.      {
29.         completed = true;
30.      }
31.    }
32.    else
33.    {
34.      if (HasHigherPriority(hole_right, replacement))
35.      { /* Pull-up hole's right child */
36.        *parent_ptr          = hole_right;
37.        parent_ptr           = & hole_right->right_child;
38.        hole_right->position /= 2;
39.        SWAP(hole_left, hole_right->left_child);
40.        hole_right           = hole_right->right_child;
41.        hole_position        = hole_position * 2 + 1;
42.      }
43.      else
44.      {
45.        completed = true;
46.      }
47.    }
48.  }
49.  *parent_ptr              = replacement;
50.  replacement->position    = hole_position;
51.  replacement->left_child  = hole_left;
52.  replacement->right_child = hole_right;
53.}
```

In the light of the above considerations, the highest priority element removal routine is presented in its entirety below:

```
1. abheap_node_t* ABHeapRemoveHighest
2. (
3.   abheap_t* heap
4. )
5. {
6.   if(heap->count > 0)
7.   {
8.     abheap_node_t* const old_head = heap->head;
9.     abheap_node_t* last_node = UnlinkLastNode(heap);
10.    BubbleDown(old_head, & heap->head, last_node);
11.    old_head->owner = NULL;
12.    heap->count--;
13.    return old_head;
14.  }
15.  else
16.  {
17.    return NULL;
18.  }
19.}
```

It can be finally noted that, since both the *UnlinkLastNode* and the *BubbleDown* traverse the ABH in its height one time each, the worst-case complexity of the *ABHeapRemoveHighest* routine is O(log(*n*)).

Arbitrary element removal algorithm

The greatest advantage introduced by ABH, over the array-backed binary heap, is the direct addressability of its elements. Direct addressability paves the way towards the efficient implementation of a further operation, fundamental for our timekeeping purposes, that is, the removal of arbitrary nodes from the ABH.

In the light of the previous discussions, the removal of a node creates a hole that can be *bubbled-down*. The choice of the initial replacement node, however, requires a more careful handling in this scenario. In fact, no assumption can be made on the relationship that exists between the priority of an arbitrary node and priority of the last ABH node (except for the singular cases in which the node being removed is an ancestor of the last node).

Thus, due to this intrinsically weak ordering, the last node won't necessarily ensure the heap property to be preserved in the first *L*(*hole*) levels of the ABH and could turn out to have a higher priority than some ancestors of the node being removed. Figure 48 graphically depicts such scenario.

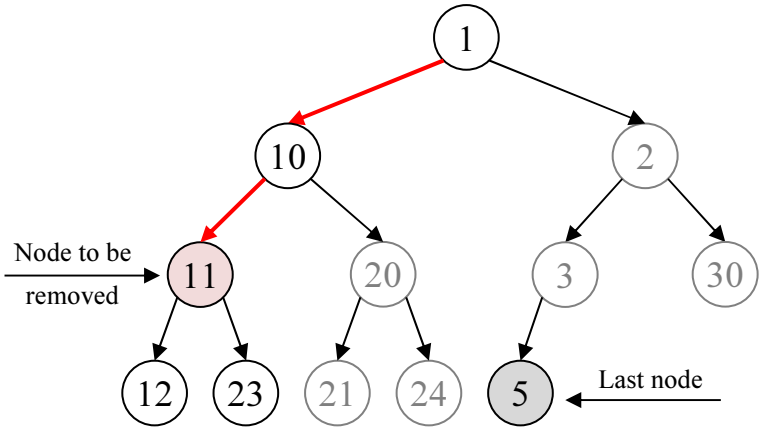


Figure 48: A particular case of arbitary element removal in an addresable binary heap.

In order to ensure the global satisfaction of the heap property, a preliminary filtering step is required: the last node must be filtered down on the path that leads from the root to the node being removed, and swapped every time a lower priority node is encountered, in order to re-establish the heap property in the first levels of the ABH.

Thus, at the end of this partial walk, the replacement node employed for the `BubbleDown` routine will be the lowest priority node encountered along the path. For instance, in the example of 8, the last node of priority (5) would be swapped with the node (10), and the latter would be used to replace the hole of the node (11) in the `BubbleDown` call.

From a practical viewpoint, the logic of the filtering step is exactly the same of the insertion algorithm, in which the last node represents the initial *dangling node*.

The complete code for removal of an arbitrary node is presented below:

```
1. void ABHeapRemove
2. (
3.   abheap_node_t* node_to_remove
4. )
5. {
6.   abheap_t*        heap       = node_to_remove->owner;
7.   abheap_node_t**  parent_ptr = & heap->head;
8.   abheap_node_t*   dangling_node;
9.   path_t path = GetPathToNode(node_to_remove->position);
10.
11.  dangling_node = UnlinkLastNode(heap);
12.
13.  if (dangling_node != node_to_remove)
14.  {
15.    while(PathHasNext(&path))
16.    {
17.      abheap_node_t* current = *parent_ptr;
18.
19.      if(HasHigherPriority(dangling_node, current))
20.      {
21.        ReplaceNode(current, dangling_node, parent_ptr);
22.        SwapPointers(dangling_node, current);
23.      }
24.
25.      if(PathMoveNext(&path) == PATH_DIR_LEFT)
26.      {
27.        parent_ptr = & current->left_child;
28.      }
29.      else
30.      {
31.        parent_ptr = & current->right_child;
32.      }
33.    }
34.    /* At this point
35.    - dangling_node is either the last_node or a node
36.      with lower priority encountered along the path
37.     (from head to node_to_be_removed).
38.    - parent_ptr points to node_to_be_removed, unless it
39.      was exactly the last_node. In such case parent_ptr
40.      points to the sentinel,since node_to_be_removed was
41.      detached by UnlinkLastNode.
42.    - node_to_be_removed has not been removed yet, but we
43.      ready to do it now. */
44.   BubbleDown(node_to_remove, parent_ptr, dangling_node);
45.  }
46.  node_to_remove->owner = NULL;
47.  heap->count--;
48.}
```

It might be noted that this routine walks the heap in its height two times: the first time to identify and unlink the last node, and the second time to filter down the last node (lines 13-33) and then to bubble-down the resulting dangling node (line 44) in the remaining levels of the ABH. Thus its worst-case run-time complexity is O(log(*n*)).

As a final note, it might be noted that, as in the case of the array-backed binary heap, also the ABH is a non-stable model, i.e. the order of two elements having the same priority is not guaranteed to be preserved. Whereas stability represents a concern, however, non-stable priority queues can be made stable by the introduction of an auxiliary priority field, a sequence number, to break ties as discussed by McCormack and Sargent in [MS1981].

4.5. Experimental evaluations

The ABH, as presented so far, allows to implement all the operations envisaged by a priority queue, including the removal of arbitrary nodes, with a logarithmic worst-case complexity and a linear memory complexity. It might be worth verifying, at this point, how the presented implementation of such data structure performs when applied to the timekeeping problem.

Performance evaluation involves three main points, which are being discussed separately in the following: (i) a methodological aspect, i.e. what operations should be evaluated and under which scenarios; (ii) a comparative aspect: which other data structure to compare; (iii) a practical aspect: how to concretely measure performances, mostly related to the underlying software and hardware platform.

Evaluation methodology

Over the years, several performance studies has been carried on priority queues, mostly in the context of discrete event simulation (DES). In such context, a priority queue is generally used to hold the pending event set (sometimes referred to as the event calendar [CSR1993]) which contains the scheduled future events. The DES scheduling problem, however, is, in its essence, a tick-less timekeeping problem, since DES

simulators basically rely on a set of timers which are scheduled to trigger future actions. Run-time performances of timekeeping in DES are as critical as in RTOS schedulers since, as shown by an empirical study by Comfort [Com1984], up to 40% of the DES execution time may be spent on the event-set management (thus on timekeeping routines) itself.

Similarly to [RA1997], synthetic experiments are preferred over real simulations since they provide better control over the variables affecting performance and, thus, they better expose the factors that influence performance. Furthermore, synthetic experiments facilitate direct comparison to earlier priority queue studies [CSR1993, MS1981, VD1975].

In this regard, a widely used method for performance evaluations is represented by the insert-hold model, introduced by Vaucher and Duval in [VD1975] and refined by Jones [Jon1986]. It models operations on a fixed-size queue where a series of hold operations (a removal followed by an insertion) are performed. In [RAFD1993] Rönngren et al. highlight as this methodology is in general not sufficient to capture the dynamic nature of queue sizes that often appears in practice, as recognized by several researchers as in [CSR1993].

An Up/Down model is proposed by Rönngren et al. [RA1993], where a sequence of insertions is followed by an equally long sequence of removals. Further refinements of these models have been presented in the scientific literature. For instance Chung et al. [CSR1993] propose a generalization of the Hold model, the Markov Hold, where operations on the queue are determined by a two-state Markov process.

However in [RA1997] Rönngren et al., after a long series of comparative evaluations, highlight as, when the queue size remains nearly constant, the classic Hold model gives as accurate and informative results as the more random access patterns generated by the Markov Hold. Furthermore, for changing queue sizes, the simple Up/Down model often gives sufficient information. In general, the simplicity of the classic Hold and the Up/Down models seems to reveal more and clearer information on the dependencies of priority increment distributions and queue sizes on the performance of the queue.

In the light of the above considerations, the evaluation methodology adopted in this work is the following: 19 base experiments evaluate and compare the run-time performances of the timekeeping operations on several data structures at varying sizes S_i of the timer queues (S_1=10 to S_{10}=100 in steps of 10, to S_{19}=1000 in steps of 100). In each base experiment, a number of S_i *up*-insertions are performed (the queue is gradually populated). Then another set of S_i *hold* operations are performed (a maximum priority removal and an insertion each) using uniform distributions for the random generation of new timers' expiration. Finally the queue is emptied with a S_i *down* operations, this time performing random removals of arbitrary nodes. Each of those base-experiments is repeated 50 times, varying the random seed that generates the priorities for insertions (both for *up* and *hold*) and selects the random nodes to be removed during the *down* removal operation.

Comparative evaluation

The ABH has been compared, in this work, with the following data structure implementations:

RBT: Red-black tree is one of the most famous and widely used variant of self-balancing binary trees. The Linux Kernel implementation of red-black trees has been herein chosen due to its popularity and its ability to being use standalone in other context than the Linux kernel, without requiring many external dependencies. Like the ABH, the Linux RBT implementation is based on an embedded-anchor model, which is very popular in embedded systems. Conversely to what happens in higher level frameworks, such as most Java or C++ STL containers, the embedded-anchor model provides that the data type of the objects being added to the data structure is aware of the container and explicitly define an anchor field, which contains the child pointers and the other relevant fields required by the data structure. Although this model may seem, at a first glance, to go against the cornerstones of software engineering principles, it has the great advantage of not requiring any dynamic memory allocation for the operations of the container. Therefore, if the nodes that are added and removed at runtime are known a priori (or at least an upper bound on their number), they can be statically allocated during the binary

initialization phase and their handling in the data structure can be performed without the intervention of dynamic memory management (often banned in military and high reliability systems, as in the case of DO-178B level A profiles).

T1H, **T1K:** Timing wheel open source library [eST2009], which implement the hashed variant of the timing wheel data structure discussed in 4.3. Such data structure exhibits, from an analytic viewpoint, a O(1) worst-case complexity for insertions and random removals and hashtable-like O(n/N) average and O(n) worst-case complexity for highest priority removals. As in the case of ABH and RBT, the timing wheel implementation also relies on the embedded-anchor model, so the performance measurements do not take into account any time required for allocation of their nodes. Two instances of the timing wheel, respectively of 100 (T1H) and 1000 (T1K) buckets, have been considered in the evaluation.

MLS: C++ STL Multiset. The C++ Standard Template Library (STL) introduces the *set* and *multiset* containers. In the HP's STL implementation, the foundation of these classes is a red-black tree, and, like Linux RBT, it supports insertion, removals and highest-priority removal in O(log(*n*)) worst-case complexity. However, conversely to Linux embedded-anchor RBT, this STL container takes implicitly advantage, in its internals, of dynamic memory management to wrap the elements of the set, so its runtime performances can differ from RBT.

BHP: Traditional array-backed binary heap implementation, which is based on a conventional C array and on the `push_heap` and `pop_heap` C++ STL methods defined in the `<algorithm>` header. Conversely to the ABH, RBT, T1H/K, and MLS, which basically rely on pointer-mangling operations and don't actually move the data nodes, the operations of the BHP, as described in 4.3, do physically move and swap nodes in order to keep the heap consistent. Since the runtime performances of the move/swap depend on the actual size of the timer descriptor, in order to

carry out a fair comparison which reflects real-world timekeeping implementations, the elements of the array have been defined as a tuple embedding the absolute expiration time of the timer, the pointer to the callback function invoked on timer expiration and the argument passed to that callback.

```
1. struct STLNode
2. {
3.     uint64_t      abs_expiration_time;
4.     void*         callback_fn;
5.     void*         callback_data;
6. }
```

The measurements do not take into account the time required for the allocation and the initialization of the array during the experiments. Random element removal measurements are not available for BHP, since such primitive is not envisaged by the C++ STL, and in general would require a O(*n*) complexity, in order to look-up the element to remove.

PQU: C++ STL *priority_queue* container. The C++ STL introduce the *priority_queue* adapter container. In the default HP's STL implementation, it models a priority queue over a STL vector, thus incurring in dynamic allocations and, more importantly, in memory relocations as the size of the queue grows up. As for BHP, the interface of the STL *priority_queue* envisages only methods for insertion and highest priority removal. Thus measurements for random element removals are not available for PQU.

Evaluation platform

The scenario being investigated consists in the aforementioned data structures being used to address the timekeeping problem in real-time systems. For such reason, an important aspect of the performance evaluations is the worst-case execution time measurement.

In real-world systems, however, measuring the actual worst-case execution time of a software algorithm is not trivial. The major issues are represented by the wide spectrum of noise, due to both hardware (e.g., peripherals

triggering interrupt requests, presence of other bus-master devices slowing down CPU accesses, dynamic CPU frequency scaling) and software activities (higher priority processes and kernel threads which may preempt the experiments), which are hard to control. This problem, of course, could be easily overcome by means of filtering out outliers from the collected samples. This kind of solution, however, is not suitable at all for our scenarios, since it would inevitably filter out not only the measurement noise but also potential peaks related to the nature of the data structures being investigated.

For such reason, the experimental evaluations have been carried out on a hardware platform simulation infrastructure called OVPSim [OVP2012].

OVPSim (namely open virtual platform simulator) is a broadly diffused open-source simulation platform, able to model a wide variety of computational architectures typically employed in embedded systems, such as the ARM, MIPS and OpenRisc processors, with instruction-level accuracy. The simulation engine is fully customizable and gives the possibility of setting up ad-hoc virtual platforms, choosing arbitrary CPU and memory layouts to best fit the simulation needs. The simulator takes advantage of just-in-time code morphing, translating dynamically target instructions to x86 host instructions. OVPsim has been specifically architected for the fast and accurate simulation and includes many optimizations enabling simulation of platforms utilizing many homogeneous and heterogeneous processors with many complex memory hierarchies.

The simulation platform gives the complete control on the virtual memory initialization, allowing to easily load custom binaries in the virtual platform memory before booting the virtual cores. Furthermore OVPSim libraries support semi-hosting for many peripherals, allowing to redirect the C/C++ standard library I/O of the target system to the host simulator. For such reasons, the software experiments can be run bare on the target virtual platform without requiring any additional driver or any operating system to be loaded.

The ARM-CM3 and the MIPS 32 virtual processor models have been chosen for the experimental evaluations, as representatives of a large class of realistic real-world scenarios.

The software experiments together with the ABH implementation, have been compiled with Mentor Graphic Sourcery CodeBench Lite GCC-based toolchains (more in detail GCC 4.6.3 for mips32-sde, and GCC 4.5.2 for armv7), enabling all compiler optimizations (-O3 switch).

Experimental results

The first set of measurement compares the average and maximum execution time (in terms of emulated machine instructions) taken by each data structure for each of the four operations previously described in a set of 50 repetitions per experiment. Thus, each point of the plots represents, respectively, the average and maximum values over a set of 50 * N (number of timers in the experiment on the x-axis) samples.

For sake of graphical intelligibility, the x-axis of the plots follows a double-linear scale with a discontinuity on N=100 (emphasized by a vertical dashed line).

Insert-up

Average ramp-up insertion times (Figure 49 and Figure 51) show that, as expected, both T1H and T1K keep, in any scenario, a perfect constant behavior regardless the length of the queue. In general, data structures based on a binary heap (in particular ABH, BHP) outperform both red-black tree implementations (RBT and MLS). The BHP exhibits the best average performances on both platforms: its array-backed physical structure allows to directly walk the heap in height very quickly with direct indexed memory access. A higher cost, instead, is paid by the tree-based data structures such as ABH, RBT and MLS. However, it might be noted as the ABH outperforms both the RBT and MLS red-black tree implementations in the ARM platform.

Furthermore, it might be noted as embedded-anchor tree models (ABH and RBT) exhibit better performances than the dynamic one MLS, which evidently suffers the run-time overhead due to the dynamic allocation of

STL container wrappers. A final note goes on the definitely odd behavior of the PQU, that finds a proper explanation on the unproportionatelly high overheads that are incurred when the underlying `Vector` grows and consequently performs dynamic memory reallocations. Since the occurrences of such reallocations are rare, their high cost is better amortized in the average value as the size of the queue increases, that explains its unexpected decreasing trend.

The situation becomes, however, more interesting when worst-case run-times are taken into account (Figure 50 and Figure 52). Unsurprisingly, the worst-case run-time for the BHP (that is, a node with a high priority being inserted when the heap is almost full, causing the percolate-up to deep-swap the contents of $\log_2(N)$ elements of the array) lifts-up, reaching an almost perfect overlap with ABH in the case of ARM-CM3, and becoming even worse, in the case of MIPS 32. Furthermore it can be noted as in both platforms the ABH largely outperforms both RBT and MLS. The worst-case plot for the PQU has been omitted for keeping the figures more readable. In both platforms, in fact, the growths and reallocation of the underlying `Vector` caused the worst-case samples to have peaks of an order of magnitude higher than the other data structures (which would have required to shrink the y-axis too much).

Insert-hold

Similar considerations apply for insert-hold measurements. The only noticeable difference, in this case, is that both average execution times (Figure 53 and Figure 55) and maximum execution times (Figure 52 and Figure 54), show the PQU backing-up to reasonable values close to the BHP behavior. The reason is that, conversely to what happens during ramp-up insertions, hold insertions keep the size of the queue constant, thus no dynamic expansions of the underlying Vector are needed.

Remove head

Average measurements for maximum-priority removals (Figure 57 and Figure 59) highlight as the runtime behavior of the timing wheels, as expected, is definitely not suitable for addressing tick-less timekeeping

problems. Clearly, for both timing wheels, the time spent looping for finding the next element represents the highest contribution to their run-time behavior, which is order of magnitudes higher than all other data structures (for this reason the charts for head-removal measurements use a logarithmic y axis). Curiously, both in the average and maximum measurements, the run-time behavior of both timing-wheels improve as the length of the queues increases. The reason of this lies in the uniformly random distribution of samples in the experiments, which is such that in larger queues the next element is statistically closer (i.e. requires shorter loops) than in smaller queues.

As regards the other data structures, both red-black trees implementations exhibit an almost O(1) amortized runtime performance. Both the RBT and MLS red-black trees tend to outperform binary heaps as the length of the queue increases

Also in this case, however, the RBT demonstrates to be a more efficient implementation than MLS, which, while keeping the same trend, pays a higher constant overhead due to the STL container wrappers and comparison callbacks.

ABH shows a very interesting behavior here, even in the average measurements, outperforming the other two BHP and PQU binary heap implementations in the MIPS 32 platform, and performing as the BHP in the ARM CM3 platform.

The most interesting results, however, come out when analyzing worst-case performances (Figure 58 and Figure 60). First of all, it can be noted as the worst-case behavior of ABH closely matches its average performances. Both the RBT and MLS red-black trees, instead, tend to have slightly worse worst-case behaviors than binary heaps (ABH, BHP and PQU). Besides, in both platforms, the ABH shows a remarkable behavior, outperforming all the others data structures (except the timing wheels when the queues length is small)

Remove random

As expected both T1H and T1K timing wheels keep a O(1) constant runtime for removal of random elements in all cases, due to the doubly

linked arrangement of the bucket lists. On average (Figure 61 and Figure 63) the performances of the ABH are better than the RBT and MLS red-black trees only when the length of the queue is modest (under 100). In general RBT has better average case performance, with MLS following its trend with its usual constant overhead offset.

However, when worst-case performances are take into account (Figure 62 and Figure 64), ABH again shows its tendency to keep very close worst and average runtime cases, outperforming MLS and keeping comparable worst-case performances than RBT on both platforms.

Run-time distributions

In order to get a broader level view that captures not only the average and maximum run times, but gives an overall graphical indication of the statistical distribution of the samples, Figure 65…Figure 72 show the violin plots [HN1998] for the insert and removal operations on the binary heap and red-black tree data structures, which all share a theoretical O(log(*n*)) worst-case runtime complexity. Each plot compares, for each operation, the statistical distribution, on both architectures, of the samples collected from the synthetic experiment involving a queue of N=500 timers. The bold rectangle inside the violins represent the inter quartile range (IQR), and the white dot marks the mean value.

During a ramp-up insertion (Figure 65 and Figure 66), the boundaries of the ABH distribution follow strictly the BHP. However, while the BHP samples are more dense around the 1st and 2nd quartiles, denoting a better average behavior, the ABH samples are almost exactly centered around their median. Furthermore these two plots make evident as, during an insertion, the RBT and MLS don't perform better than binary trees in any architecture, while PQU (which violin plot has been trimmed in its upper part for graphical reasons) confirms to suffer both longer execution times in average, and very high, yet sporadic, peaks.

During insert-hold insertions (Figure 67 and Figure 68), the distributions of ABH, BHP and PQU highlight that all the heap-based implementations tend behave as in their best-case behavior for most of the samples, and their worst-case behaviors are always better than RBT and MLS. Furthermore, ABH has worse average and minimum times than BHP and PQU. On the other side, its worst-case behavior is slightly better than the other two, especially in the MIPS 32 platform.

When highest priority removal is taken into account, Figure 69 and Figure 70 show a very interesting situation. RBT and MLS red-black trees exhibit better overall better performances. Their samples distribute on a lower but wider range, compared to binary heaps (in particular PQU and BHP), exhibiting an excursion between best and worst execution times of, respectively, 225 and 189 instructions on ARM CM3, and 239 and 180 instructions on MIPS 32. BHP and PQU, conversely, keep a narrower

range (112 and 106 instructions on ARM CM3 and 115 and 178 instructions on MIPS 32), but a higher mean and worst-case value.

The most interesting trend, however, is exhibited by ABH. While keeping a similar mean and range than BHP and PQU, its worst-case execution time is, in both platforms, smaller than all the other ones, including the red-black trees which have a sensibly smaller mean value. The shape of the two ABH violin plots give a reasonable view of this behavior, showing as, conversely to what happens in all other cases, on both platforms the ABH samples density is highly concentrated around the worst-case value. In summary, ABH demonstrate to require the worst-case execution time in most of the removals, but its worst-case is the best among all the data structures considered, in both platforms.

A similar situation is observed during random element removals (Figure 71 and Figure 72).

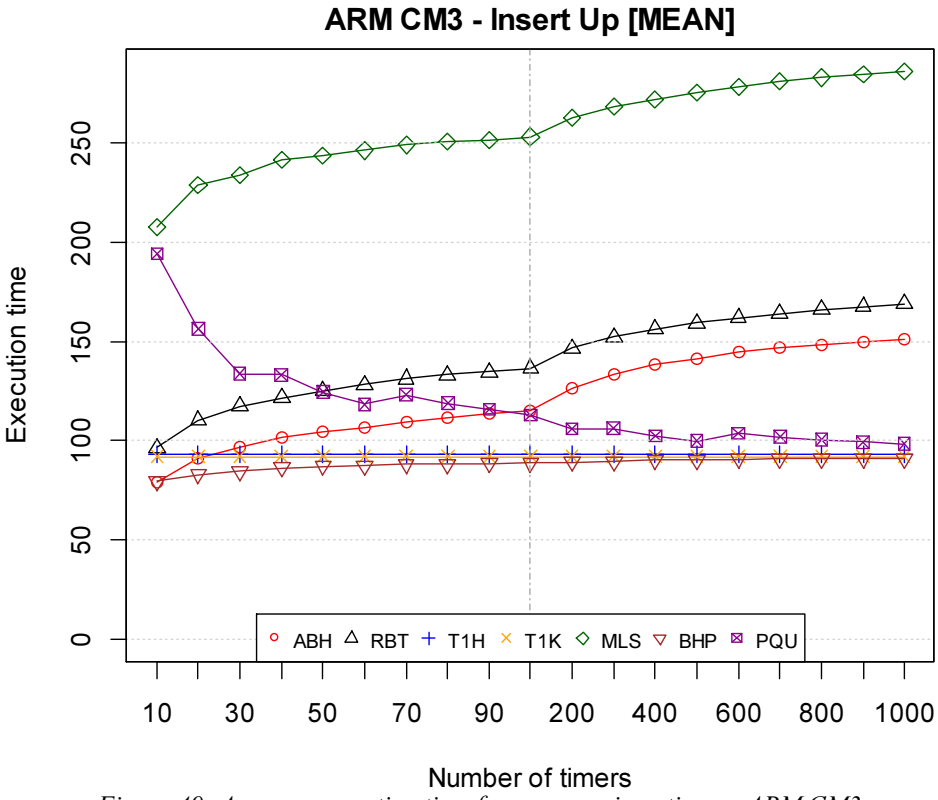


Figure 49: Average execution time for ramp-up insertion on ARM CM3.

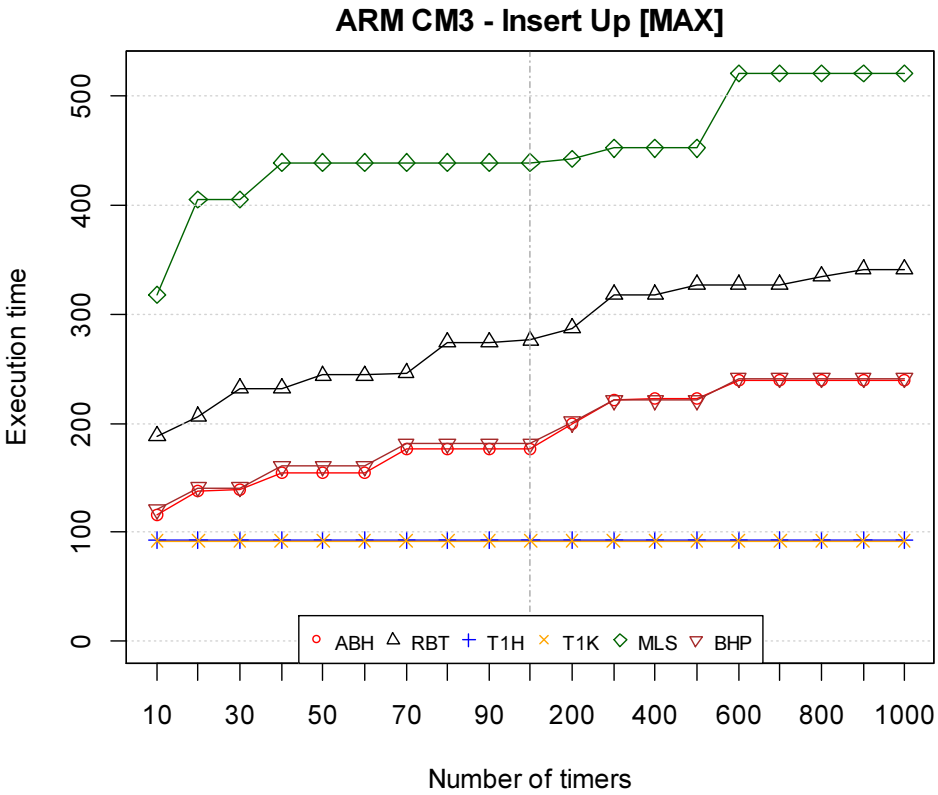


Figure 50: Maximum execution time for ramp-up insertion on ARM CM3.

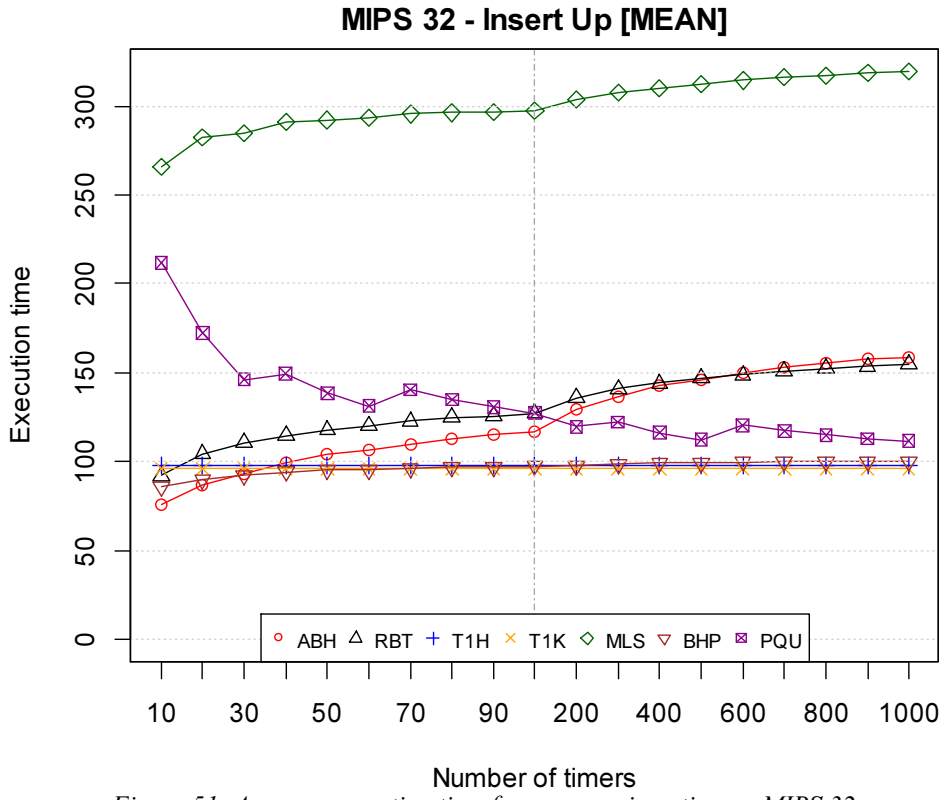


Figure 51: Average execution time for ramp-up insertion on MIPS 32.

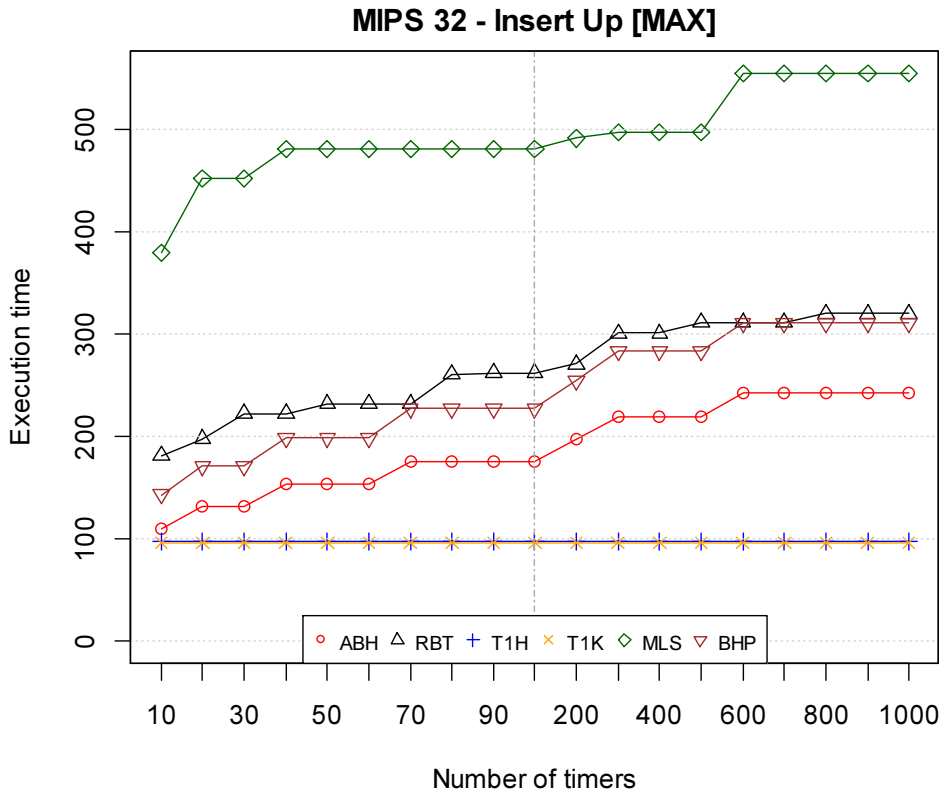


Figure 52: Maximum execution time for ramp-up insertion on ARM CM3.

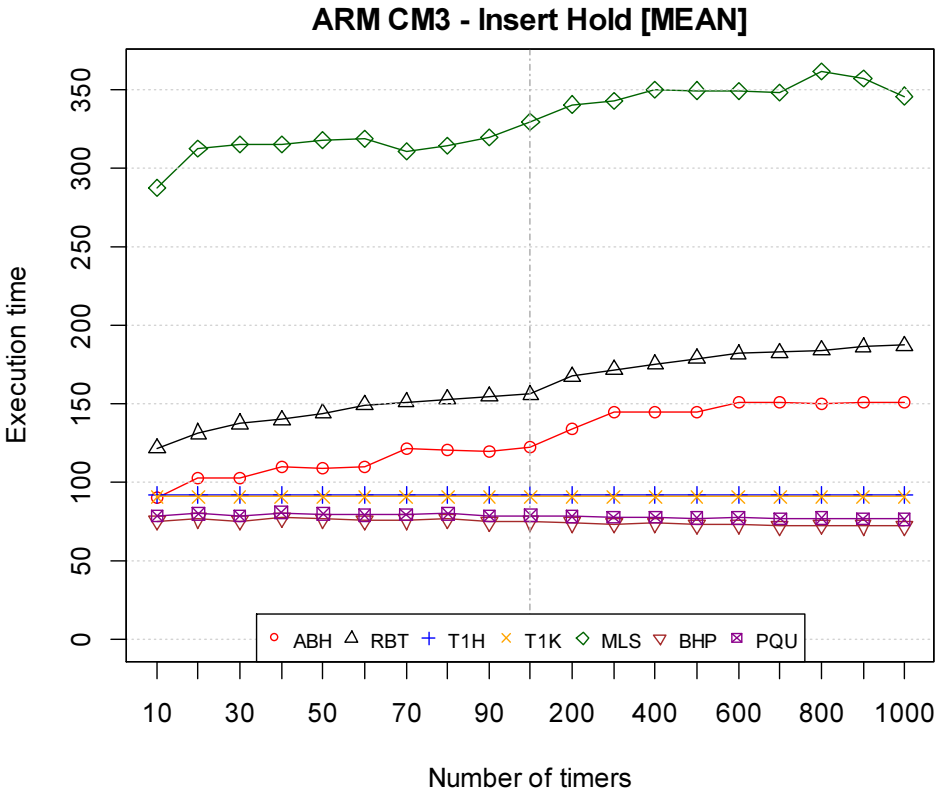


Figure 53: Average execution time for hold insertion on ARM CM3.

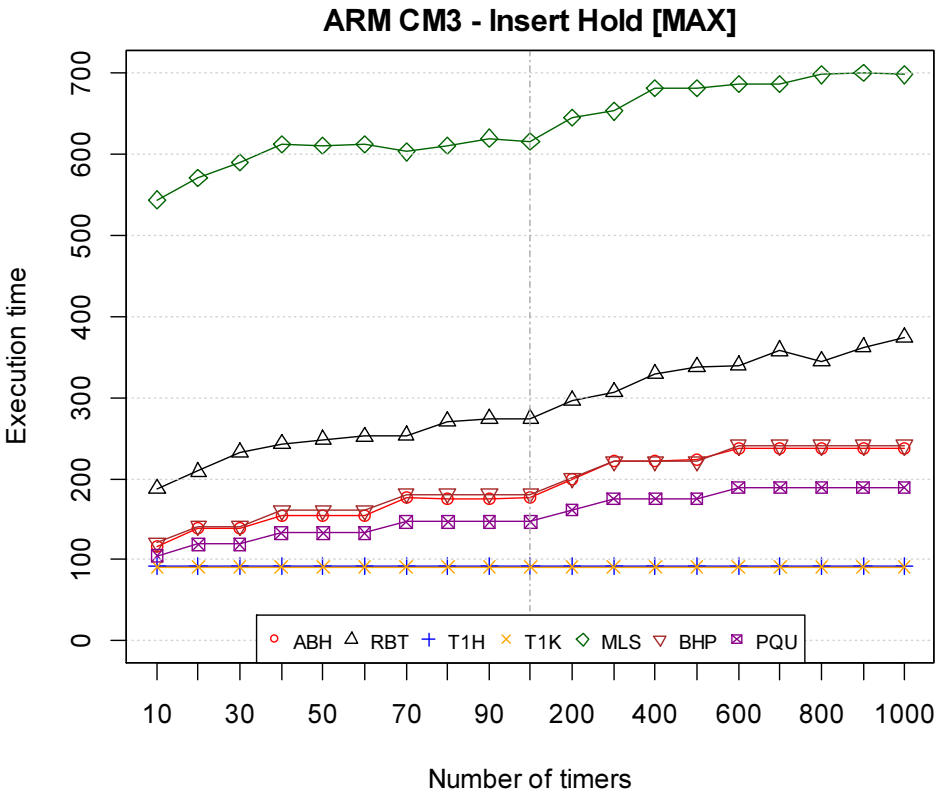


Figure 54: Maximum execution time for hold insertion on ARM CM3.

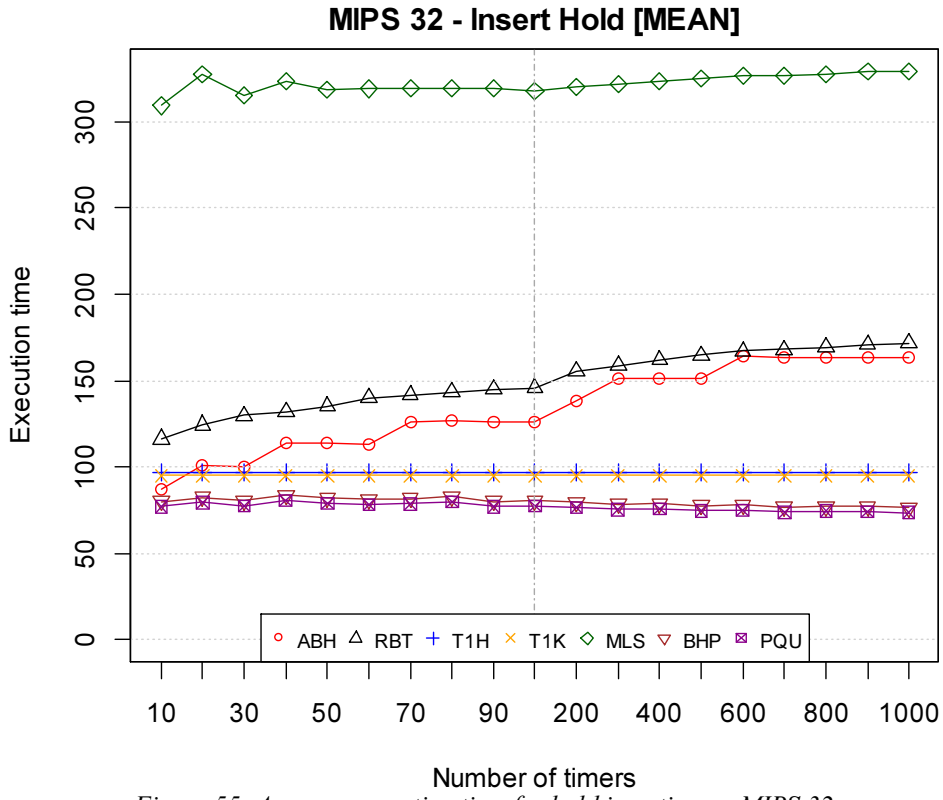


Figure 55: Average execution time for hold insertion on MIPS 32.

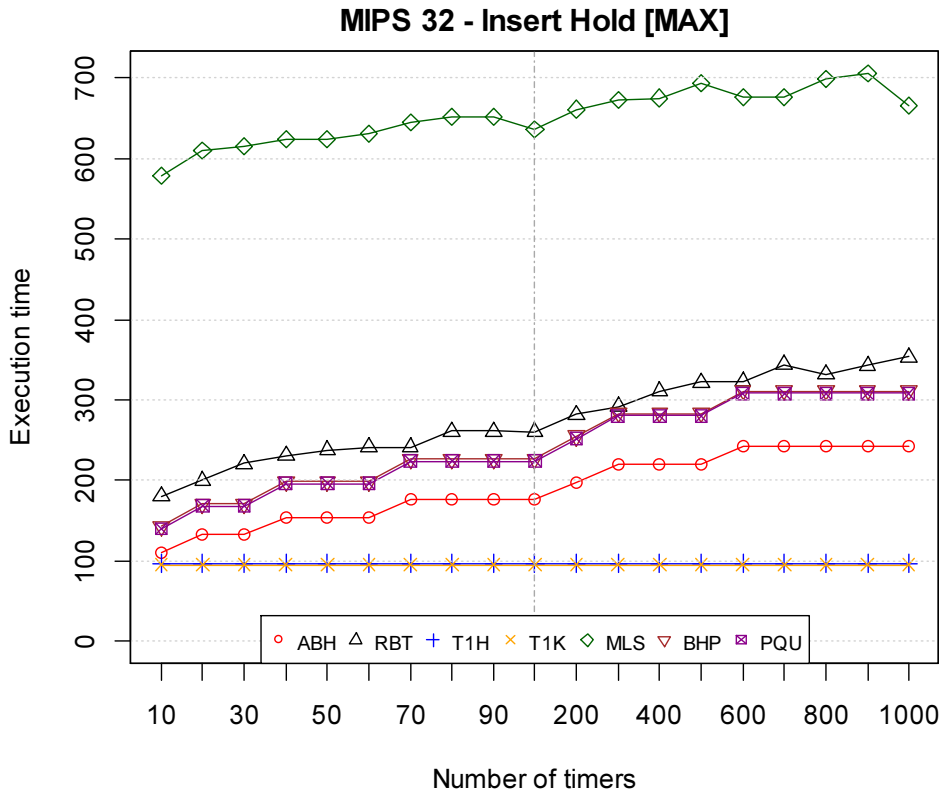


Figure 56: Maximum execution time for hold insertion on MIPS 32.

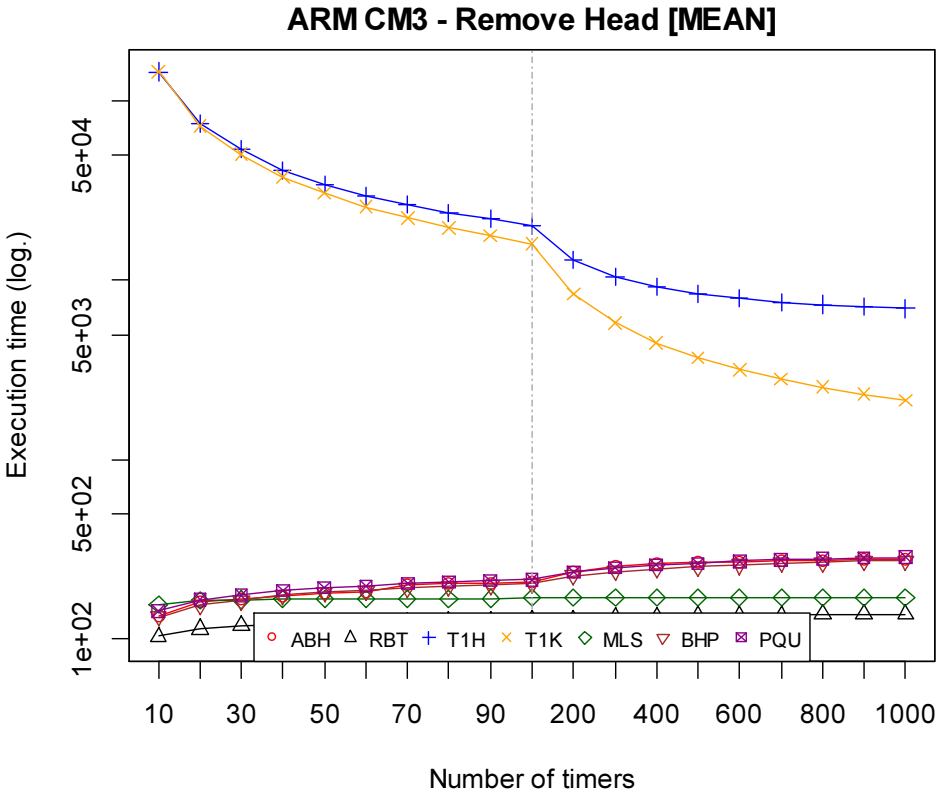


Figure 57: Average execution time for highest priority removal on ARM CM3.

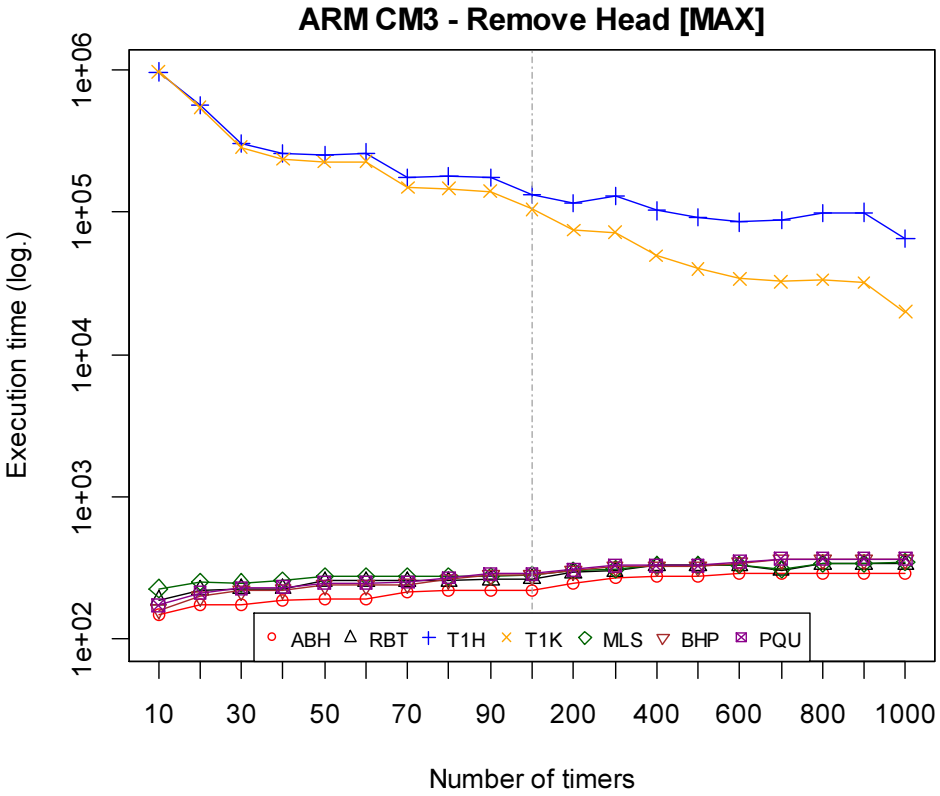


Figure 58: Maximum execution time for highest priority removal on ARM CM3.

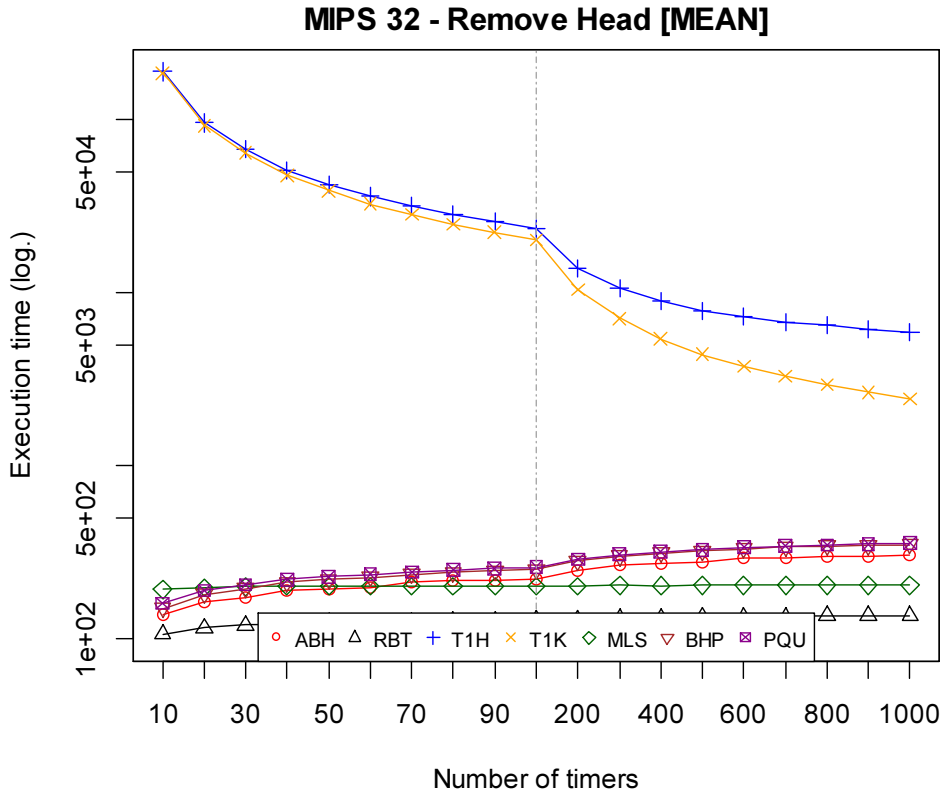


Figure 59: Average execution time for highest priority removal on MIPS 32.

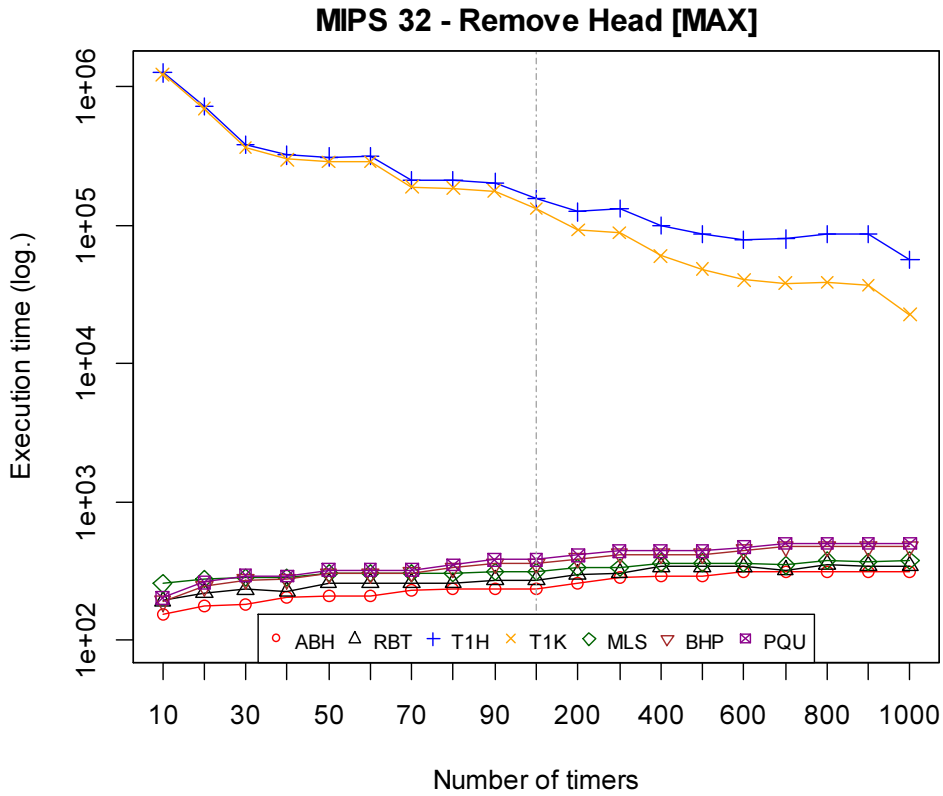


Figure 60: Maximum execution time for highest priority removal on MIPS 32.

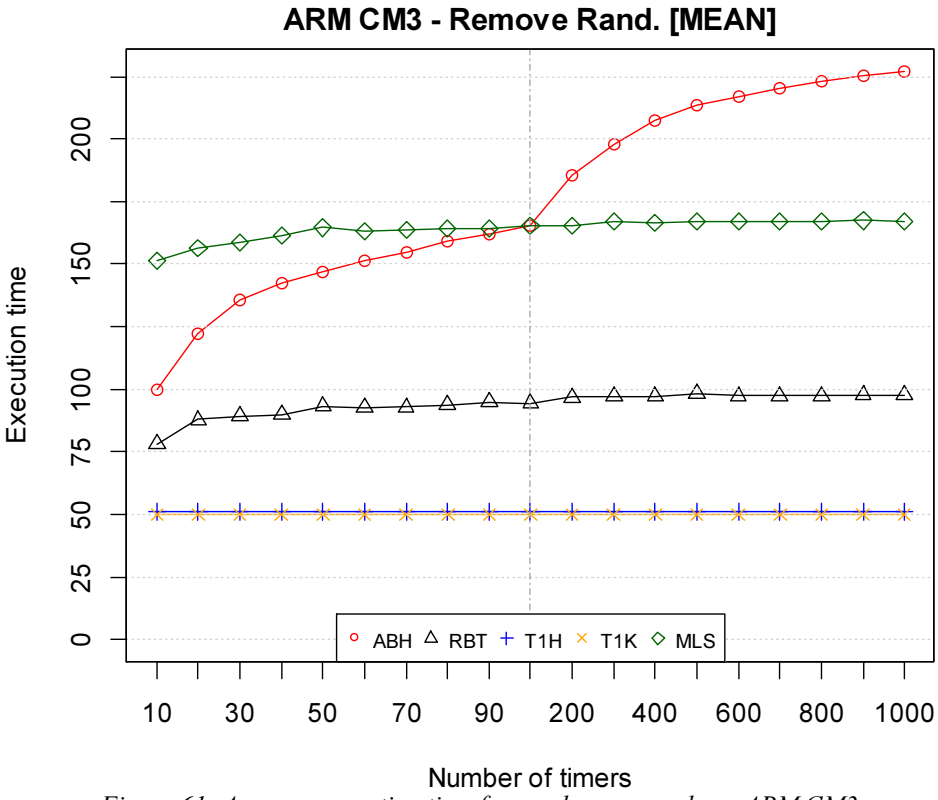


Figure 61: Average execution time for random removals on ARM CM3.

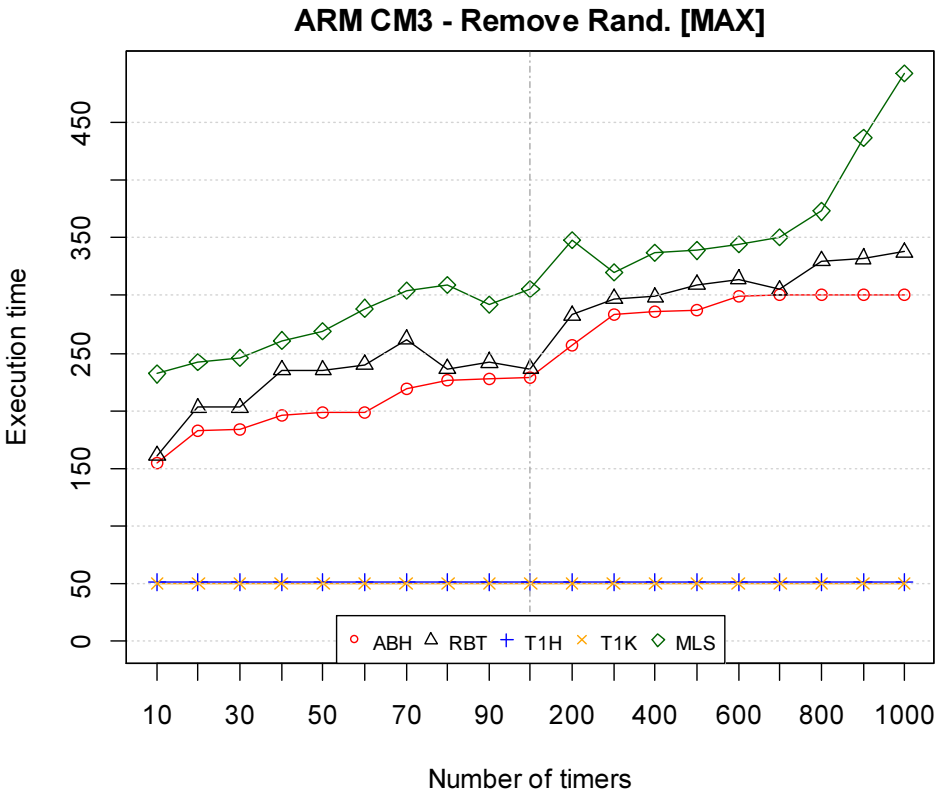


Figure 62: Maximum execution time for random removals on ARM CM3.

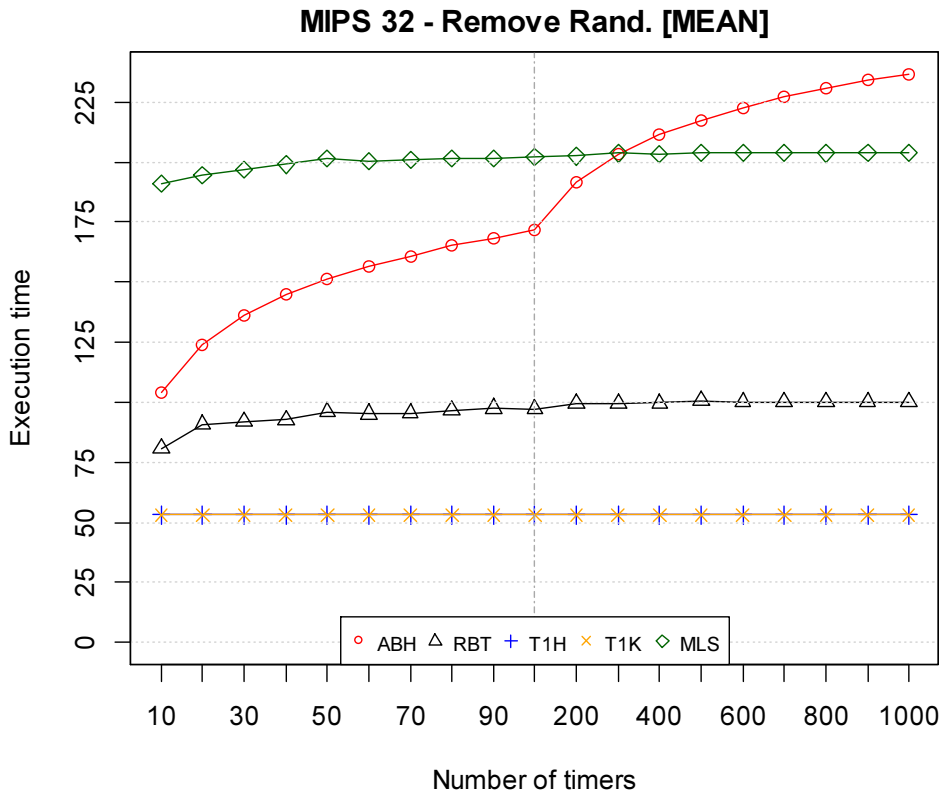


Figure 63: Average execution time for random removals on MIPS 32.

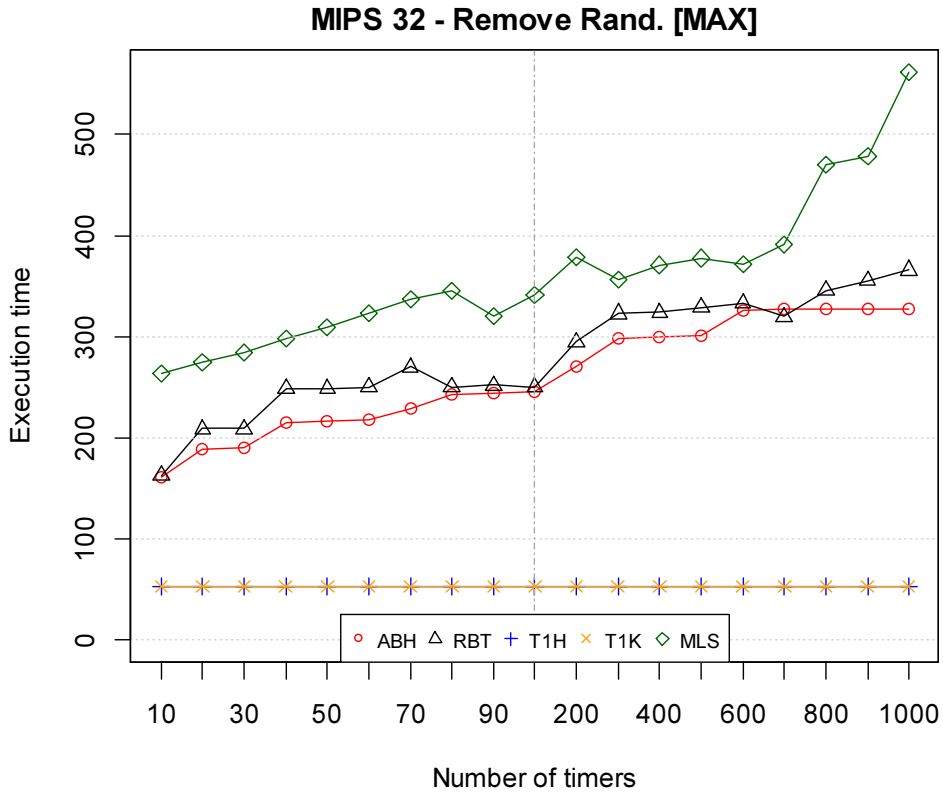


Figure 64: Maximum execution time for random removals on MIPS 32.

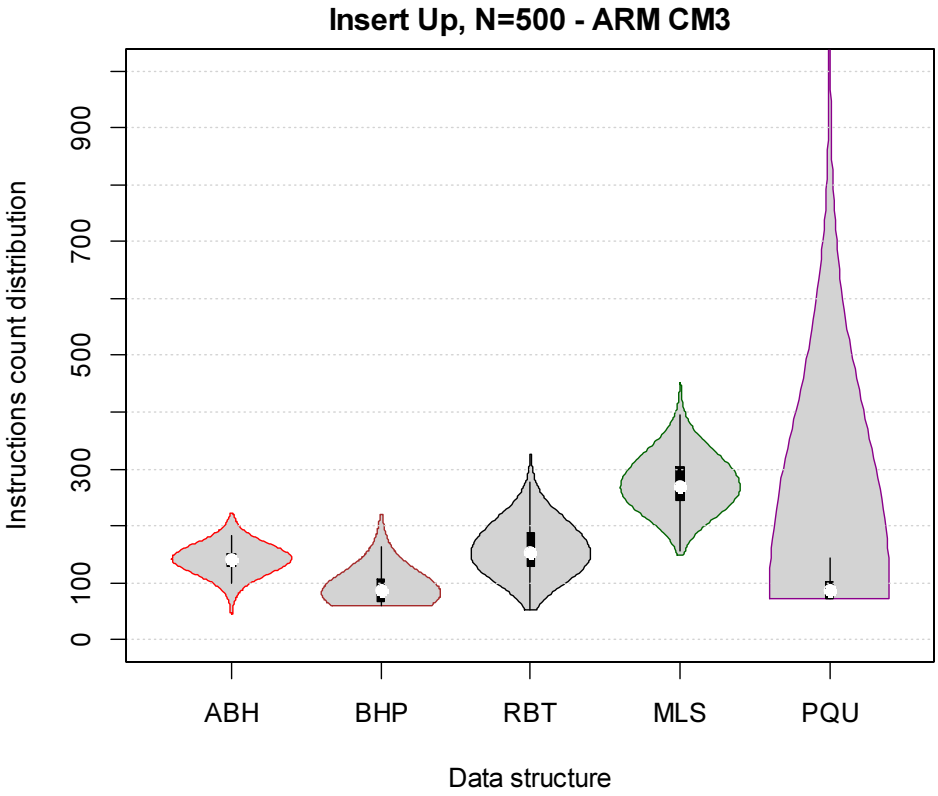


Figure 65: Execution time distribution for ramp-up insertion on ARM CM3

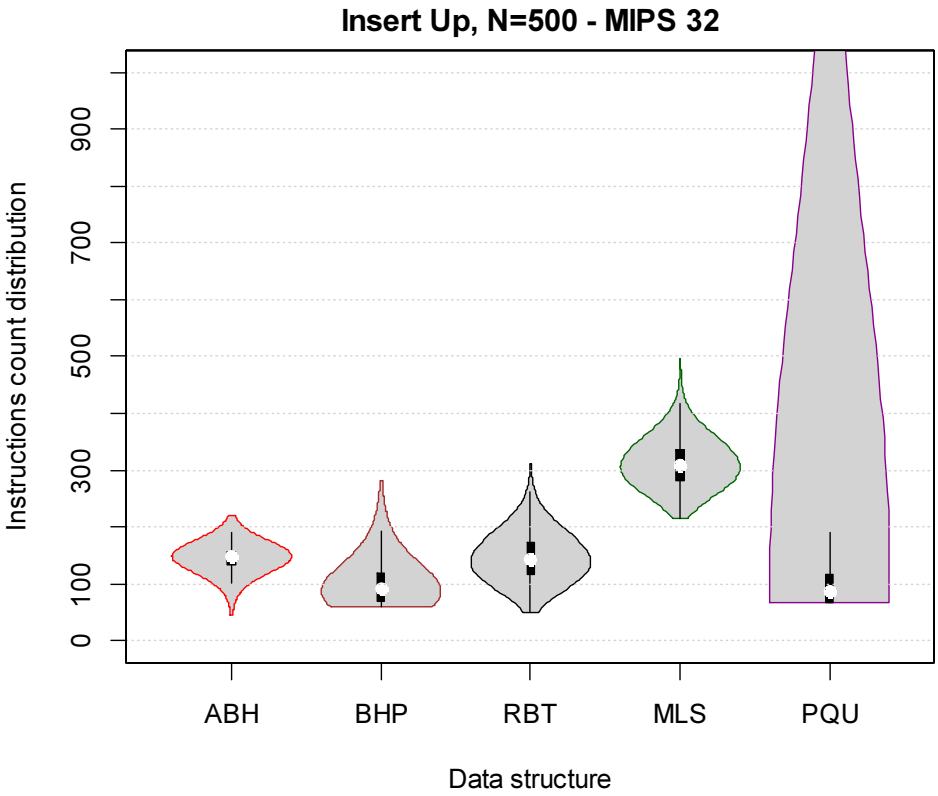


Figure 66: Execution time distribution for ramp-up insertion on MIPS 32.

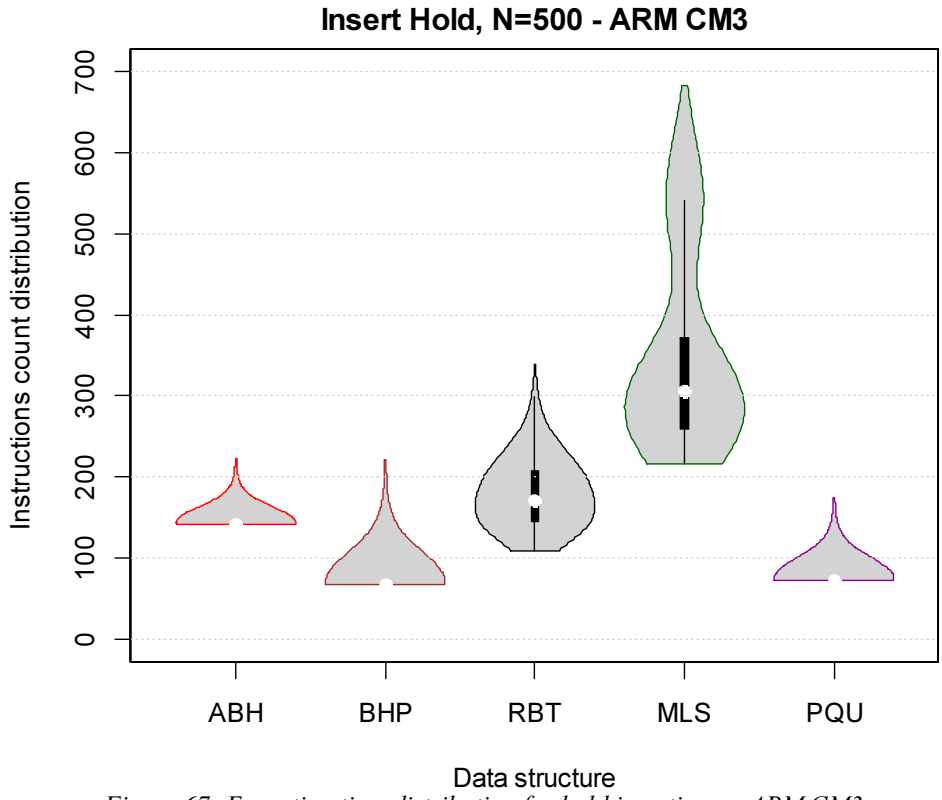


Figure 67: Execution time distribution for hold insertion on ARM CM3.

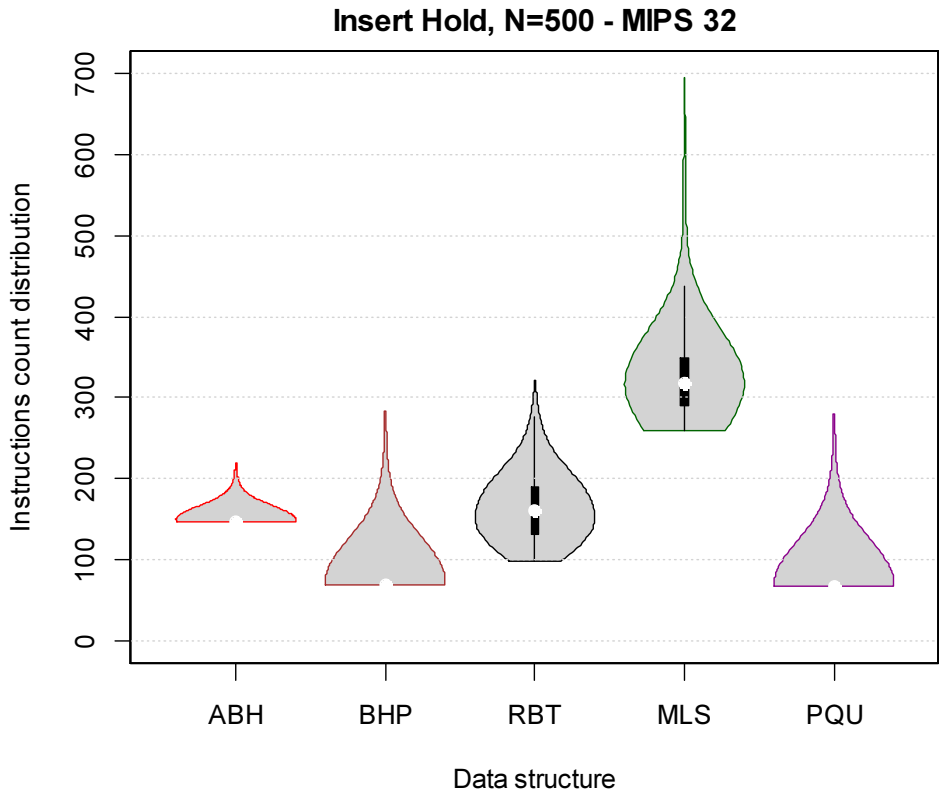


Figure 68: Execution time distribution for hold insertion on MIPS 32.

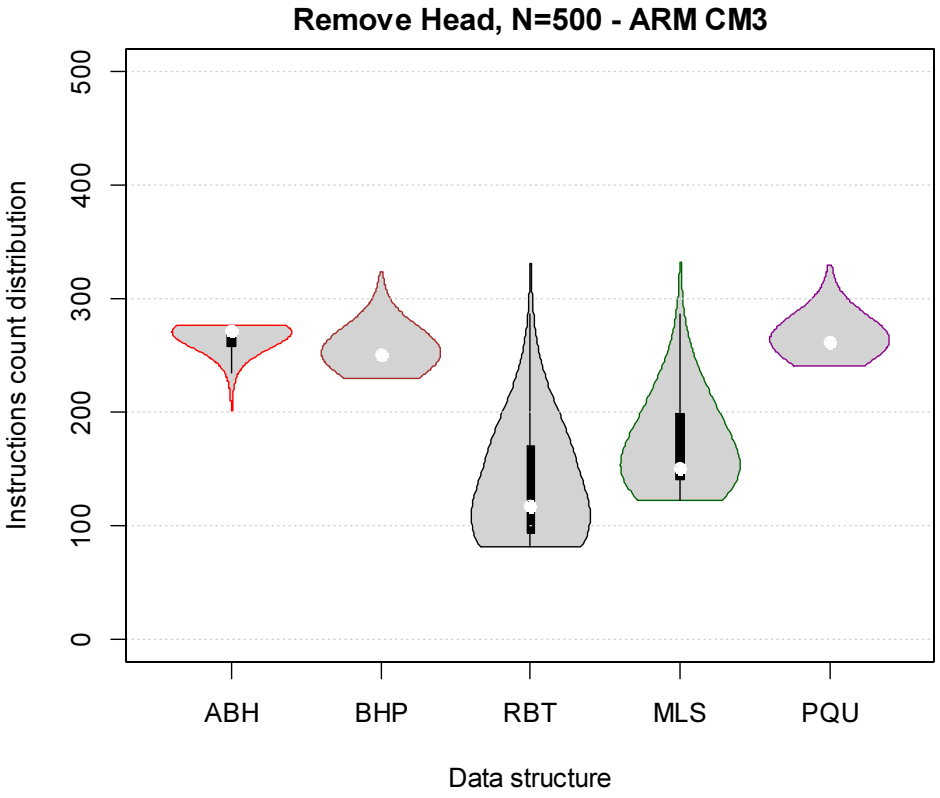


Figure 69: Execution time distribution for highest priority removal on ARM CM3.

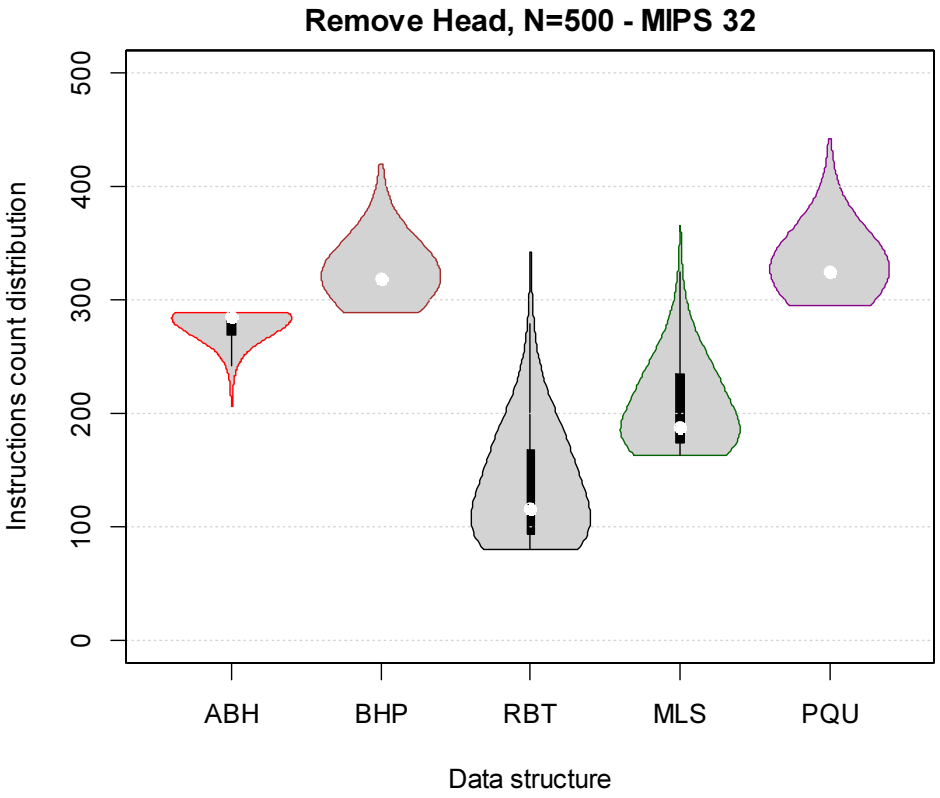


Figure 70: Execution time distribution for highest priority removal on MIPS 32.

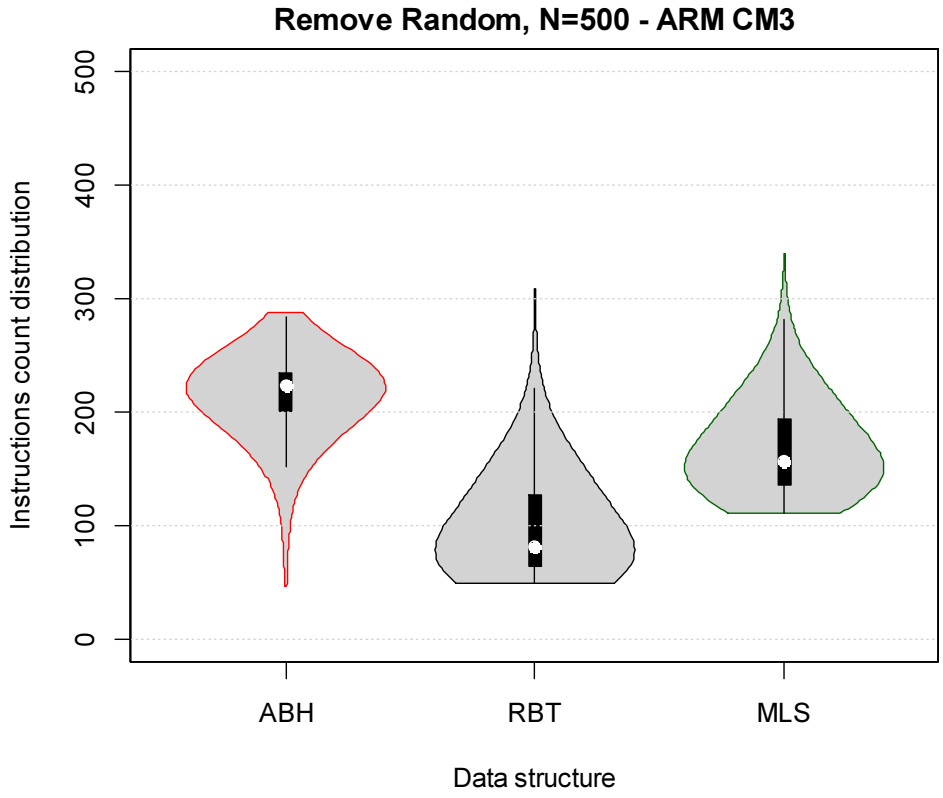


Figure 71: Execution time distribution for random removal on ARM CM3.

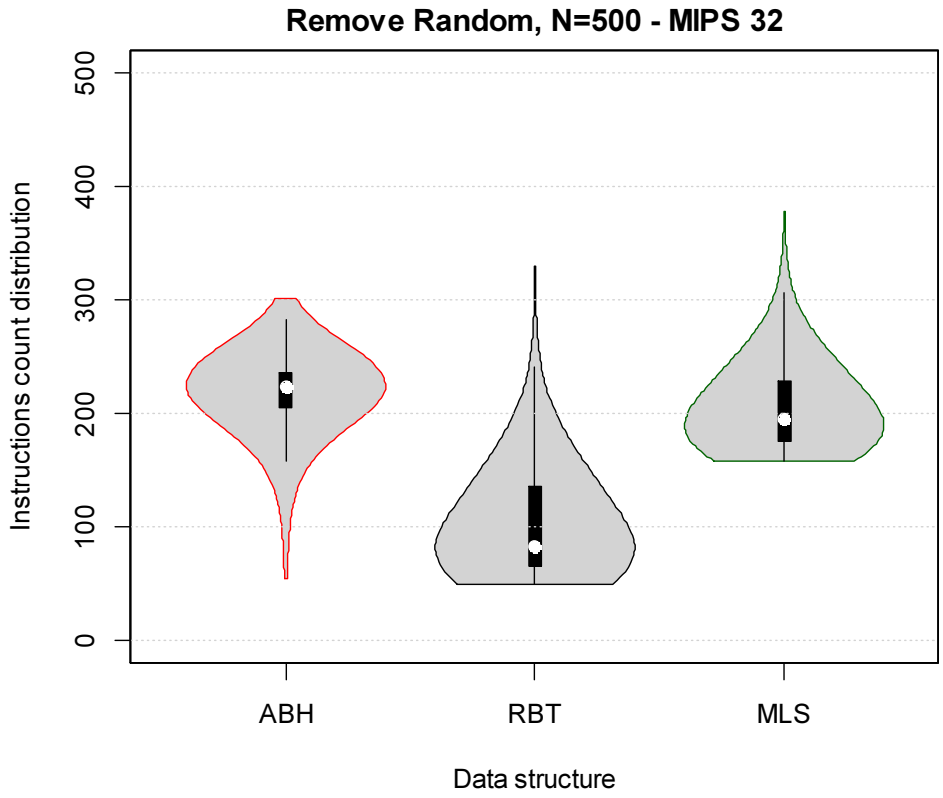


Figure 72: Execution time distribution for random removal on MIPS 32.

4.1. Concluding remarks.

This chapter presented a novel data structure, called addressable binary heap. At first, the theoretical properties which underpin its pointer-based physical structure are presented. Furthermore, a complete C implementation of the data structure is discussed in its full details, describing the operating principles of its main operations (*insert with priority*, *remove highest priority* and *remove random*) that enable the ABH to model a fully-fledged priority queue which operations all exhibit a O(log(*n*)) worst case run-time complexity and a O(1) memory run-time complexity (plus O(*n*) memory required for the data structure itself).

The viability and the performances of the ABH data structure have been later measured on an instruction-accurate virtual platform simulator, by means of synthetic experiments which emulate the manifold behavior of a timer queue in different scenarios and with different queue lengths.

Such experiments provided a detailed analysis of the mean, worst-case, and overall distribution of execution times, comparing the ABH with some mainstream data structures well known in literature to address timekeeping (and for priority queues in general) that are, timing wheels, red-black trees (both Linux kernel implementation and the C++ STL) and array-backed binary heaps (both STL `heap` and `Vector`-based `priority_queue`).

The timing wheel demonstrated to have excellent overall performances but only on tick-driven timekeeping. Its array-based structure, in fact, revealed definitely unsuitable behaviors for tick-less timekeeping, conversely to all the other data structure herein considered.

On the other side, the traditional array-backed binary heap (which is largely employed in most priority queue implementations), exhibits logarithmic worst-case complexity for all its operations, and good performances in general, being a good candidate for tackling the timekeeping problem with high determinism. However, it has two big drawbacks: (i) it complicates random elements removal, a frequent operation in timekeeping problems for canceling outstanding timers. (ii) Its physical structure is based on an array, which implies that the size of the problem (i.e. the maximum number of active timers), or at least a very good upper-bound, must be known a priori.

The C++ STL *priority_queue* implementation provides a solution to the latter issue, modeling a priority queue on top of a STL *Vector*, that is, in a nutshell, a dynamically expandable array. However, the experimental evaluations showed as its behavior makes such data structure absolutely not suitable for being employed in real-time systems. Its average performances, in general, suffer of substantial overhead due to the STL container logic. Furthermore the memory reallocations required to dynamically expand the queue length (which in some standards for embedded real-time systems is a completely banned practice) revealed to suffer disproportionately high overheads for the insertion operations.

For such reasons, red-black trees are generally preferred for deterministic handling of fine grained resolution timers, as, for instance, in the case of the Linux kernel's *hrtimers*, giving the possibility to arbitrarily insert and remove elements with O(log(*n*)) worst-case complexity. In this regard the Linux red-black tree implementation revealed to be more efficient compared to the HP implementation of the STL *multiset*, more likely due to the embedded anchor model.

In this scenario, the experimental evaluations of the ABH revealed very interesting results. During insertions, its average behavior outperformed red-black trees in almost every situation, and the measured worst-cases were never worse than any binary heap or red-black tree implementation (with the only exception of the hold insertion on ARM CM3, where PQU behaved slightly better). During highest priority removals, the average performances of the ABH were comparable (yet never worse) to the two red-black tree implementations, while the measured worst-cases of ABH were the best among all of them. Finally, the average behavior during random removals did not show any exceptional results when compared to red-black trees. However, the measured worst case behavior was constantly better than both of them.

In summary ABH showed a good average behavior, comparable (and sometimes better) to red-black trees, and a surprisingly good worst-case behavior, outperforming in many cases even the traditional array-backed binary heap implementation. Such bounded worst-case behavior makes

ABH a very good candidate for tackling timekeeping problems in highly deterministic real-time systems.

5. A hardware scheduling accelerator for MP-SoPCs

5.1. Introduction

In Chapters 2 and 3 it has been discussed as RTOSs and more in general software run-time infrastructures play a crucial role in an embedded real-time system. Besides the provisioning of fundamental building-blocks for the agile development of software applications, a RTOS is further responsible of ensuring the satisfaction of extra-functional requirements, in particular timeliness, ensuring that all the real-time tasks meet their deadlines.

In order to do so, however, a RTOS unavoidably needs to "steal" some computational power at runtime to take the proper scheduling decisions. As discussed in chapter 0(and in the works therein referred), the scheduling overhead is generally small as regards the percentage of CPU time, but still represents a non-negligible amount of absolute time which may jeopardize the schedulability of high-rate periodic tasks with periods in the sub-millisecond range.

Furthermore, in many scenarios such as digital control, data acquisition and telecommunication applications, another important extra-functional requirement is represented by bounding jitter. Due to the high complexity of modern (multi)processor architectures, keeping a low jitter can turn out to be more critical and difficult than handling the scheduling operations themselves (in particular in presence of caches and architectures with speculative and out-of-order execution).

Such contrasting requirements often force to make critical tradeoffs, i.e. using extremely elementary scheduling strategies in order to keep low scheduling jitter.

For such reasons, a vast number of publications in the field of embedded real-time systems have introduced alternative approaches that aim to exploit dedicated hardware resources to address these scheduling issues.

In general, the use of hardware co-processors for offloading frequent and critical software computations is a definitely not new strategy that has been out for decades. However, while in many other fields (e.g., floating-point calculus, encryption, audio/video (de)coding, TCP/IP networking) the wide

homogeneity and the large scale of the problem easily justified the high costs of dedicated hardware, the situation has been always different for the field of embedded systems. Embedded systems, in fact, usually exhibit very custom and singular requirements, even as regards the mere problem of real-time scheduling, mainly due to the wide heterogeneity of the underlying computational platforms involved.

This situation, however, drastically changed in the last decade, as the evolution reconfigurable hardware platforms (FPGAs) led to high powerful and inexpensive platforms (compared to low-volumes of dedicated ASICs), which today can integrate in a single physical chip all the resources required for a complete embedded system (CPU, memory, peripherals). This system on programmable chip (SoPC) paradigm introduced by mainstream vendors, has made FPGA interesting targets for the development of many embedded real-time systems, where the hardware/software co-design reaps the benefits of both rapid development and large possibilities of customization, reducing nearly to zero the cost of using ad-hoc co-processors and accelerators to address custom needs.

5.2. Related work

In 1991 Lindh at al. presented [Lin1991] a proof-of-concept of a hybrid hardware-software RTOS implementation, moving critical parts of the RTOS kernel in hardware, in order to reduce indeterminism of the conventional pipelined processor architectures. FASTCHART consists of a simple hardware real-time kernel, supporting 64 tasks with 8 priority levels, which execution is handled by mean of a simple multi-level FIFOs dispatching. The proof-of-concept, later turned into a more advanced and complete project called FASTHARD [Lin1992].

Based on this work, later in 1996 another project called RTU (Real-Time Unit) is carried out by Lindh et al., a completely hardware based kernel supporting more run-time services such as tasks delays, semaphores, event queues, and interrupt handling simultaneously on three homogeneous processors. The RTU is interfaced through a memory mapped bus, which

is accessed through a round-robin arbiter, and uses a single interrupt input of each processor to control context switching [LMID+2003].

In 2005 in [NLJS2005] Lindh et al. introduce support interfaces for integrating the RTU hardware kernel in the μC/OS-II RTOS. The RTU is also at the time commercialized as a commercial product (Sierra Kernel). RTU only supports binary semaphores for process synchronization, and its priority scheme is fixed (changes of priority levels are not possible after task creation). The timing measurements show that the functions implemented in hardware are accelerated up to a factor of 370%, and in [NA2007] hardware configurability is added to the single processor version of the RTU. A newer version of the RTU is available in the form of a customizable IP core for implementation in FPGA. It can be configured to have 2-512 tasks, 2–1024 priority levels and binary semaphores.

A similar approach is introduced by the STRON project (silicon real-time operating system nucleus) in [NUI+1995]. The system is based upon the μITRON real-time OS, re-implementing most of its system calls and core functions in hardware. A small micro kernel has been implemented to take care of the features not implemented in hardware, and to serve as the API to the hardware kernel. The STRON handles contains task management, event flags, semaphores and timers, as well as external interrupt management. The interface with the CPU is handled by means of a basic memory mapped scheme using, using a single interrupt line to the CPU. Implementation tests show that the circuit can be realized in VLSI CMOS technology and that the RTOS function calls are accelerated between 6 and 50 times compared to the software version, while task release and activation jitter is almost completely removed. The STRON hardware RTOS is less capable than the RTU as it doesn't directly support periodic release of tasks and has a timer horizon limited to only 255 ticks.

In [PSJC+1997] Parisoto et al. introduce a FPGA-targeted accelerator called F-Timer, aimed at implementing task scheduling and interrupt handling in hardware. It can handle up to 32 tasks and 64 different priority levels. The interface with general purpose processors is handled by means of a standard memory mapped bus and a single interrupt line to the CPU. The overall system is designed much like a software RTOS with memory

based queues for the ready, timer and interrupt handling queues. The paper, however, gives no information about the scheduling algorithm employed, and there is also no hardware support for task synchronization.

In [SR1991] Stankovic et al. present the Spring kernel which, compared to traditional RTOSs, takes a radically different approach for handling real-time task scheduling. The conventional approach of priority-driven scheduling is abandoned in favor of a dynamic and speculative scheduling, implemented by the means of heuristic algorithms. Taking into account all active tasks' WCET, deadline and resource constraints, the scheduling algorithm constructs a custom schedule which guaranteed that all tasks will meet their deadlines and never block waiting for resources. When a new task arrives it is only added to the current task set if a new feasible schedule can be constructed. The Spring kernel is designed for large and complex real-time system running on multiprocessor systems. The system is partitioned into application processors and a system processor. Each application processor runs a lightweight dispatcher which executes the task allocated for that processor in the order determined in the schedule calculated by the system processor. In addition, the system processor takes care of the remaining RTOS activities (e.g., servicing interrupt requests) in order to insure that the AP processors aren't affected by external events.

The approach, however, has the evident main drawback of requiring long and complex computations in order to produce a feasible schedule, thus being less suitable for dynamic systems. In order to address this issue, in [BKN+1999] Stankovic et al. introduce the spring scheduling coprocessor, which implements in hardware the planning algorithm used originally used in the spring kernel. The implementation is done in 2µm VLSI CMOS technology and is designed with low on-chip memory requirements, making the tree search less effective if backtracking is needed, but still much faster than if done in software thanks to the massive hardware parallelism.

In [KGJ2003] Kohout et al. identify the software scheduler as the major cause of performance degradation of a RTOS and aim at replacing it with a hardware implementation called real-time task manager (RTM). RTM supports static priority scheduling, and offers hardware support for time

and event management. RTM makes massive use of hardware registers to keep the state of the tasks and allow parallel implementation of the scheduling algorithms. The computational delay of either scheduling or event management is O(log(*n*)) and the hardware cost scales linearly with the number of resources (tasks, events, timers) handled. The RTM requires roughly 2600 register-bits for 64 task records which is stated to require around the same die size as a 32bitx64 word register file, making it feasible for on chip implementation next to a CPU. The experimental results using various benchmarks comparing the performance with µC/OS-II and a homemade non-preemptive OS (NOS) showed a decrease in RTOS overhead of 90% and an 81% decrease in response latency.

In [CRL2006] a different approach is used for hardware-accelerating an already existing RTOS (eCos). The project discussed in the paper aims at automatically generating a hardware implementation of the popular RTOS by means of a behavioral software synthesis. The goal of the project are to improve system performances and reduce the RTOS footprint, by means of moving task management, scheduling and synchronization into hardware. The authors argued that the main reason of scarce commercial diffusion of hardware accelerated RTOS is to be found in the restricted chip-to-chip communication infrastructure and their relatively slow communication speed of the older architecture employed in prior works, which hinder the speed-up provided by the hardware. The project is interfaced to an ARM processor using a memory mapped approach similar to the other works herein referred. The hardware implementation is stated to require less than 10K gates and provide on average a 15x speedup for an image filtering algorithm. The number of tasks and resources supported by the implementation is not known.

In [MIY2010] Maruyama et al. propose a hybrid hardware-software implementation of a networked RTOS for TCP/IP processing (called ARTESSO), aiming at improving not only the dependability of the system, but also the network performances, by means of offloading both critical RTOS routines and part of the TCP/IP protocol processing in hardware. The hardware-RTOS is endowed with a priority-driven scheduler, which is implemented through a novel queue structure called VQueue. Such

VQueue aims at reducing the hardware requirements in terms of logic-gates, compared to traditional FIFO-based queue implementations. The ARTESSO hardware RTOS supports 256 tasks and 16 priority levels, and uses a parallel tree structure which guarantees a O(log(*n*)) complexity for taking scheduling decisions. The experimental evaluations are carried out on an ARM926 processor clocked at 50Mhz and show a 6x-9x performance boost over STRON, and an increased 11x network processing bandwidth.

5.3. Motivations

The wide number of projects and papers published in this field suggest that the principle of offloading the RTOS operations (or at least part of them) to the hardware is generally a viable approach, especially in reconfigurable hardware platforms like FPGA-based SoPCs.

It might be worth noting, however, that while some of the aforementioned approaches deal with multiprocessor systems, most of them support just static priority-driven policies. Other few works in this field [HGT1999, MIB2002] introduce support also for the EDF policy, but only for uniprocessor systems.

Therefore, considered the current state of the art, the purpose of this work is to make a step forward in this direction, presenting the design and implementation of a FPGA-based hardware scheduling accelerator which supports the multiprocessor G-EDF and R-EDF (restricted migration variant) policies.

Similarly to the rest of the software produced in this thesis, also the full VHDL sources implementing the scheduling accelerator are available for free in the code repository of the X-RT project [TUC2012].

5.4. Hardware design

Extending the preliminary architecture introduced in 3.5, which was already exploiting an FPGA (yet only as a general-purpose AMP platform) some of the hardware cells of the FGPA, in this project, have been further employed to implement a R-EDF scheduling accelerator.

The scheduling accelerator has been developed as a VHDL (VHSIC hardware description language) IP core according to the Altera SoPC methodology.

The Altera SOPC is a design methodology promoted by Altera, which aims at rapid development of reusable IP cores, interacting each other by means of a common switch fabric, called Avalon. The Avalon switch fabric is a flexible and modular bus designed for on-chip interconnection of processors and peripherals. Its specifications define several interfaces that the IP cores can implement in order to interface to the bus (e.g. memory-mapped master/slaves, interrupt sender/receivers, streaming source/sinks).

The switch fabric for memory-mapped peripherals supports simultaneous multi-mastering, allowing multiple bus masters to perform bus transactions concurrently, as long as they do not access the same slave during the same bus cycle. In the event that multiple masters attempt to access the same slave at the same time, a built-in arbitrator prioritizes accesses to that slave (Figure 73).

The real advantage brought in by the Altera SoPC methodology is represented by the SoPC builder, that is a software tool that enables to rapidly and easily integrate proprietary and third party IP cores, taking care of automatically and transparently generating all the hardware related to the bus, address mapping, decoding logic and arbitration. Essentially it is a system-generation tool that let the designer define, parameterize, link, and integrate a wide variety of IP cores (such as soft processor, DSP, communication and memory controller cores) in the company's high-density FPGAs.

This reduces the amount of time designers must spend on peripheral integration and increases their ability to reuse peripherals in subsequent designs. The interconnect fabric uses minimal FPGA logic resources to support address decoding, wait-state generation, pipelined and burst transactions, peripheral address alignment, interrupt-priority assignment, data path multiplexing and clock domain crossing

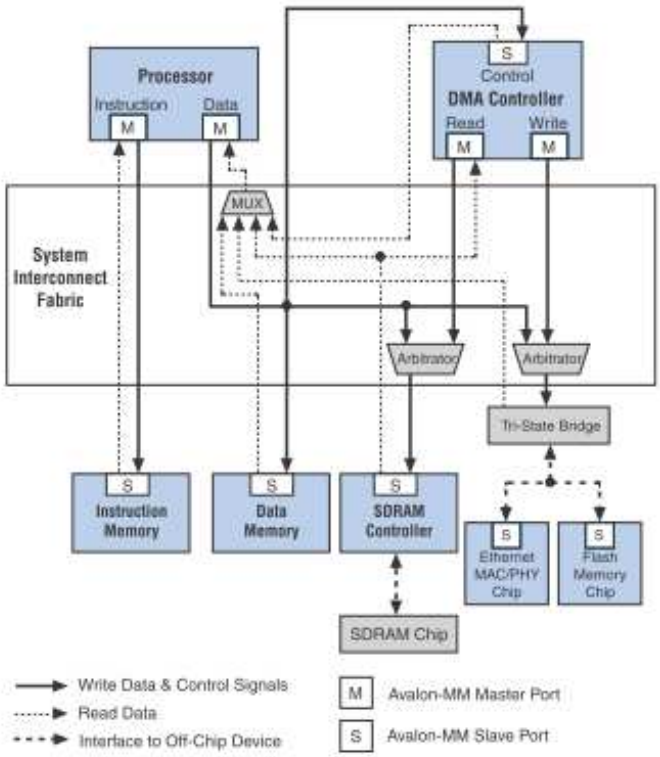


Figure 73: Overview of the Altera SoPC architecture.

Design principles

Before delving straight in a discussion about the design of the scheduling accelerator, this paragraph introduces the principles that driven the design of the overall hardware scheduling architecture. Inspired by a divide and conquer methodology, the design process aims at organizing the hardware components in a multi-layer architecture, composed by a *functional units layer*, an *operational services layer* and an *orchestration layer*. This multi-layer architecture attempts to define the basic contract interfaces for the hardware components, independently of the particular function that they implement, in terms of basic I/O signals expected, their semantic and the handshaking protocols for their interoperability.

Functional units layer

A functional unit is a low level entity which encapsulates the data structures needed to accomplish a specific function and exhibits

elementary micro-operations which can be performed on that. As a general interface contract, every functional unit must provide:

- A input signal (one-bit) for requesting each operation exposed.
- A global READY output signal (one-bit) which states, when asserted, that the functional unit is ready to accept and carry out a new operation.

The interaction protocol with the upper layers is defined as follows:

- Each request signal must be asserted for at least one clock cycle and cleared before the operation is completed.
- The READY signal must go down as the request is acknowledged and held down until the rising edge of the clock that follows the completion of the operation.
- Changes of inner state (content of the data structures) of a functional unit can occur only at the rising edge of the system clock (i.e. purely synchronous design).
- The READY signal can be tied to '1' for functional units which involve only mono-cycle operations.

Operational services layer

The elementary operations exposed by the functional units are, in general, not sufficient to respond to the needs of the problem domain. The services offered by the infrastructure are more sophisticated and involve one or more complex sequences of micro-operations in order to accomplish a whole high-level service.

In this sense, an operational service is a high level component that encapsulates the decisional logic for each service offered by the infrastructure, handling the sequence of micro-operations by means of finite state machine evolutions, taking advantage of one or more functional units.

As a general interface contract, every operational service must provide:

- REQUEST: an input signal (one-bit) to request the execution of the service.
- DONE: an output signal (one-bit) to notify the completion of the service.

The interaction protocol with the upper layers is defined as follows:

- The REQUEST signal must be asserted for at least one clock cycle and cleared before the service is completed.
- The DONE signal must be asserted only when the service is fully completed and all the involved functional units have refreshed the new output configurations.

Distinct services may require, by design, to interact with the same functional units, leading to unavoidable conflicts. Such conflicts are resolved introducing ad-hoc entities, called *resource arbiters*, which handle by means of static-priority resolution, the multiplexed access to the shared functional units.

Orchestration layer

The orchestration layer represents the higher-level layer that bridges the services offered by the operational layer together with the hardware environment in which the infrastructure is employed (the Avalon bus in our case).

It exposes an unique interface to the rest of the environment, in the form of a memory-mapped slave, decodes the incoming requests, routes them to the proper operation services, and provides the results, properly respecting the specificities of the interface protocol (e.g. handshaking with wait-requests on the bus, performing serialization and encoding of the in/out data, etc.)

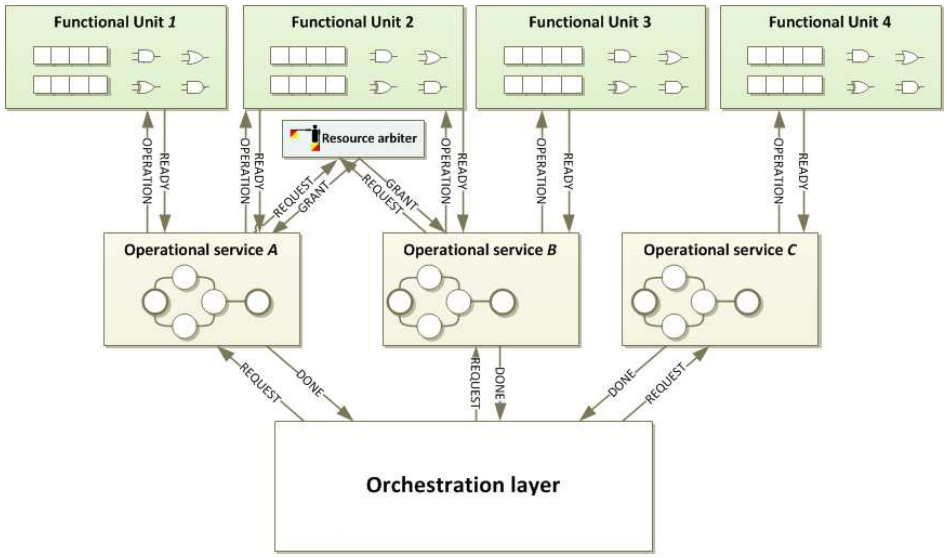


Figure 74: Interactions between components in the three-layer architecture of the scheduling accelerator.

Now that the core design principles and interface contract have been presented, we can delve into the in-depth discussion of the scheduling accelerator components. In the following, the components of the scheduling accelerators are described in a top-down fashion, starting from higher level ones down to the individual functional units. Figure 75 gives a preliminary overview of the overall architecture.

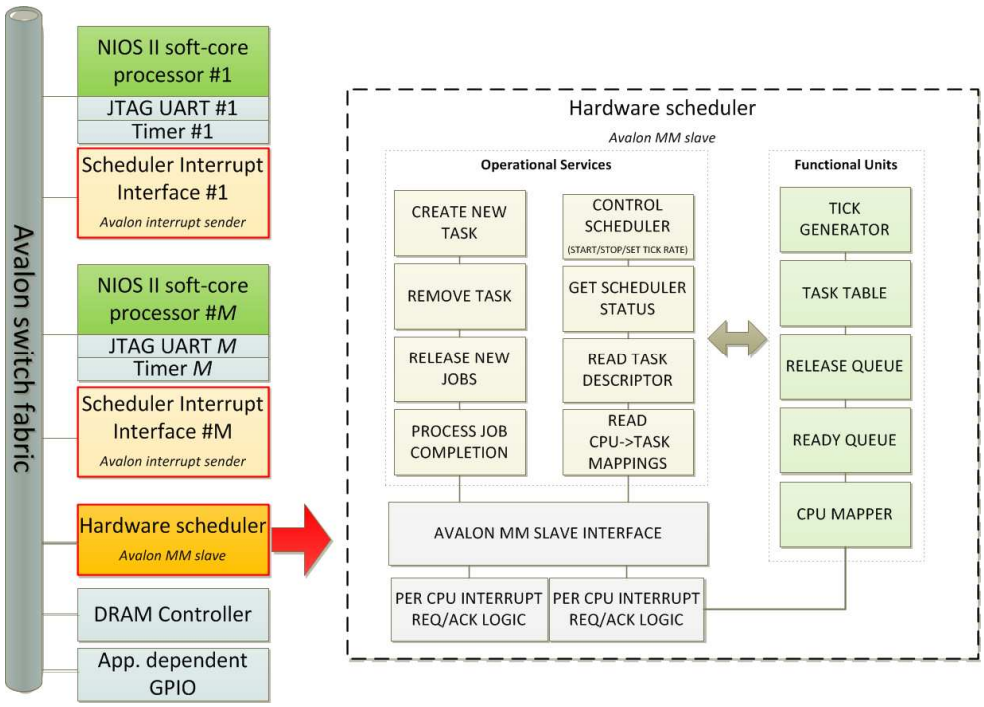


Figure 75: Hardware architecture of the scheduling accelerator based on Altera Avalon.

Functional units

Task table

The task table (Figure 76) is a register file which holds the task descriptors, storing for each of them the state, period, deadline and statistic counters (jobs executed, missed deadlines, overruns). It is periodically updated by the other components and can be enquired to retrieve details of tasks.

Figure 76: I/O signals of the the task table functional unit.

Tick generator

The tick generator (Figure 77) produces the time reference signal for the whole infrastructure. It cyclically triggers a periodic signal called TICK, whose interval is programmable. Such signal is delivered to the *release queue* and *ready queue* units for the timekeeping activities, and defines the resolution of their timing registers (release and deadline counters).

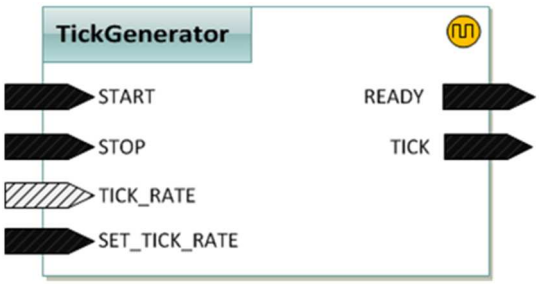


Figure 77: I/O signals of the the tick generator functional unit.

Release queue

The release queue (Figure 78) is a set of per-task countdown timers which are decremented at every TICK. When the *n*-th counter reaches zero, the output bit RELEASE_TASK[*n*] is asserted, notifying that the release period of the *n*-th task is expired and a new job must be released. Figure 84 shows the inner gate-level RTL architecture of the functional unit.

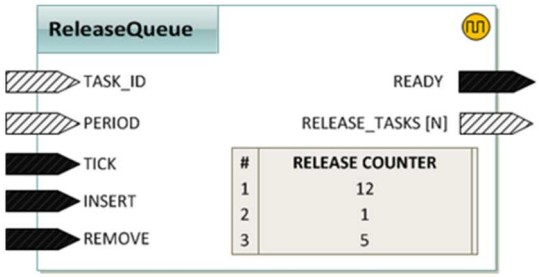


Figure 78: I/O signals of the the release queue functional unit.

Ready queue

The ready queue (Figure 79) is a priority queue of ready tasks, ordered by increasing values of absolute deadlines. For each task, a countdown timer is inserted in the queue upon job release and is removed when it completes. If a deadline counter reaches zero, the corresponding bit in the MISSED_DEADLINE[*n*] output signal is asserted, notifying that a task missed deadline notification is raised. Figure 85 shows the inner gate-level RTL architecture of the functional unit.

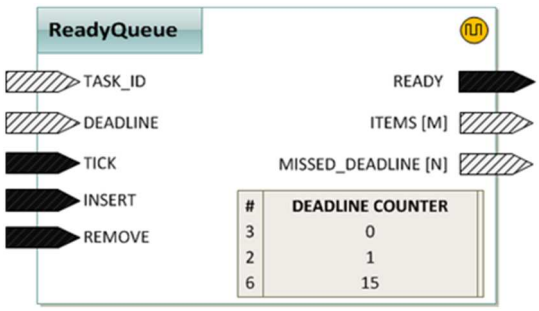


Figure 79: I/O signals of the the ready queue functional unit.

CPU mapper

The CPU mapper (Figure 80) is the core component of the scheduling infrastructure. It takes the scheduling decisions according to the scheduling policy, mapping ready tasks on the available CPUs as long as a change of the ready queue is notified. In the current implementation the CPU mapper implements the G-EDF policy.

Figure 86 shows the inner gate-level RTL architecture of the functional unit: a set of $2m$ registers hold the *current* and *next* running tasks assignments (i.e. which task is running on each CPU). Whenever a change is detected on the ITEMS output of the ready queue, a priority encoder is used to determine the m-th highest priority tasks among them and assigning the *next* mapping registers. The assignment process is performed in three sequential stages: in the first stage the new set of (at most) m tasks that must be running is determined using the priority encode; in the second stage, the tasks that were already running (before the ready queue change that triggered the remapping process) are confirmed on their previous CPUs, in order to minimize the impact of migrations; in the third stage the remaining tasks are assigned arbitrarily to the CPUs left.

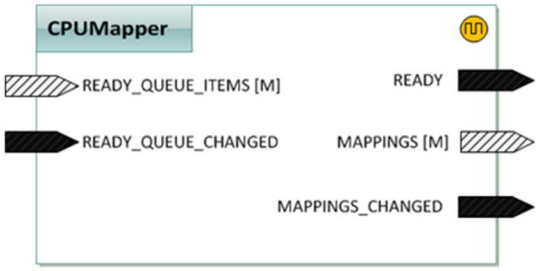


Figure 80: I/O signals of the the CPU mapper functional unit.

Operational services

Create new task

The *CreateNewTask* service (Figure 81) is invoked by the software upon system initialization, in order to introduce new tasks into the infrastructure. For each task, the caller must provide its relative deadline and the period. When the service is requested, the task table and the release queue are populated with the relative entries and the numeric identifier of the new task is returned. The service in practice takes care of inserting atomically the new task both in the task table and in the release queue, after acquiring exclusive access to both.

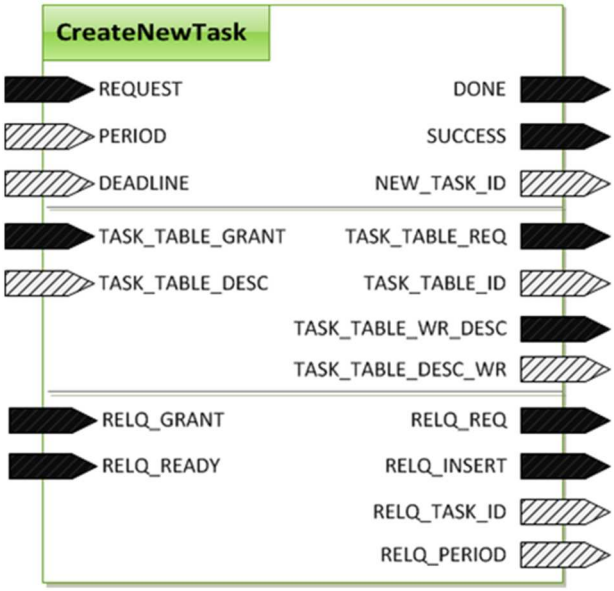


Figure 81: I/O signals of the the CreateNewTask operational service.

Notify job completion

Every time a task's job completes, the software notifies the event to the hardware infrastructure. The *JobCompletion* service (Figure 82) takes care of updating the task statistic counters, its state in the task table and removing it from the ready queue, in order to allow the next higher priority task, if any, to be promoted.

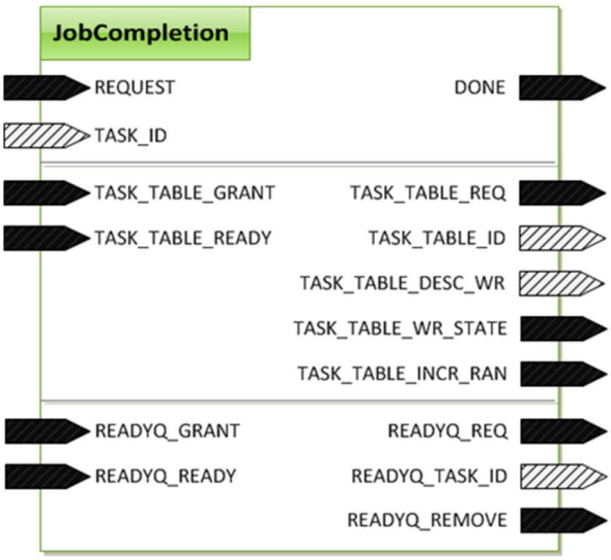


Figure 82: I/O signals of the the NotifyJobCompletion operational service.

Job release

As soon as the release queue signals the need to release one or more jobs, the *ReleaseJobs* service (Figure 83) is invoked. It is in charge of updating the released tasks' state into the task table and inserting them into the ready queue, in order to allow them to be consequently scheduled by the CPU mapper according to the priority resulting from their deadline.

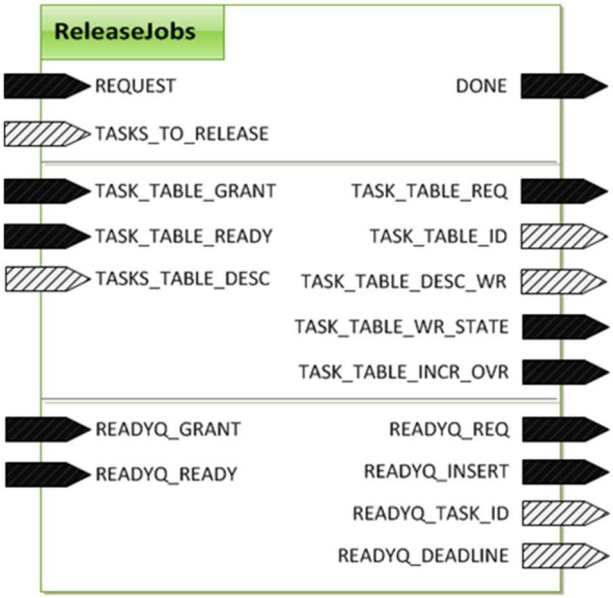


Figure 83: I/O signals of the the JobCompletion operational service.

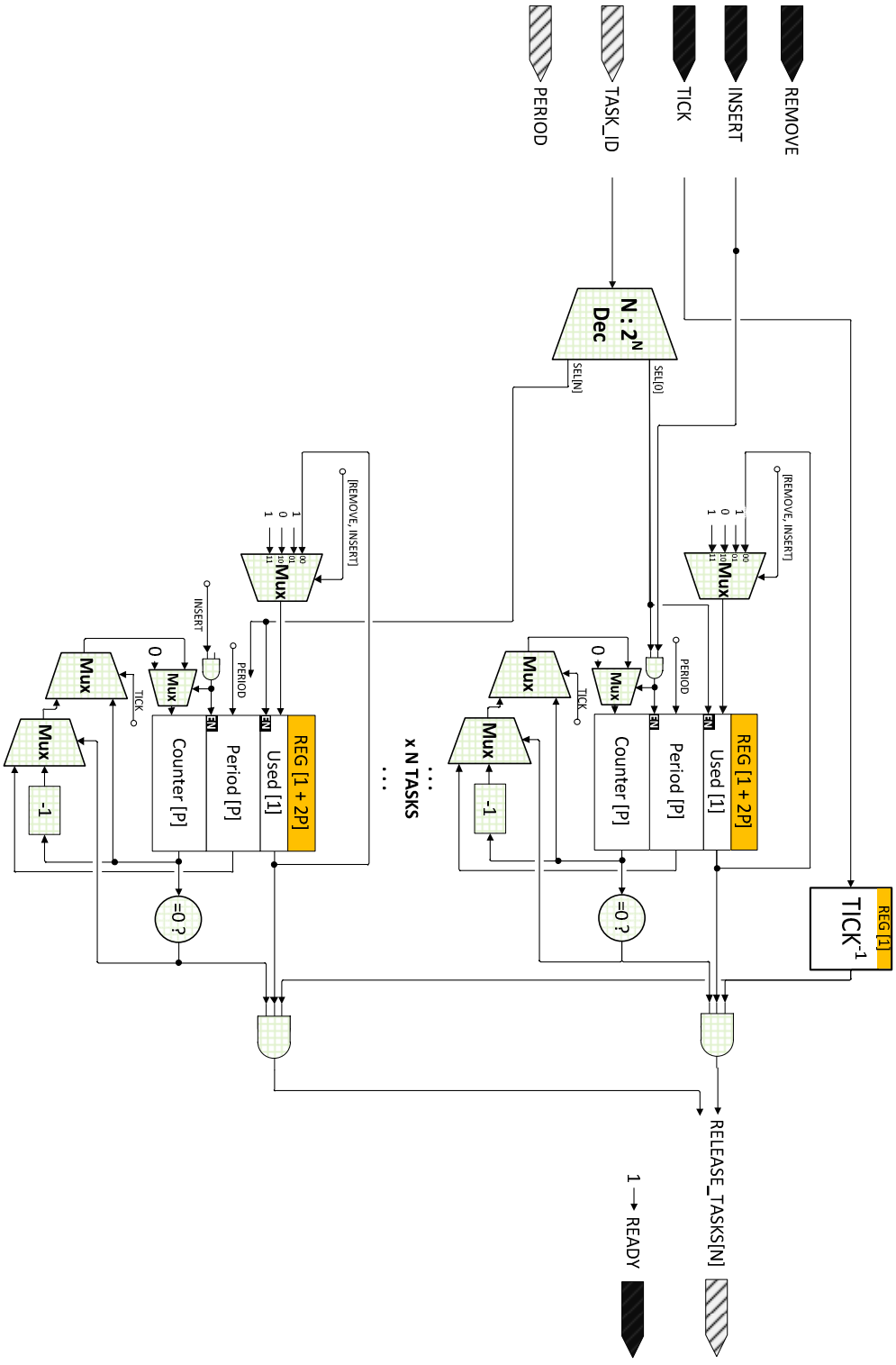


Figure 84: Hardware architecture (RTL) of the release queue.

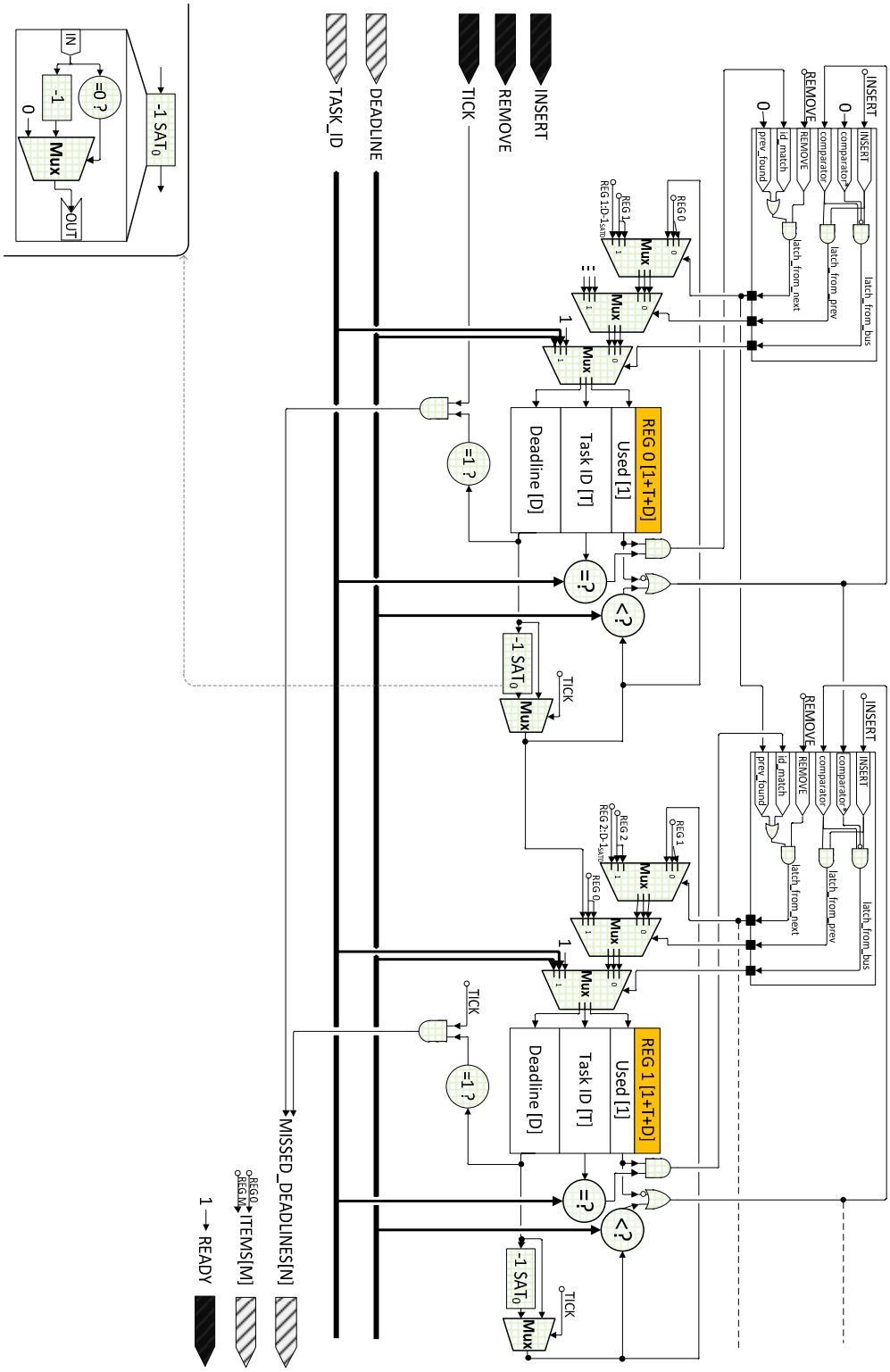


Figure 85: Hardware architecture (RTL) of the ready queue.

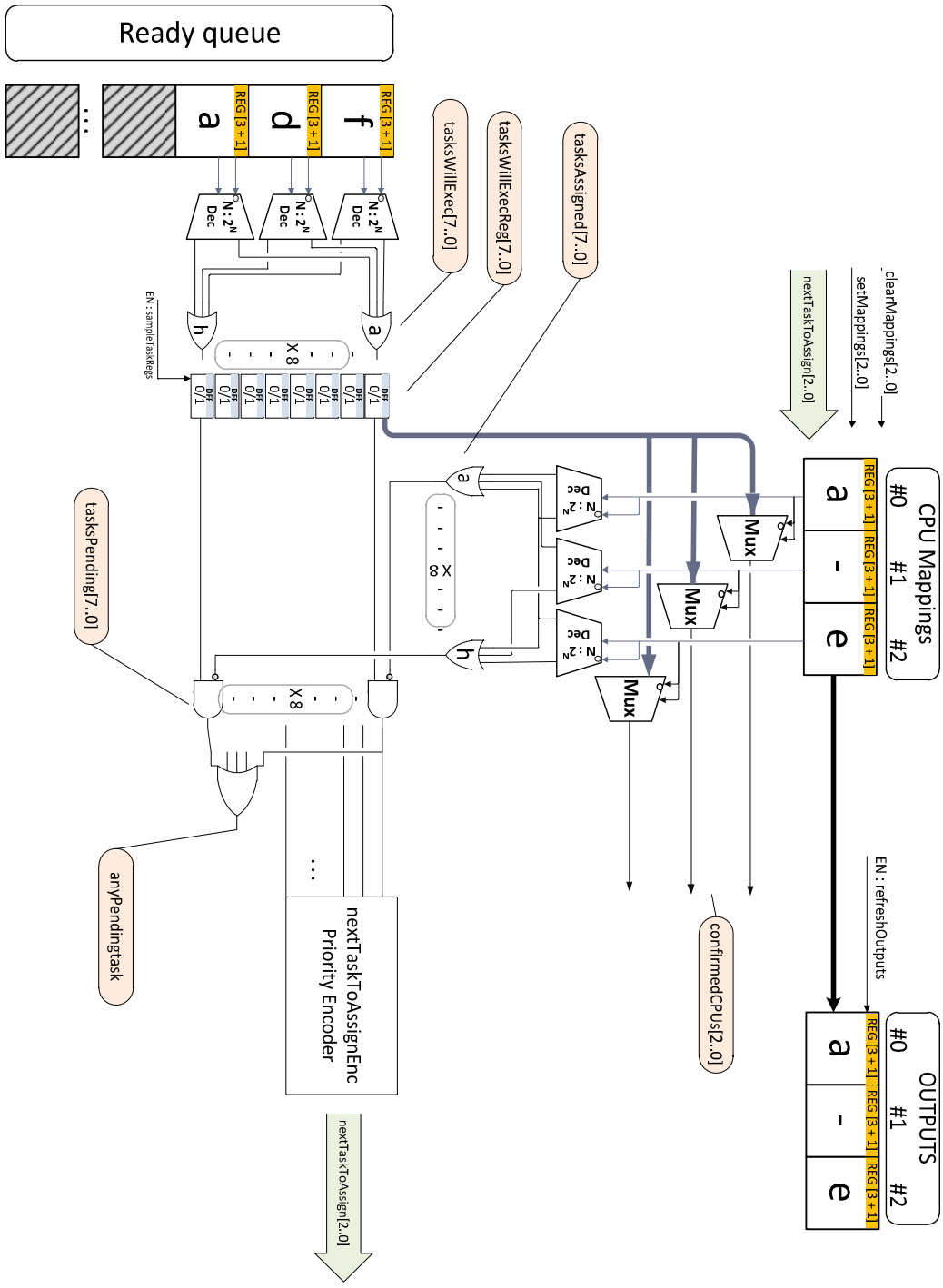


Figure 86: Hardware architecture (RTL) of the CPU mapper.

5.5. MP-SoPC architecture

Using Altera NIOS II soft-cores, the Avalon bus, and Altera IP cores (memory controller, GPIO, etc.), the reference SoPC platform used to validate and test the hardware scheduling accelerator has been organized as follows:

- *m* NIOS II/*f* processors, *m* can be chosen arbitrarily by the end-user on the basis of the number of the tasks, the required computational power and FPGA resources availability.
- A SDRAM memory controller employed to hold on an external memory the RTOS and application footprints and the corresponding run-time memory (stack, heap and data).
- The hardware accelerator, a memory-mapped Avalon slave IP core that takes care of the task scheduling and the dispatching process, interacting with the NIOS processors by means of interrupt signaling.
- Application dependent peripherals, e.g. GPIO for motor control, UART for user interface, etc.

Figure 87 shows the HW/SW architecture of the SoPC platform used as a testbed for the hardware scheduling accelerator. The hardware scheduler memory-mapped slave is connected to the data bus of all the NIOS processors, in a memory range that is accessed exclusively by the scheduling framework. Furthermore *m* dedicated interrupt interfaces connect to the IRQ0 line (the one with highest priority) of the NIOS processors.

Due to the use of NIOS processors, the resulting SoPC architecture is an AMP system, which takes advantage, on the software side, of the same principles illustrated in 3.5.

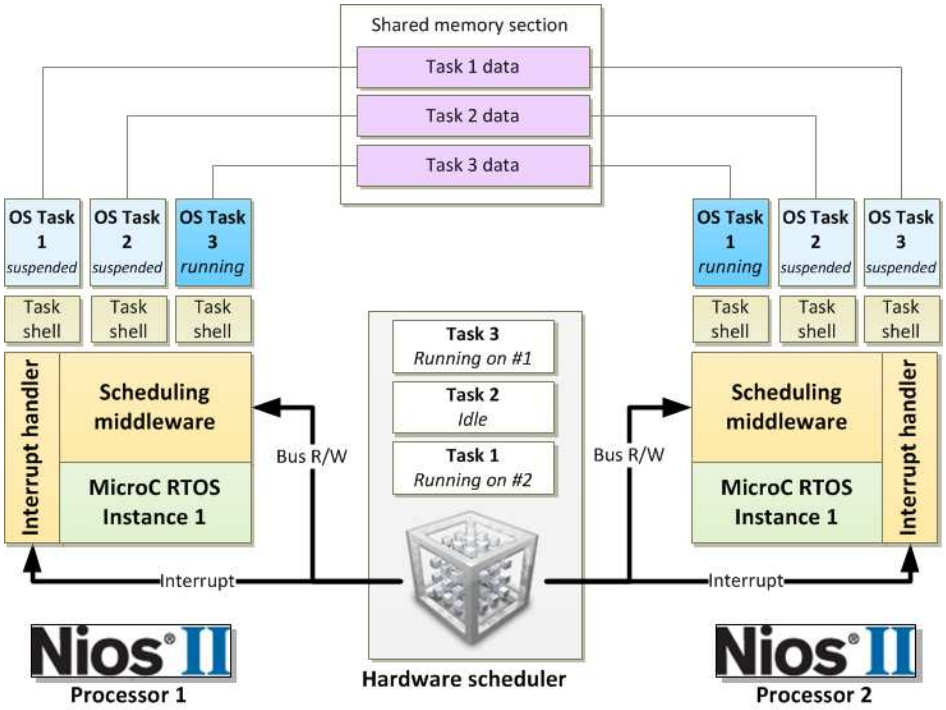


Figure 87: Architecture of the SoPC platform integrating the scheduling hardware accelerator.

In particular, as regards the memory layout, a single external SDRAM memory chip has been exploited for all the *m* processors. The reason of this choice is that the purpose of this testbed is not to evaluate the performances of the soft-cores themselves (this analysis has been already carried out in 3.6). Compared to on-chip memories, the use of an external SDRAM memory allows to save many hardware cells on the FPGA, which can be employed, instead, to assess the timing performances and the expandability of the hardware scheduler itself.

Furthermore, since each processor has dedicated instruction and data caches, the performance evaluations are not affected by the latencies of the SDRAM, especially considering the very small footprint of the software layer involved.

The memory available on the SDRAM is logically organized in $m + 1$ partitions (Figure 88) through the use of custom linker scripts.

Each partition consists of a code section, containing the code for the RTOS which is pointed by the reset vector of each processor, and a data section, used for storage of RTOS variables, stack and heap. Those *m* partitions are

completely independent of each other, and can be replaced at any time with dedicated on-chip memory connected exclusively to each processor.

Finally a shared memory partition, accessible by all the processors, has been envisaged for the task code and data, and for the shared scheduling framework library.

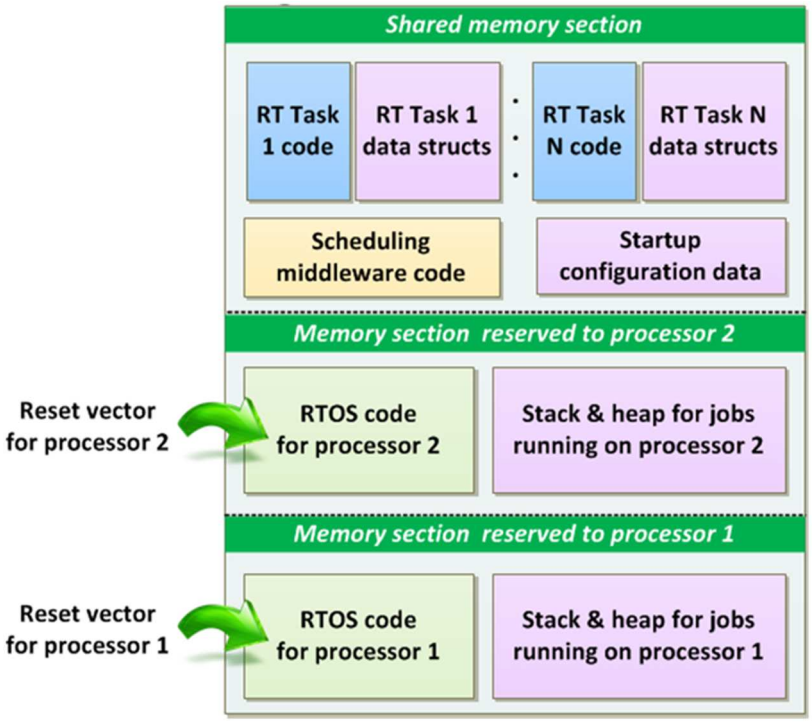


Figure 88: Memory organization of the embedded software running on the soft-cores.

5.6. Hardware synthesis results

The preliminary introduction of a clear and well defined multi-layer architecture has brought several advantages as regards the extensibility of the overall design. First of all, the introduction of new functionalities (e.g. a new scheduling policy) requires just the integration of the new component without impacting with the stability and the complexity of the existing design, allowing to completely reuse the other existing functional units and operational services layers.

In particular, the proposed approach achieves a clean decoupling between the functionalities offered by each layer and the timings with which they are carried out. Just to take a mere but concrete example, during the

implementation of the *CPU mapper* the design needed to be reiterated due to low f_{MAX} caused by its heavy combinatorial logic, introducing two pipeline stages (its inputs and outputs are registered). Obviously, the latency, in terms of clock cycles, required by the unit to complete its operations has increased. However, such modification did not require any intervention on the upper layer (*operational services*), since all the interactions stand on the *request/ready* protocol. Pipelining just caused, in this case, a delay in the generation of the *ready* signal and, therefore, a delay in the completion of the upper layer services involved with the *CPU mapper* unit.

The use of VHDL generics allowed to model the components independently of the dimensions of the scheduler, more specifically, independently of the (maximum) number of tasks in the system, the number of processors and the desiderated resolution for task counters. All these parameters that state the dimension, and therefore the complexity, of the infrastructure are uniquely described in a package declaration file, for which a brief outlook is here reported:

```
1. package HWGlobalScheduler is
2.   constant  N_CPUS          : natural := 2;
3.   constant  TASKID_SIZE     : natural := 3;
4.   constant  PERIOD_SIZE     : natural := 16;
5.   constant  DEADLINE_SIZE   : natural := 16;
6.   constant  STATS_SIZE      : natural := 8;
```

.

Every component takes advantage of these definitions to dynamically adapts its structure (in terms of inferred logic gates). In this way, it is possible to completely adjust the scheduler complexity according to the needs of the end-user and the complexity of the application, just editing nothing but 5 rows of VHDL.

Figure 89 reports the results (related to the only hardware infrastructure) of the synthesis on a Cyclone II EP2C20F484C7 using the Quartus II EDA (area vs. speed optimizations: balanced) and the TimeQuest timing analyzer (slow-model).

In order to provide a term of comparison for the f_{MAX} (which is strongly dependent on the FPGA process technology and doesn't provide much

valuable information itself), an equivalent system, in terms of hardware cells usage, made of 6 NIOS II/*f* processors, plus the other standard Avalon peripherals, reached a f_{MAX} of 64 MHz on the same FPGA.

Hardware scheduler configuration	Total comb. functions	Total logic registers	f_{MAX}
2 cores, 8 Tasks, 8 bit deadline and period counters, 8 bit stats counters	1246 (7%)	598 (3%)	59.8 MHz
4 cores, 16 Tasks, 8 bit deadline and period counters, 8 bit stats counters	2363 (13%)	1114 (6%)	51.86 MHz
8 cores, 32 Tasks, 16 bit deadline and period counters, 16 bit stats counters	7167 (38%)	3738 (20%)	39.08 MHz

Figure 89: Synthesis results of the scheduling accelerator on an Altera Cyclone II FPGA.

5.7. Scheduling jitter analysis

The aim of the scheduling jitter analysis is to estimate the uncertainty related to the release of periodic tasks, intended as the interval between the moment in which a task release counter goes down to zero and the instant in which an interrupt is concretely dispatched to the proper processor. According to the design of the hardware scheduling accelerator, such interval depends on the following parameters:

- T_{RELQ}: time required by the *release queue* functional unit to properly issue the RELEASE_TASKS output signal once a tick is received.
- T_{RELSVC}: time required by the *release jobs* operational service to acknowledge the latter signal, update the entries in the *task table* and insert the proper elements into the *ready queue.*
- T_{CPUMAP}: time required by the *CPU mapper* functional unit to acknowledge the change of the *ready queue* and determine the new task mappings for the CPUs according to the new priorities.

- T_{INTREQ}: time required by the interrupt logic to acknowledge the latter mappings and send interrupts to the processor that need to issue scheduling changes on the running tasks.

In the current hardware implementation, such timings are defined as follows (intervals are expressed in clock cycles unless otherwise stated; *m* : number of processors, *n* : maximum number of tasks allowed):

T_{RELQ} = 1, for each *m*, *n*

T_{RELSVC} = 2 + *n*, for each *m*

T_{CPUMAP} = 4 + [1.. *m*], for each *n*

T_{INTREQ} = 1, for each *m*, *n*

Thus, $T_{REL\_TO\_INT}$ = 1 + 2 + N + 4 + [1..*m*] + 1 = 8 + *n* + [1.. *m*]

The only uncertainty, therefore, is related to the term [1..*m*] due to the *CPU mapper* operations, which complexity is linear in the number of processors (configured through the `N_CPUS` VHDL constant). Considering our test-bench in which *n*=8, *m* = 2 , Clock = 50 MHz, it translates into:

A maximum delay of 8 + 8 + 2 = 18 T_{CLK} = 360 ns.

A minimum delay of 8 + 8 + 1 = 17 T_{CLK} = 340 ns.

$T_{REL\_TO\_INT}$ = 350 **± 10 ns.**

5.8. Concluding remarks

The development of a hybrid hardware-software scheduling infrastructure, such as the one proposed in this chapter, has revealed very interesting results. In the projects discussed in the previous chapters, a pure-software implementation of a multiprocessor scheduling framework has been carried out (focused in particular on the G-EDF policy). Surprisingly, moving from a pure software implementation to hardware/software co-designing revealed to be an amazingly smooth experience, less complex than expected.

The reasons are soon clear. As regard the hardware design, the availability of a reference architecture that clearly defines the methodologies and the interaction patterns between the components, revealed to be the winning strategy for the rapid development of a clear and well-arranged system.

This, in particularly, relates to both the hardware scheduler's inner architecture and the Avalon bus interface specifications that, standing on a adaptive design strategy (i.e. increase the complexity of the component interface as long as it is really needed) and a great availability of reference documentation, made the development process of the hardware components a straightforward path.

The greatest difficulties were, undoubtedly, represented by the software layer, in particular due to the very elementary task model of μC/OS-II, which required a lot of interventions on the software layer, hindering the plain applicability of the X-RT framework as designed in chapter 3.

As a final comment, both the software and hardware performances achieved were pretty satisfying. In particular, the scheduling accelerator, despite its complexity, turned out to have an impressively low jitter and modest area requirements, especially considering the adaptability and flexibility of the design that can be seamlessly expanded just touching few lines of VHDL constants.

Of course, still some additional work would be still needed in order to enhance the hardware design of this proof-of-concept: there is still a number of long combinatorial paths which limit the overall f_{MAX} of the sequential logic (although the flexibility of the SOPC platform could allow decoupling the clock domains of the scheduler and the processors).

As future work directions, it would be definitely interesting to reproduce the experiments with a larger hardware availability, replacing the NIOS soft-cores with SMP-compliant processors, such as, for instance, the new hybrid Altera Cyclone V FPGAs which embed an ARM Cortex A9 SMP processor. Furthermore, more investigations should be undertaken as regards the scalability of this approach on massively multi-core platforms, carrying out deeper studies to highlight any bottlenecks of the hardware architecture, in particular as regards the influence and the scalability of the switch fabric.

6. Concluding remarks

This dissertation has dealt with implementative aspects of real-time schedulers in embedded multiprocessor systems, addressing two main topics: tick-less timekeeping and the implementation of global (i.e. non-partitioned) schedulers in modern RTOSs.

As regards the former, this work presented a novel data structure, called addressable binary heap (ABH), which implements all the typical operations of a priority queue using a pointer-based binary heap.

At first, the theoretical properties that underpin its physical structure and ensure logarithmic worst-case complexity to all its operations are presented. Furthermore, a complete C implementation of the data structure is discussed in its full details. The viability and the performances of the ABH have been evaluated on an instruction-accurate virtual platform simulator, comparing other data structure typically employed for this purpose, such as various implementations of array-backed binary heaps and self-balancing binary trees. In the experimental evaluations, the ABH demonstrated to be a very good candidate for tackling timekeeping problems in a highly deterministic manner, as its worst-case behavior is almost always better than all the other data structures analyzed.

The ABH data structure has been employed as a building block for the realization of a scheduling framework called X-RT, which represents the second major contribution of this thesis. Such framework aims at providing runtime facilities for the development of real-time applications, by means of a user-space runtime library which is able to implement global scheduling policies, such as G-EDF, in the mainstream RTOSs exploiting their priority-driven scheduler.

Some other research works in this field addressed the same problem by means of interventions on the operating-system kernel. The work presented in this thesis, instead, adopts a different approach, based on a user-space framework. The X-RT framework, released in the form of an open-source project [TUC2012], showed, in fact, as these advanced real-time multiprocessor scheduling policies can be realized without modifying the RTOS kernel. The experimental work undertaken highlights that the

performances of this approach are, both in terms of raw overhead and schedulability tests, very close to kernel-level approaches.

The principles that underpin the operation of this framework, originally designed for symmetric multiprocessors, have been further extended to asymmetric ones. Such platforms are typically subjected to major restrictions, such as the lack of support for task migrations, hindering the plain applicability of global scheduling policies. However, by introducing some limitations on the scheduling model, such as restricting the granularity of migrations to job boundaries, interesting results can be still obtained.

Finally, the last chapter of this thesis investigates the world of re-programmable hardware platforms, notably FPGAs, presenting a scheduling accelerator, which offloads most of the scheduling operations to the hardware and exhibits extremely low scheduling jitter.

Future research directions

As regards the ABH data structure, more experimental investigations should be carried out on actual hardware platforms, in order to assess the validity of the results in presence of caches, branch predictors and super-scalar pipelines, which are not modeled by the platform simulator used in the experiments.

The X-RT scheduling framework have revealed satisfactory results in the schedulability tests, in many cases very close to the theoretical bounds known for the G-EDF policy. However, it has to be underlined that the tasks of the experimental workbench were simulating pure CPU-bound processes, without performing any memory or I/O access. More interesting results could be obtained taking into account these factors, as has been recently done in some of the research work cited, in order to validate the behavior of the scheduling framework in presence of cache-hotness effects.

As regards the last point of this thesis, the hardware scheduling accelerator, more investigations should be undertaken as regards the integration of the accelerator with other bus-master peripherals in the system. In fact, while the scheduling accelerator guarantees extremely predictable timings for the

computation of scheduling decisions, in many scenarios this is not sufficient to guarantee a deterministic behavior of the overall system.

In fact, the presence of other bus-masters in the system other than the CPU, notably DMA controllers, can delay the execution of tasks as soon as they try to access memory. In general this problem is difficult to address in traditional systems where scheduling is a purely software activity. However, the introduction of a hardware scheduler paves the way to the synergic coordination of hardware peripherals, as all the information about task timings are now available in hardware. This, however, would require major interventions on the architecture of the bus/switching fabric, but, in theory at least, could contribute increasing the determinism of real-time embedded systems, by means of enforcing scheduling decisions also to hardware peripherals, arbitrating the bus accesses in a deadline-aware fashion.

Bibliography

[AB1998]: ABENI, L. & BUTTAZZO, G.: Integrating multimedia applications in hard real-time systems. In:: . : Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE., 1998, S. 4--13

[AL1963]: ADELSONVELSKII, M. & LANDIS, E.: An algorithm for the organization of information: Defense Technical Information Center., 1963

[AFLS1996]: ADOMAT, J.; FURUNAS, J.; LINDH, L. & STARNER, J.: Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems. In:: . : Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on., 1996, S. 164--168

[AS2005]: ALBERS, K. & SLOMKA, F.: Efficient feasibility analysis for real-time systems with EDF scheduling. In:: . : Design, Automation and Test in Europe, 2005. Proceedings., 2005, S. 492--497

[Alt2010]: ALTERA: Nios II Processor Reference - Handbook, Altera Corp., 2010

[Alt2009]: ALTERA: Mailbox Core, Quartus II 9.1 Handbook, Online, 2009

[Alt2003]: ALTERA: Avalon Bus Specification Reference Manual, Altera Corp., 2003

[Alt2002]: ALTERA: Simultaneous Multi-Mastering with the Avalon Bus, Altera Corp., 2002

[ABD2005]:ANDERSON, J. H.; BUD, V. & DEVI, U. C.: An EDF-based scheduling algorithm for multiprocessor soft real-time systems, 2005

[ABJ2001]: ANDERSSON, B.; BARUAH, S. & JONSSON, J.: Static-priority scheduling on multiprocessors. In:: . : Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE., 2001, S. 193--202

[ANA2004]:ANDREWS, D.; NIEHAUS, D. & ASHENDEN, P.: Programming models for hybrid CPU/FPGA chips. In: Computer 37 (2004), Nr. 1, S. 118--120

[ANJ+2004]:ANDREWS, D.; NIEHAUS, D.; JIDIN, R.; FINLEY, M.; PECK, W.; FRISBIE, M.; ORTIZ, J.; KOMP, E. & ASHENDEN, P.: Programming models for hybrid FPGA-cpu computational components: a missing link. In: IEEE Micro 24 (2004), Nr. 4, S. 42--53

[Bak2006]: BAKER, T.: An analysis of fixed-priority schedulability on a multiprocessor. In: Real-Time Systems 32 (2006), Nr. 1, S. 49--71

[Bak2003]: BAKER, T.: An Analysis of Deadline-Monotonic Schedulability on a Multiprocessor, 2003

[Bak2003a]: BAKER, T.: Multiprocessor EDF and deadline monotonic schedulability analysis. In:: . : Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE., 2003, S. 120--129

[BB2009]: BAKER, T. & BARUAH, S.: An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. In: Real-Time Systems 43 (2009), Nr. 1, S. 3--24

[Bak2005]: BAKER, T. P.: A comparison of global and partitioned EDF schedulability tests for multiprocessors, Bericht, In International Conf. on Real-Time and Network Systems, 2005

[Bak2005b]: BAKER, T. P.: Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time, 2005

[Bak2005c]:BAKER, T. P.: Further improved schedulability analysis of EDF on multiprocessor platforms, 2005

[Bar2007]:BARUAH, S.: Techniques for multiprocessor global schedulability analysis. In:: . : Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International., 2007, S. 119--128

[Bar2006]:BARUAH, S.: The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. In: Real-Time Systems 32 (2006), S. 9-20

[BBMS2010]: BARUAH, S.; BONIFACI, V.; MARCHETTI-SPACCAMELA, A. & STILLER, S.: Improved multiprocessor global schedulability analysis. In: Real-Time Systems 46 (2010), Nr. 1, S. 3--24

[BC2003]: BARUAH, S. & CARPENTER, J.: Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In:: . : Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on., 2003, S. 195 - 202

[Bas2011]:BASTONI, A.: Towards the integration of theory and practice in multiprocessor real-time scheduling. In: PhD in computer science and automation engineering, University of Rome" Tor Vergata (2011)

[BBA2011]: BASTONI, A.; BRANDENBURG, B. & ANDERSON, J.: Is semi-partitioned scheduling practical?. In:: . : Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on., 2011, S. 125--135

[BBA2010]: BASTONI, A.; BRANDENBURG, B. & ANDERSON, J.: An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In:: . : Real-Time Systems Symposium (RTSS), 2010 IEEE 31st., 2010, S. 14--24

[BBA2010a]: BASTONI, A.; BRANDENBURG, B. & ANDERSON, J.: Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In:: . : Proc. of the 6th Int'l Workshop on Operating Sys. Platforms for Embedded Real-Time Apps., 2010, S. 33--44

[BOBSBS2008]:BEN OTHMAN, S.; BEN SALEM, A. K. & BEN SAOUD, S.: MPSoC design of RT control applications based on FPGA SoftCore processors. In:: . : Proc. 15th IEEE Int. Conf. Electronics, Circuits and Systems ICECS 2008., 2008, S. 404--409

[vBer2009]: VAN BERKEL, C. H.: Multi-core for mobile phones. In:: . : Proc. DATE '09. Design, Automation & Test in Europe Conf. & Exhibition., 2009, S. 1260--1265

[BCL2009]: BERTOGNA, M.; CIRINEI, M. & LIPARI, G.: Schedulability analysis of global scheduling algorithms on multiprocessor platforms. In: Parallel and Distributed Systems, IEEE Transactions on 20 (2009), Nr. 4, S. 553--566

[BCL2005]: BERTOGNA, M.; CIRINEI, M. & LIPARI, G.: Improved schedulability analysis of EDF on multiprocessor platforms. In:: . : Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on., 2005, S. 209--218

[BBCG2008]: BETTI, E.; BOVET, D.; CESATI, M. & GIOIOSA, R.: Hard real-time performances in multiprocessor-embedded systems using ASMP-Linux. In: EURASIP Journal of Embedded Systems 2008 (2008), S. 1--16

[BM2006]:BJERREGAARD, T. & MAHADEVAN, S.: A survey of research and practices of network-on-chip. In: ACM Computing Surveys (CSUR) 38 (2006), Nr. 1, S. 1

[BA2009]: BRANDENBURG, B. & ANDERSON, J.: On the implementation of global real-time schedulers. In:: . : Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE., 2009, S. 214--224

[BBC+2007]: BRANDENBURG, B.; BLOCK, A.; CALANDRINO, J.; DEVI, U.; LEONTYEV, H. & ANDERSON, J.: LITMUSRT: A status report. In:: . : Proceedings of the 9th real-time Linux workshop., 2007, S. 107--123

[BCA2008]: BRANDENBURG, B.; CALANDRINO, J. & ANDERSON, J.: On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In:: . : Real-Time Systems Symposium, 2008., 2008, S. 157 -169

[BLA2009]: BRANDENBURG, B.; LEONTYEV, H. & ANDERSON, J.: Accounting for interrupts in multiprocessor real-time systems. In:: . : Proceedings of the 15th IEEE international conference on embedded and real-time computing systems and applications., 2009, S. 273--283

[Bro1988]:BROWN, R.: Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. In: Communications of the ACM 31 (1988), Nr. 10, S. 1220--1227

[BLOS1995]: BURCHARD, A.; LIEBEHERR, J.; OH, Y. & SON, S.: New strategies for assigning real-time tasks to multiprocessor systems. In: Computers, IEEE Transactions on 44 (1995), Nr. 12, S. 1429--1442

[BKN+1999]: BURLESON, W.; KO, J.; NIEHAUS, D.; RAMAMRITHAM, K.; STANKOVIC, J.; WALLACE, G. & WEEMS, C.: The spring scheduling coprocessor: a scheduling accelerator. In: Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 7 (1999), Nr. 1, S. 38--47

[But2011]: BUTTAZZO, G.: Hard real-time computing systems: predictable scheduling algorithms and applications: Springer., 2011

[But2005]: BUTTAZZO, G.: Rate monotonic vs. EDF: judgment day. In: Real-Time Systems 29 (2005), Nr. 1, S. 5--26

[BLAC2005]: BUTTAZZO, G.; LIPARI, G.; ABENI, L. & CACCAMO, M.: Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency: Springer., 2005

[CA2009]: CALANDRINO, J. & ANDERSON, J.: On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler. In:: . : Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on., 2009, S. 194 - 204

[CLB+2006]:CALANDRINO, J.; LEONTYEV, H.; BLOCK, A.; DEVI, U. & ANDERSON, J.: LITMUS RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In:: . : Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International., 2006, S. 111--126

[CGKS2005]: CHANDRA, D.; GUO, F.; KIM, S. & SOLIHIN, Y.: Predicting inter-thread cache contention on a chip multi-processor architecture. In:: . : High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on., 2005, S. 340--351

[CRL2006]: CHANDRA, S.; REGAZZONI, F. & LAJOLO, M.: Hardware/software partitioning of operating systems: a behavioral synthesis approach. In:: . : Proceedings of the 16th ACM Great Lakes symposium on VLSI., 2006, S. 324--329

[Chi2005]: CHIANG, M.: Balancing transport and physical layers in wireless multihop networks: Jointly optimal congestion control and power control. In: Selected Areas in Communications, IEEE Journal on 23 (2005), Nr. 1, S. 104--116

[CI2001]: CHILDS, S. & INGRAM, D.: The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In:: . : Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE., 2001, S. 135--140

[CY2012]: CHISHIRO, H. & YAMASAKI, N.: Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems. In:: . : Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on., 2012, S. 127--134

[CSR1993]: CHUNG, K.; SANG, J. & REGO, V.: A performance comparison of event calendar algorithms: an empirical approach. In: Software: Practice and Experience 23 (1993), Nr. 10, S. 1107--1138

[CJGJ1978]: COFFMAN JR, E.; GAREY, M. & JOHNSON, D.: An application of bin-packing to multiprocessor scheduling. In: SIAM Journal on Computing 7 (1978), Nr. 1, S. 1--17

[Com1984]: COMFORT, J.: The simulation of a master-slave event set processor. In: Simulation 42 (1984), Nr. 3, S. 117--124

[Con2009]: CONSORTIUM, T. E. M. B.: EEMBC Benchmark Suite, Online, 2009

[CG2011]: CUCINOTTA, T. & GOGOUVITIS, S.: Real-Time Attributes in Operating Systems. In: Achieving Real-Time in Distributed Computing: From Grids to Clouds. IGI Global (2011)

[Cur2006]: CURTIS, K. E.: Embedded Multitasking (Embedded Technology): Newnes., 2006

[DCC2007]: DAVID, F.; CARLYLE, J. & CAMPBELL, R.: Context switch overheads for Linux on ARM platforms. In:: . : Proceedings of the 2007 workshop on Experimental computer science., 2007, S. 3

[DB2011]: DAVIS, R. & BURNS, A.: A survey of hard real-time scheduling for multiprocessor systems. In: ACM Computing Surveys (CSUR) 43 (2011), Nr. 4, S. 35

[DKS1989]: DEMERS, A.; KESHAV, S. & SHENKER, S.: Analysis and simulation of a fair queueing algorithm. In:: . 19, Nr. 4 : ACM SIGCOMM Computer Communication Review., 1989, S. 1--12

[DA2008]: DEVI, U. & ANDERSON, J.: Tardiness bounds under global EDF scheduling on a multiprocessor. In: Real-Time Systems 38 (2008), S. 133-189

[DL1978]: DHALL, S. & LIU, C.: On a real-time scheduling problem. In: Operations Research 26 (1978), Nr. 1, S. 127--140

[DW2005]: DIETRICH, S. & WALKER, D.: The evolution of real-time linux. In:: . : Proc. 7th Real-Time Linux Workshop., 2005, S. 3--4

[DNB2007]:DI NATALE, M. & BINI, E.: Optimizing the FPGA Implementation of HRT Systems. In:: . : Proc. 13th IEEE Real Time and Embedded Technology and Applications Symp. RTAS '07., 2007, S. 22--31

[DJM+2009]: DORTA, T.; JIMENEZ, J.; MARTIN, J. L.; BIDARTE, U. & ASTARLOA, A.: Overview of FPGA-Based Multiprocessor Systems. In:: . : Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig '09., 2009, S. 273--278

[DM2003]: DREPPER, U. & MOLNAR, I.: The native POSIX thread library for Linux. In: White Paper, Red Hat (2003)

[DTH1992]:DUPUY, S.; TAWBI, W. & HORLAIT, E.: Protocols for high-speed multimedia communications networks. In: Computer Communications 15 (1992), Nr. 6, S. 349--358

[DJR2001]: DUTTA, S.; JENSEN, R. & RIECKMANN, A.: Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. In: IEEE Des Test Comput 18 (2001), Nr. 5, S. 21--31

[EDB2010]:ERICKSON, J.; DEVI, U. & BARUAH, S.: Improved tardiness bounds for global EDF. In:: . : Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on., 2010, S. 14--23

[Ern1998]:ERNST, R.: Codesign of embedded systems: status and trends. In: IEEE Des Test Comput 15 (1998), Nr. 2, S. 45--54

[eST2009]:ESTW: embedded Single Timer Wheel (eSTW), Online, 2009

[FCTS2009]: FAGGIOLI, D.; CHECCONI, F.; TRIMARCHI, M. & SCORDINO, C.: An EDF scheduling class for the Linux kernel. In:: . : Proc. of the Real-Time Linux Workshop., 2009

[FTC2009]: FAGGIOLI, D.; TRIMARCHI, M. & CHECCONI, F.: An implementation of the earliest deadline first algorithm in linux. In:: . : Proceedings of the 2009 ACM symposium on Applied Computing., 2009, S. 1984--1989

[FJM+1995]: FLOYD, S.; JACOBSON, V.; MCCANNE, S.; LIU, C. & ZHANG, L.: A reliable multicast framework for light-weight sessions and application level framing. In:: . 25, Nr. 4 : ACM SIGCOMM Computer Communication Review., 1995, S. 342--356

[FCKN2007]: FREITAS, H.; COLOMBO, D.; KASTENSMIDT, F. & NAVAUX, P.: Evaluating Network-on-Chip for Homogeneous Embedded Multiprocessors in FPGAs. In:: . : Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on., 2007, S. 3776 -3779

[FB2004]: FUNK, S. & BARUAH, S.: Restricting EDF migration on uniform multiprocessors. In:: . : Proceedings of the 12th International Conference on Real-Time Systems., 2004

[GJ1979]: GAREY, M. & JOHNSON, D.: Computers and intractability: Freeman San Francisco, CA. 174, 1979

[Gee2004]: GEER, D.: Survey: Embedded Linux Ahead of the Pack. In: IEEE Distributed Systems Online (2004)

[GN2006]: GLEIXNER, T. & NIEHAUS, D.: Hrtimers and beyond: Transforming the linux time subsystems. In:: . 1 : Linux Symposium., 2006, S. 333--346

[GFB2003]: GOOSSENS, J.; FUNK, S. & BARUAH, S.: Priority-driven scheduling of periodic task systems on multiprocessors. In: Real-Time Systems 25 (2003), Nr. 2, S. 187--205

[HP2008]: HARDY, D. & PUAUT, I.: WCET analysis of multi-level non-inclusive set-associative instruction caches. In:: . : Real-Time Systems Symposium, 2008., 2008, S. 456--466

[HGT1999]: HILDEBRANDT, J.; GOLATOWSKI, F. & TIMMERMANN, D.: Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. In:: . : Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on., 1999, S. 208--215

[HN1998]: HINTZE, J. & NELSON, R.: Violin plots: a box plot-density trace synergism. In: The American Statistician 52 (1998), Nr. 2, S. 181--184

[HMo2006]: HOROWITZ, E.; MEHTA, D. & OTHERS: Fundamentals of data structures in C++: Galgotia Publications., 2006

[HPB2005]: HUBNER, M.; PAULSSON, K. & BECKER, J.: Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores. In:: . : Proc. 19th IEEE Int. Parallel and Distributed Processing Symp., 2005

[HCMP2007]: HUERTA, P.; CASTILLO, J.; MARTINEZ, J. & PEDRAZA, C.: Exploring fpga capabilities for building symmetric multiprocessor systems. In:: . : Programmable Logic, 2007. SPL'07. 2007 3rd Southern Conference on., 2007, S. 113--118

[HCP+2009]: HUERTA, P.; CASTILLO, J.; PEDRAZA, C.; CANO, J. & MARTINEZ, J. I.: Symmetric Multiprocessor Systems on FPGA. In:: IEEE Computer Society. 0, 2009, S. 279-283

[HBK2005]:HUNG, A.; BISHOP, W. & KENNINGS, A.: Symmetric multiprocessing on programmable chips made easy. In:: . : Proc. Design, Automation and Test in Europe., 2005, S. 240--245

[IBM2001]: IBM: SA-14-2528-02 On-Chip Peripheral Bus Architecture Speci?cations, Online, 2001

[JCASB2004]: J. CARPENTER, S. FUNK, P. H.; A. SRINIVASAN, J. A. & BARUAH, S.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. In: LEUNG, J. Y. (Hrsg.), Boca Raton, Florida: Chapman and Hall/CRC., 2004, S. 30-1 - 30-19

[JLS+2010]: JALIER, C.; LATTARD, D.; SASSATELLI, G.; BENOIT, P. & TORRES, L.: Flexible and distributed real-time control on a 4G telecom MPSoC. In:: . : Proc. IEEE Int Circuits and Systems (ISCAS) Symp., 2010, S. 3961--3964

[JBP2006]: JERRAYA, A. A.; BOUCHHIMA, A. & PETROT, F.: Programming models and HW-SW interfaces abstraction for multi-processor SoC. In:: . : Proc. 43rd ACM/IEEE Design Automation Conf., 2006, S. 280--285

[Jon1986]:JONES, D.: An empirical comparison of priority-queue and event-set implementations. In: Communications of the ACM 29 (1986), Nr. 4, S. 300--311

[JS2006]:JOOST, R. & SALOMON, R.: Advantages of FPGA-based multiprocessor systems in industrial applications. In:: . : Industrial Electronics Society, 2005. IECON 2005. 31st Annual Conference of IEEE., 2006, S. 6

[JCR2007]: JU, L.; CHAKRABORTY, S. & ROYCHOUDHURY, A.: Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In:: . : Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07., 2007, S. 1--6

[KM2005]:KARGAHI, M. & MOVAGHAR, A.: Non-preemptive earliest-deadline-first scheduling policy: a performance study. In:: . : Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on., 2005, S. 201 - 208

[Ken1996]: KENYON, C.: Best-fit bin-packing with random order. In:: . : Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms., 1996, S. 359--364

[KBDV2006]: KIM, M.; BANERJEE, S.; DUTT, N. & VENKATASUBRAMANIAN, N.: Design space exploration of real-time multi-media MPSoCs with heterogeneous scheduling policies. In:: . : Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference., 2006, S. 16 -21

[Kin1985]:KINGSTON, J.: Analysis of tree algorithms for the simulation event list. In: Acta Informatica 22 (1985), Nr. 1, S. 15--33

[KEH+2009]: KLEIN, G.; ELPHINSTONE, K.; HEISER, G.; ANDRONICK, J.; COCK, D.; DERRIN, P.; ELKADUWE, D.; ENGELHARDT, K.; KOLANSKI, R.; NORRISH, M. & OTHERS: seL4: Formal verification of an OS kernel. In:: . : Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles., 2009, S. 207--220

[Knu2006]: KNUTH, D.: The art of computer programming: addison-Wesley., 2006

[KGJ2003]: KOHOUT, P.; GANESH, B. & JACOB, B.: Hardware support for real-time operating systems. In:: . : Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on., 2003, S. 45--51

[KS1995]: KOREN, G. & SHASHA, D.: Skip-over: Algorithms and complexity for overloaded systems that allow skips. In:: . : Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE., 1995, S. 110--117

[LGDG2000]: LÓPEZ, J.; GARCIA, M.; DIAZ, J. & GARCIA, D.: Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In:: . : Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on., 2000, S. 25—33

[LJ2009]: Real-Time Linux Kernel Scheduler. Linux Journal. August 2009. Issue 184. Online.

[LRL2009]: LAKSHMANAN, K.; RAJKUMAR, R. & LEHOCZKY, J.: Partitioned fixed-priority preemptive scheduling for multi-core processors. In:: . : Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on., 2009, S. 239--248

[Lam2001]: LAMPRET, D.: OpenRISC 1200 IP Core specification, Online, 2001

[LMM1998]: LAUZAC, S.; MELHEM, R. & MOSSE, D.: Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In:: . : Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on., 1998, S. 188 -195

[Lee2000]: LEE, E. A.: What's ahead for embedded software?. In: Computer 33 (2000), Nr. 9, S. 18--26

[LMID+2003]: LEE, J.; MOONEY III, V.; DALEBY, A.; INGSTRÖM, K.; KLEVIN, T. & LINDH, L.: A comparison of the RTU hardware RTOS with a hardware/software RTOS. In:: . : Proceedings of the 2003 Asia and South Pacific Design Automation Conference., 2003, S. 683--688

[LSD1989]: LEHOCZKY, J.; SHA, L. & DING, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In:: . : Real Time Systems Symposium, 1989., Proceedings.., 1989, S. 166--171

[LSK+2005]: LEHTORANTA, O.; SALMINEN, E.; KULMALA, A.; HANNIKAINEN, M. & HAMALAINEN, T.: A parallel MPEG-4 encoder for FPGA based multiprocessor SoC. In:: . : Field Programmable Logic and Applications, 2005. International Conference on., 2005, S. 380 - 385

[LFCL2012]: LELLI, J.; FAGGIOLI, D.; CUCINOTTA, T. & LIPARI, G.: An experimental comparison of different real-time schedulers on multicore systems. In: Journal of Systems and Software (2012)

[LHPo2009]: LESAGE, B.; HARDY, D.; PUAUT, I. & OTHERS: WCET analysis of multi-level set-associative data caches. In:: . : 9th Intl. Workshop on Worst-Case Execution Time WCET Analysis., 2009

[Let2008]: LETEINTURIER, P. BREWERTON, S. S. K.: MultiCore Benefits & Challenges for Automotive Applications. In: SAE SP - SOCIETY OF AUTOMOTIVE ENGINEERS INC. NUMB 2159 (2008), S. 103-114

[LW1982]: LEUNG, J. & WHITEHEAD, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. In: Performance evaluation 2 (1982), Nr. 4, S. 237--250

[LDS2007]: LI, C.; DING, C. & SHEN, K.: Quantifying the cost of context switch. In:: . : Proceedings of the 2007 workshop on Experimental computer science., 2007, S. 2

[LRSF2004]: LI, P.; RAVINDRAN, B.; SUHAIB, S. & FEIZABADI, S.: A formally verified application-level framework for real-time scheduling on posix real-time

operating systems. In: Software Engineering, IEEE Transactions on 30 (2004), Nr. 9, S. 613--629

[LBKH2007]: LI, T.; BAUMBERGER, D.; KOUFATY, D. & HAHN, S.: Efficient operating system scheduling for performance-asymmetric multi-core architectures. In:: . : Proceedings of the 2007 ACM/IEEE conference on Supercomputing., 2007, S. 53

[LT2009]: LIGGESMEYER, P. & TRAPP, M.: Trends in Embedded Software Engineering. In: IEEE Software 26 (2009), Nr. 3, S. 19--25

[Lin1992]: LINDH, L.: Fasthard-a fast time deterministic hardware based real-time kernel. In:: . : Real-Time Systems, 1992. Proceedings., Fourth Euromicro workshop on., 1992, S. 21--25

[Lin1991]: LINDH, L.: Fastchart-a fast time deterministic cpu and hardware based real-time-kernel. In:: . : Real Time Systems, 1991. Proceedings., Euromicro'91 Workshop on., 1991, S. 36--40

[LL1973]: LIU, C. & LAYLAND, J.: Scheduling algorithms for multiprogramming in a hard-real-time environment. In: Journal of the ACM (JACM) 20 (1973), Nr. 1, S. 46--61

[Liu2000]: LIU, J.: Real-time systems: Prentice Hall PTR., 2000

[LDN1997]:LUCULLI, G. & DI NATALE, M.: A cache-aware scheduling algorithm for embedded systems. In:: . : Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE., 1997, S. 199 -209

[Mac1980]: MACLAREN, L.: Evolving toward Ada in real time systems. In:: . 15, Nr. 11 : ACM Sigplan Notices., 1980, S. 146--155

[MEB1988]: MAJUMDAR, S.; EAGER, D. L. & BUNT, R. B.: Scheduling in multiprogrammed parallel systems. In: SIGMETRICS Perform. Eval. Rev. 16 (1988), S. 104--113

[Mar2006]: MARTIN, G.: Overview of the MPSoC design challenge. In:: . : Proc. 43rd ACM/IEEE Design Automation Conf., 2006, S. 274--279

[MIY2010]: MARUYAMA, N.; ISHIHARA, T. & YASUURA, H.: An RTOS in hardware for energy efficient software-based TCP/IP processing. In:: . : Application Specific Processors (SASP), 2010 IEEE 8th Symposium on., 2010, S. 58--63

[MCF2010]: MASRUR, A.; CHAKRABORTY, S. & FÄRBER, G.: Constant-time admission control for deadline monotonic tasks. In:: . : Proceedings of the Conference on Design, Automation and Test in Europe., 2010, S. 220--225

[MDF2008]: MASRUR, A.; DRÖSSLER, S. & FÄRBER, G.: Improvements in polynomial-time feasibility testing for EDF. In:: . : Proceedings of the conference on Design, automation and test in Europe., 2008, S. 1033--1038

[MC2004]:MATTSSON, D. & CHRISTENSSON, M.: Evaluation of synthesizable CPU cores. In: Master's thesis, Department of Computer Engineering, Chalmers University of Technology (2004)

[MS1981]: MCCORMACK, W. & SARGENT, R.: Analysis of future event set algorithms for discrete event simulation. In: Communications of the ACM 24 (1981), Nr. 12, S. 801--812

[MST1994]: MERCER, C.; SAVAGE, S. & TOKUDA, H.: Processor capacity reserves: Operating system support for multimedia applications. In:: . : Multimedia Computing and Systems, 1994., Proceedings of the International Conference on., 1994, S. 90--99

[MB1991]:MOGUL, J. & BORG, A.: The effect of context switches on cache performance: ACM. 26, Nr. 4, 1991

[MHSM2009]: MOLKA, D.; HACKENBERG, D.; SCHONE, R. & MULLER, M.: Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In:: . : Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on., 2009, S. 261--270

[MIB2002]: MOONEY III, V. & BLOUGH, D.: A hardware-software real-time operating system framework for SoCs. In: Design & Test of Computers, IEEE 19 (2002), Nr. 6, S. 44--51

[NUI+1995]: NAKANO, T.; UTAMA, A.; ITABASHI, M.; SHIOMI, A. & IMAI, M.: Hardware implementation of a real-time operating system. In:: . : TRON Project International Symposium, 1995., Proceedings of the 12th., 1995, S. 34--42

[NNB2010]:NEMATI, F.; NOLTE, T. & BEHNAM, M.: Partitioning real-time systems on multiprocessors with shared resources. In: Principles of Distributed Systems (2010), S. 253--269

[NVC2010]:NOLLET, V.; VERKEST, D. & CORPORAAL, H.: A Safari Through the MPSoC Run-Time Management Jungle. In: Journal of Signal Processing Systems 60 (2010), S. 251-268

[NA2007]: NORDSTROM, S. & ASPLUND, L.: Configurable hardware/software support for single processor real-time kernels. In:: . : System-on-Chip, 2007 International Symposium on., 2007, S. 1--4

[NLJS2005]: NORDSTROM, S.; LINDH, L.; JOHANSSON, L. & SKOGLUND, T.: Application specific real-time microkernel in hardware. In:: . : Real Time Conference, 2005. 14th IEEE-NPSS., 2005, S. 4--pp

[OKP2010]:OBERMAISSER, R.; KOPETZ, H. & PAUKOVITS, C.: A Cross-Domain Multiprocessor System-on-a-Chip for Embedded Real-Time Systems. In: IEEE Transactions on Industrial Informatics 6 (2010), Nr. 4, S. 548--567

[Ope2002]: OPENCORES: Wishbone Bus Specifications Revision B.3, Online, 2002

[OVP2012]:OVPSIM, O. V. P.: , Online, 2012

[PSJC+1997]: PARISOTO, A.; SOUZA JR, A.; CARRO, L.; PONTREMOLI, M.; PEREIRA, C. & SUZIM, A.: F-Timer: Dedicated FPGA to real-time systems design support. In:: . : Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on., 1997, S. 35--40

[PLC+1997]:PAULIN, P. G.; LIEM, C.; CORNERO, M.; NACABAL, F. & GOOSSENS, G.: Embedded software in real-time signal processing systems: application and architecture trends. In: Proceedings of the IEEE 85 (1997), Nr. 3, S. 419--435

[PSG1998]: PUESCHEL, H.; SCHMIDT, G. & GERDES, M.: Method and device for controlling an ABS antilock braking/ASR traction control system: Google Patents., 1998

[RA1997]: RÖNNGREN, R. & AYANI, R.: A comparative study of parallel and sequential priority queue algorithms. In: ACM Transactions on Modeling and Computer Simulation (TOMACS) 7 (1997), Nr. 2, S. 157--209

[RAFD1993]: RÖNNGREN, R.; AYANI, R.; FUJIMOTO, R. & DAS, S.: Efficient implementation of event sets in Time Warp. In:: . 23, Nr. 1 : ACM SIGSIM Simulation Digest., 1993, S. 101--108

[RSJK2005]: RAVINDRAN, K.; SATISH, N.; JIN, Y. & KEUTZER, K.: An FPGA-based soft multiprocessor system for IPv4 packet forwarding. In:: . : Field Programmable Logic and Applications, 2005. International Conference on., 2005, S. 487 - 492

[RW2010]:REICHENBACH, F. & WOLD, A.: Multi-core Technology – Next Evolution Step in Safety Critical Systems for Industrial Applications?. In:: . :

Proc. 13th Euromicro Conf. Digital System Design: Architectures, Methods and Tools (DSD)., 2010, S. 339--346

[RA1993]: RÖNNGREN, R., R. J. & AYANI, R.: Lazy queue: New approach to implementing the pending event set.. In: Int. J. Comput. Simul 3 (1993), S. 303-332

[Ros2004]:ROSINGER, H.: Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link (FSL) channel. In: Xilinx Application Note (2004)

[SVC1998]: SÁEZ, S.; VILA, J. & CRESPO, A.: Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In:: . : Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on., 1998, S. 53--60

[Sch2007]:SCHIRRMEISTER, F.: Multi-core Processors: Fundamentals, Trends, and Challenges. In:: . : Imperas, Embedded Systems Conference (ESC)., 2007

[Sch2002]:SCHMIDT, D. C.: Middleware for real-time and embedded systems. In: Commun. ACM 45 (2002), S. 43--48

[SRL1990]: SHA, L.; RAJKUMAR, R. & LEHOCZKY, J.: Priority inheritance protocols: An approach to real-time synchronization. In: Computers, IEEE Transactions on 39 (1990), Nr. 9, S. 1175--1185

[SPV2007]: SIDDHA, S.; PALLIPADI, V. & VEN, A.: Getting maximum mileage out of tickless. In:: . 2 : Linux Symposium., 2007, S. 201--207

[ST1985]: SLEATOR, D. & TARJAN, R.: Self-adjusting binary search trees. In: Journal of the ACM (JACM) 32 (1985), Nr. 3, S. 652--686

[SA2004]: STÄRNER, J. & ASPLUND, L.: Measuring the cache interference cost in preemptive real-time systems. In:: . 39, Nr. 7 : ACM SIGPLAN Notices., 2004, S. 146--154

[Sta1988]: STANKOVIC, J.: Misconceptions about real-time computing: A serious problem for next-generation systems. In: Computer 21 (1988), Nr. 10, S. 10--19

[SR2004]: STANKOVIC, J. A. & RAJKUMAR, R.: Real-Time Operating Systems. In: Real-Time Syst. 28 (2004), S. 237--253

[SR1991]: STANKOVIC, J. & RAMAMRITHAM, K.: The Spring kernel: A new paradigm for real-time systems. In: Software, IEEE 8 (1991), Nr. 3, S. 62--72

[SK2011]: STAVRINIDES, G. & KARATZA, H.: Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques. In: Simulation Modelling Practice and Theory 19 (2011), Nr. 1, S. 540--552

[SWP2004]:STEIGER, C.; WALDER, H. & PLATZNER, M.: Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. In: Computers, IEEE Transactions on 53 (2004), Nr. 11, S. 1393--1407

[Ste1994]: STEINMAN, J.: Discrete-event simulation and the event horizon. In:: . 24, Nr. 1 : ACM SIGSIM Simulation Digest., 1994, S. 39--49

[SBas1994]:SWAGATO BASUMALLICK, K. N.: Cache Issues in Real-Time Systems, 1994

[THT2008]: TOMIYAMA, H.; HONDA, S. & TAKADA, H.: Real-time operating systems for multicore embedded systems. In:: . 01 : Proc. Int. SoC Design Conf. ISOCC '08., 2008

[TAK2006]: TONG, J.; ANDERSON, I. & KHALID, M.: Soft-Core Processors for Embedded Systems. In:: . : Microelectronics, 2006. ICM '06. International Conference on., 2006, S. 170 -173

[Tsa2007]:TSAFRIR, D.: The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In:: . : Proceedings of the 2007 workshop on Experimental computer science., 2007, S. 4

[Tsa2007a]:TSAFRIR, D.: The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In:: . : Proceedings of the 2007 workshop on Experimental computer science., 2007, S. 4

[Tuc2012]:TUCCI, P.: X-RT scheduling framework., Online: https://sourceforge.net/p/xrt/

[TBC+2008]:TUMEO, A.; BRANCA, M.; CAMERINI, L.; CERIANI, M.; PALERMO, G.; FERRANDI, F.; SCIUTO, D. & MONCHIERO, M.: A dual-priority real-time multiprocessor system on FPGA for automotive applications. In:: ACM. : Proceedings of the conference on Design, automation and test in Europe., 2008, S. 1039--1044

[VL1997]: VARGHESE, G. & LAUCK, A.: Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. In: Networking, IEEE/ACM Transactions on 5 (1997), Nr. 6, S. 824--834

[VL1987]: VARGHESE, G. & LAUCK, T.: Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In:: . 21, Nr. 5 : ACM SIGOPS Operating Systems Review., 1987, S. 25--38

[VD1975]: VAUCHER, J. & DUVAL, P.: A comparison of simulation event list algorithms. In: Communications of the ACM 18 (1975), Nr. 4, S. 223--230

[VOM+2006]: VETROMILLE, M.; OST, L.; MARCON, C.; REIF, C. & HESSEL, F.: Rtos scheduler implementation in hardware and software for real time applications. In:: . : Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on., 2006, S. 163--168

[VHag2005]: VON HAGEN, W.: Real-Time and Performance Improvements in the 2. 6 Linux Kernel. In: Linux Journal 134 (2005), S. 8

[WL1998]: WANG, Y. & LIN, K.: Enhancing the real-time capability of the Linux kernel. In:: . : Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on., 1998, S. 11--20

[WEE+2008]: WILHELM, R.; ENGBLOM, J.; ERMEDAHL, A.; HOLSTI, N.; THESING, S.; WHALLEY, D.; BERNAT, G.; FERDINAND, C.; HECKMANN, R.; MITRA, T. & OTHERS: The worst-case execution-time problem—overview of methods and survey of tools. In: ACM Transactions on Embedded Computing Systems (TECS) 7 (2008), Nr. 3, S. 36

[Wol2004]: WOLF, W.: The future of multiprocessor systems-on-chips. In:: ACM. : Proceedings of the 41st annual Design Automation Conference., 2004, S. 681--685

[WJM2008]: WOLF, W.; JERRAYA, A. A. & MARTIN, G.: Multiprocessor System-on-Chip (MPSoC) Technology. In: IEEE T Comput Aid D 27 (2008), Nr. 10, S. 1701--1713

[XWB2007]: XIE, X.; WILLIAMS, J. & BERGMANN, N.: Asymmetric Multi-Processor Architecture for Reconfigurable System-on-Chip and Operating System Abstractions. In:: . : Proc. Int. Conf. Field-Programmable Technology ICFPT 2007., 2007, S. 41--48

[YZ2008]: YAN, J. & ZHANG, W.: WCET analysis for multi-core processors with shared L2 instruction caches. In:: . : Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE., 2008, S. 80--89