Alma Mater Studiorum – Università di Bologna

Dottorato di Ricerca in
Informatica

Ciclo XXVI

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF/01

Operating System Contribution to Composable Timing Behaviour in High-Integrity Real-Time Systems

Presentata da: Andrea Baldovin

Coordinatore Dottorato
Prof. Maurizio Gabbrielli

Relatore
Prof. Tullio Vardanega

Esame finale anno 2014

ABSTRACT

The development of High-Integrity Real-Time Systems has a high footprint in terms of human, material and schedule costs. Factoring functional, reusable logic in the application favors incremental development and contains costs. Yet, achieving incrementality in the timing behavior is a much harder problem. Complex features at all levels of the execution stack, aimed to boost average-case performance, exhibit timing behavior highly dependent on execution history, which wrecks time composability and incrementaility with it.

Our goal here is to restitute time composability to the execution stack, working bottom up across it. We first characterize time composability without making assumptions on the system architecture or the software deployment to it. Later, we focus on the role played by the real-time operating system in our pursuit. Initially we consider single-core processors and, becoming less permissive on the admissible hardware features, we devise solutions that restore a convincing degree of time composability.

To show what can be done for real, we developed TiCOS, an ARINC-compliant kernel, and re-designed ORK+, a kernel for Ada Ravenscar runtimes. In that work, we added support for limited-preemption to ORK+, an absolute premiere in the landscape of real-word kernels. Our implementation allows resource sharing to co-exist with limited-preemptive scheduling, which extends state of the art.

We then turn our attention to multicore architectures, first considering partitioned systems, for which we achieve results close to those obtained for single-core processors. Subsequently, we shy away from the over-provision of those systems and consider less restrictive uses of homogeneous multiprocessors, where the scheduling algorithm is key to high schedulable utilization. To that end we single out RUN, a promising baseline, and extend it to SPRINT, which supports sporadic task sets, hence matches real-world industrial needs better.

To corroborate our results we present findings from real-world case studies from avionic industry.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PUBLICATIONS

- A. Baldovin, G. Nelissen, T. Vardanega and E. Tovar. "SPRINT: an Extension to RUN for Scheduling Sporadic Tasks". To be submitted to the *20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2014)*, Chongqing, China, August 2014.

  Personal contribution to: solution definition, experimental evaluation, paper writing.

- A. Baldovin, E. Mezzetti and T. Vardanega. "Limited Preemptive Scheduling of Non-independent Task Sets". In *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT 2013)*, Montréal, Canada, October 2013. `http://dx.doi.org/10.1109/EMSOFT.2013.6658596`

  Personal contribution to: solution definition, implementation of the limited-preemptive scheduler, experimental evaluation.

- F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, F. J. Cazorla. "Measurement-Based Probabilistic Timing Analysis: Lessons from an Integrated-Modular Avionics Case Study". In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, Porto, Portugal, June 2013. `http://dx.doi.org/10.1109/SIES.2013.6601497`

  Personal contribution to: provisioning of the OS kernel used in the industrial case study.

- A. Baldovin, E. Mezzetti and T. Vardanega. "Towards a Time-Composable Operating System". In *Proceedings of the 18th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe 2013)*, Berlin, Germany, June 2013. `http://dx.doi.org/10.1007/978-3-642-38601-5_10`

  Personal contribution to: re-engineering of ORK+ kernel services, experimental evaluation.

- A. Baldovin, A. Graziano, E. Mezzetti and T. Vardanega. "Kernel-level Time Composability for Avionics Applications". In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, Coimbra, Portugal, March 2013. `http://dx.doi.org/10.1145/2480362.2480651`

  Personal contribution to: TiCOS kernel services implementation.

- A. Baldovin, E. Mezzetti and T. Vardanega. "A Time-composable Operating System". In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012)*, Pisa, Italy, 2012. http://dx.doi.org/10.4230/OASIcs.WCET.2012.69

  Personal contribution to: definition of composability in the OS, TiCOS kernel services implementation, exeperimental evaluation.

LIST OF ACRONYMS

APEX Application Executive

AUTOSAR AUTomotive Open System ARchitecture

CBD Component-Based Development

CRPD Cache-Related Preemption Delay

EDF Earliest Deadline First

ET Execution Time

ETP Execution Time Profile

FCDC Flight Control Data Concentrator

FPDS Fixed-Priority Deferred Scheduling

FPP Fixed Preemption Points

FPS Fixed-Priority Scheduling

HIRTS High-Integrity Real-Time System

HyTA Hybrid Timing Analysis

ICPP Immediate Ceiling Priority Protocol

IMA Integrated Modular Avionics

MAF MAjor Frame

MBTA Measurement-Based Timing Analysis

MDE Model-Driven Engineering

MIF MInor Frame

NPR Non-Preemptive Region

PIP Priority Inheritance Protocol

PTA Probabilistic Timing Analysis

PTS Preemption Threshold Scheduling

RM Rate Monotonic

RTA Response-Time Analysis

RTOS Real-Time Operating System

RTS Real-Time System

STA Static Timing Analysis

UoC Unit of Composition

WCET Worst-Case Execution Time

1

INTRODUCTION

1.1 HIGH-INTEGRITY REAL-TIME SYSTEMS

In last few decades computing systems have dramatically changed our everyday lives: their presence has become more and more pervasive in almost every human activity and is now fundamental in a variety of complex products, from big airplanes and satellites to tiny personal devices.

As a consequence of their intensive interaction with the real world those systems are required to react to external stimuli within a certain amount of time: as such, the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical instant at which those results are produced. Software systems can be classified in different ways according to the perspective taken on them, as for example their application domain, the requirements to achieve or the realized internal behaviour. In the case of real-time systems, a major distinction is performed according to their timeliness needs: *throughput-oriented* systems focus on executing parallel jobs with the goal of maximising the overall work product, even at the price of sacrificing the performance of individual tasks. On the other side of the spectrum instead *safety-critical* systems are required to guarantee predictable runtime behaviour for all tasks, as their failure may potentially cause severe consequences on either humans lives, the environment, or existing financial assets. Example domains where safety-critical systems find their application include the biomedical, the automotive, the avionics and the space industries: the strict real-time nature of several software programs in those domains causes a growing demand for guaranteed performance, which needs to be asserted early in the development cycle, well before deployment. For this reason, safety-critical system usually incur hard integrity requirements specified at design time and validated through an accurate and rigorous certification process. Domain-specific qualification standards, as for example the DO-178B [2] for the avionics, have been defined for this purpose, to regulate the design, development and verification processes that must be adopted to deliver correct, time-predictable, and dependable systems.

The timing behaviour of high-integrity systems is assessed in industry by means of schedulability analysis: provided that the temporal behaviour of individual tasks is known, schedulability analysis is capable of deciding whether a system can meet its timing constraints with the provisioned resources. The problem therefore becomes being able to assess the timing behaviour of elementary tasks, which assembled together determine the system behaviour end-to-end. To this end Worst-Case Execution Time (WCET) analysis techniques have been devised to compute the time bounds of individual pieces of software, either by direct measurements on the target system or by simulating its execution on a faithful model, to be later passed as input to the scheduling analysis. The ability of providing accurate execution time bounds is central to determining the success or the failure of a system: on the one hand the calculated WCET values must be *safe* to ensure correctness, on the other hand they must be as *tight* as possible to avoid over provisioning and minimise factory costs. As the safety requirements are hard to comply with, the only way HIRTS providers can compete is by reducing the costly investments in the verification process, while still preserving conformance with the safety standards.

For the above reasons industry takes a conservative approach to system development: resting on simple hardware and software makes it more affordable to comply with the obligations of a demanding certification process. However, even restricting to a favourable setting is sometimes not enough, since the unpredictable behaviour arising from the direct or indirect interactions among individual components may impair timing analysis guarantees and eventually harm the safeness of the system itself. This is why the verification of a high-integrity system is still an expensive task, as a good deal of engineering effort is involved in the delivery of those systems.

Nonetheless, the increasing demand for new and more complex functionality, thus for more computational power, has recently started to push the market towards the opposite direction and calls well-established engineering practices into question to explore more ambitious architectures and meet growing customer needs. That is why even the most conservative industries have recently seen their interest growing towards multicore systems and advanced HW/SW features in general. As an example from the avionics, the size of software programs used in aircraft on-board processors has increased almost exponentially in the past (see Figure 1, courtesy of Airbus [3]), causing an equivalent increase in the demand for processing power. This trend is predicted to continue in the future owing to the use of new, complex and hard real-time functionalities for flight control, collision avoidance and auto-piloting. A similar

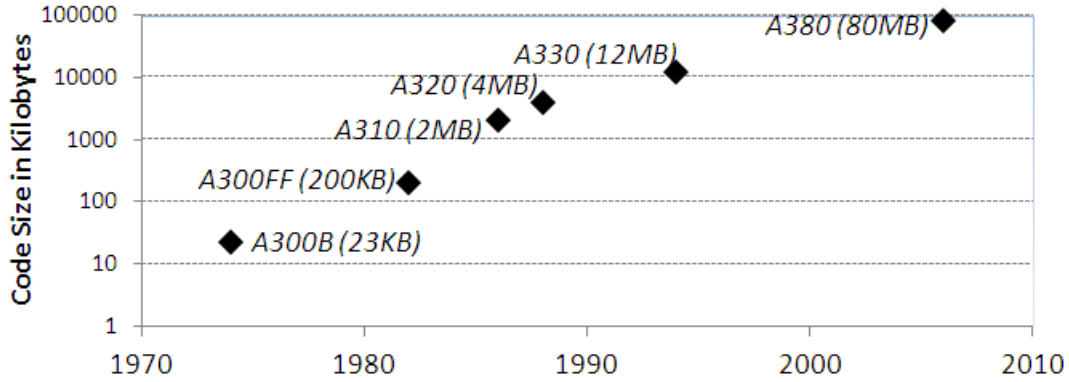


Figure 1.: Code size evolution for safety-critical systems in Airbus aircrafts.

trend has been observed in other application domains like the automotive, where the number of networked ECUs in a car has grown of 2-3 times in just a decade [4].

In a complex scenario like this, where contrasting needs shift critical decisions in very different directions, a consolidated development methodology and a trustworthy verification approach – rooted in solid timing analysis techniques – are of paramount importance more than ever before.

1.1.1 *Development Process*

Incrementality is a distinctive and unifying trait of the industrial software development process: because of the stringent requirements mentioned in the previous section, HIRTS development draws high potential benefit from taking an incremental approach to better master the complexity of delivering increasingly large and heterogeneous systems. Besides additive design and development, an incremental approach also enables step-by-step verification during system integration. This makes the production process more effective, which is a mandatory prerequisite to increase productivity and shorten time to market.

As a conceptual effort, the design process proceeds in two directions, *top-down* and *bottom-up*. The former aims at defining the elementary parts of the system starting from a set of high-level requirements: from this viewpoint it is convenient to take a divide-et-impera approach to dominate the complexity of the system to be built, following the principle of separation of concerns first advocated by Dijkstra in [5]. Decomposing a system design into smaller units to be individually developed and analysed only makes sense if the overall software product lends itself to an easy compositional aggregation. The principle of compositionality defined below takes a

top-down perspective to ensure that the system as a whole can be fully characterised as a function of its constituting components.

COMPOSITIONALITY. Properties of any composite and of the system as a whole should be derivable from properties of individual components [6][7].

The principle of compositionality is widely accepted as an industrial practice and is supported by the methodological framework provided by Model Driven Engineering (MDE) [8]: in order to address the complexity of system development, MDE proposes to combine domain-specific modeling languages and transformation engines to define different views on the system, whose abstraction level is progressively lowered until sufficient detail is provided to enable actual implementation. In this process the principle of separation of concerns helps attain separation between the functional and non-functional parts of a system, thus permitting to reuse the former within different designs independently of the specific implementations of the latter. The software entities involved in this approach are *components*, *containers* and *connectors* [9][10]: according to [11] components are the elementary units of composition of software, which conform to a *component model* prescribing the properties they must satisfy and the methods for composing them. Within components pure functional logic is encapsulated, making them suitable for reuse, whereas non-functional properties for each component are wrapped into containers. A *computational model* and a *programming model* are defined to match components and containers to a set of analysis techniques and programming languages, respectively, to convey analysis and implementation semantics. Finally *connectors* are dedicated to decoupling inter-component communication by mediating between containers. Containers and connectors have strong dependencies on the *execution platform* and are therefore the only part to refactor when the execution environment changes.

The viability of such a divide-and- conquer approach silently builds on the principle of composability, which takes the *bottom-up* perspective on the system and guarantees that incrementally aggregating components and their functionality is in fact possible:

COMPOSABILITY. Properties defined for the elementary components of the system must hold independently of the execution context and be preserved after composition in the deployed system [6][7].

The principle of composability is easily explained in the functional dimension. For example, if the functional behaviour of a module a_i has been proven correct in isolation, then it is so even in case a becomes part of a system A that includes other functions, say a_j, a_k , etc. For this reason, the system components can be considered in relative isolation when assessing their functional correctness, as the latter is naturally preserved across software increments. Similarly, in the timing dimension, it would be desirable for a software module to exhibit the same timing behaviour independently of the presence and operation of any other component (present or future) in the system. Cost and time efficiency concerns require that components, which are developed and verified in isolation, should exhibit the same exact behaviour regardless of how they are eventually aggregated.

The benefit that can be drawn from compositionality and composability is twofold: on the one hand functional and non-functional requirements that have been decomposed and assigned to elementary components in the design phase can be later composed bottom-up to incrementally verify larger portions of the system. On the other hand, whenever changes to already existing parts of a system are needed, the misbehaving parts can be modified and verified again in isolation. Unfortunately, industrial penetration of these principles still encounters some difficulties [7], mainly due to the lack of composability in the time dimension, as we discuss in this thesis.

1.1.2 *Timing Analysis*

As temporal correctness is one fundamental goal of real-time systems, techniques need to be developed to provide safe and affordable timing analysis of delivered systems. More than meeting safety requirements, timing analysis is of paramount interest to industry to the goal of correctly sizing system requirements, to avoid over provisioning of computation resources and rationalise costs; obviously, this can be achieved only if the costs of verification are kept low as well. In hard real-time systems therefore composability is expected to go beyond the functional dimension, to enable separate timing analysis on smaller, functionally cohesive building blocks that could be tasks or groups thereof. Time composability should guarantee that the timing behaviour of any such building blocks is not inordinately affected by the presence and activity of any other task in that system. The commonly adopted method to verify the timing behaviour of a system is schedulability analysis [12]: given the active entities (tasks) defined in a system and some temporal constraints over them to characterise their behaviour (such as their activation period, deadline and execution

time), schedulability analysis checks whether such a defined system is feasible; if so, the scheduler associated with the analysis is capable of calculating a valid schedule for the systems, that is an assignment of computing resources to tasks over time, satisfying the constraints imposed to the system.

The state-of-the-art approach to schedulability analysis is Response Time Analysis (RTA) [13], which calculates worst-case response times for the individual tasks in a system, that is the longest time it takes for a job of a particular task to complete its execution since its release, accounting for worst-case interference from the other tasks. Such a value clearly depends on the execution time of a task, i.e. the amount of processor time a task requires to complete its jobs. Additionally, the analysis assumes that disturbance can be safely bounded: for example in systems where preemption is enabled and according to the scheduling policy in use, response times might extend to account for blocking due to contention for shared resources. For this reason the soundness itself of the analysis results critically depends on the actual degree of time composability in the system. While some of the parameters required by RTA are fixed and statically decided at design time, others are strictly dependent on the dynamic behaviour of the system at run time and cannot be determined beforehand. The most critical of them is the execution time of a task, which naturally depends on a variety of factors and their interactions, which may be too expensive or even impossible to determine before the system is running. Execution time can therefore vary across different execution scenarios between a minimum and a maximum value, which are called Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET), respectively (see Figure 2). Timing analysis focuses on the latter, as HIRTS must necessarily be sized on the worst-case scenario to contend with hard certification arguments and provide the desired safety guarantees. Looking at Figure 2, it is clear that the WCET of a task is a representative value which upper-bounds *all* actual execution times of that task in every possible execution once the system is deployed. This is a strong yet reasonable requirement, since WCET values should be calculated once at validation time and then be always valid while the system is in the production phase, in order for the timing analysis process to be affordable.

As already noticed, the upper bound represented by the WCET value should present two important properties, *safeness* and *tightness*. Unfortunately, it has been observed that the problem of determining the execution time of a unit of software is equivalent to deciding its termination [14]; this daunting challenge suggests abandoning the hope of determining exact execution times by analysis in reasonable time and complexity, and has shifted attention to measurement-based and approximated methods.

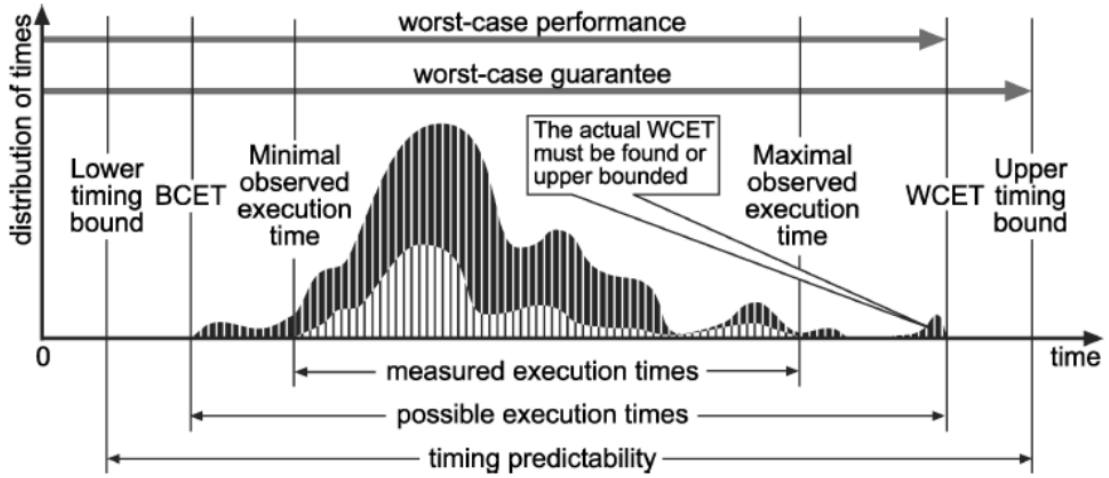


Figure 2.: Execution time distributions [1].

Current state-of-the-art timing analysis techniques can be broadly classified into three strands: static timing analysis (STA), measurement based (MBTA) and hybrid approaches (HyTA) using combinations of both[1]. More recently Probabilistic Timing Analysis (PTA) techniques have been proposed to tackle the difficulties encountered by traditional approaches [15][16].

STATIC TIMING ANALYSIS. Static timing analysis techniques and tools (for example [17]) do not execute the code directly, instead they rely on the construction of a formal model of the system and a mathematical representation of the application: such representation is processed with linear programming techniques to determine an upper-bound on the WCET of that specific application on the model. Accordingly, the analysis process is split into two phases, namely *program analysis* and *microarchitecture modeling* [18]. The former is the problem of determining what sequence of instructions will be executed in the worst-case scenario: in this step existing paths and branches in the code are identified and additional information like bounds on loop iterations and ranges on data variables is collected. Since the number of program paths is typically exponential with the program size, it is important to remove the largest possible number of infeasible paths from the solution search space; this can be done through program path and data flow analysis, but human intervention is still required as this process cannot be fully automated. However, analysis accuracy can be

traded for simplicity in this phase by deciding at what level code should be evaluated: considering the source code makes analysis easier but its results may be invalidated at the machine level because of the compilation process standing in between.

The second challenge of a static analysis process is instead the problem of modeling the target hardware system and computing the WCET of a given sequence of instructions executing on it; the model abstracts significant properties of the processor, the memory hierarchy, the buses and the peripherals in use and must be conservative with respect to the timing behaviour of the real hardware. Obviously, the more complex such a hardware architecture is the more difficult will be modeling it safely: in fact some of the hardware accelerator features introduced in latest years to speed up execution on the average case make the execution time of a single run highly unpredictable. In those cases determining a safe and tight timing bound may be very difficult or even highly impractical, since execution time becomes highly dependent on execution history and on local hardware states. Finally, WCET bounds are computed starting from blocks of smaller granularity and then combining results together according to the structure of the program's control flow graph produced in the first step. Unfortunately, subtle phenomena like timing anomalies [19] may arise and risk to harm the safeness of the analysis process: timing anomalies manifest when considerations made to model the local worst-case do not apply to the global worst-case and therefore introduce some underestimation in the analysis; such pathological cases originate from the non-determinism introduced by dynamic hardware features, such as speculation and out-of-order execution in pipelines. In fact, non-locality of instruction timing prevents the analysis from knowing exact machine states and might break the approach taken by static analysis to reconstruct global WCET bottom-up [20]. In this landscape, one of the most challenging issues is understanding how to compose WCET bounds calculated for smaller parts of a system into an end-to-end WCET estimate. To further complicate analysis, systems where a *real-time operating system* (RTOS) layer is present usually experience additional interference from the operating system itself: system calls nested into an application's control flow may take unpredictable time to execute and pollute the machine state in a way that degrades timing performance and makes it much more difficult to determine.

The strength of static WCET analysis methods is the safeness of the results they produce, as overestimations are made in all required steps; at the same time this often reveals as a drawback, since upper bounds computed by static analysis are not tight enough to be used in real industrial applications as they push towards exceedingly pessimistic system oversizing. Moreover, this type of analysis is usually

too expensive to carry out owing to the need to acquire exhaustive knowledge of all factors, both hardware and software, that determine the execution history of the program under analysis. Some processor architectures may dramatically increase this cost. Others, possibly subject to intellectual property restrictions or incomplete documentation, may even make it altogether impossible. In such scenarios static analysis is usually discouraged in industrial practice, at least for the qualification of a system as a whole, rather confining its adoption to those specific sub-components whose size and complexity is manageable to fit this purpose or alternatively resorting to observations.

MEASUREMENT-BASED TIMING ANALYSIS. Another approach to WCET analysis is the measurement-based one: the simple idea behind it is to execute the code of interest directly on the target hardware or a cycle-accurate simulator for a certain number of times and possibly with different sets of inputs to measure real, end-to-end execution times [21]. Such timing values are usually referred to as WCET estimates rather than bounds, as it cannot be guaranteed that actual WCET is present among them; this is again because of the same problems faced by the different steps of the static analysis process. First, the observed execution times are subject to data dependencies influencing the choice of paths taken in the control flow: for this reason one can never know whether the worst-case path is included in the observed measurements. Additionally, as WCET depends on the initial hardware state, execution times also depend on the initial conditions of both hardware and software which are difficult to define, control and reproduce with sufficient detail, especially for complex hardware architectures. For this reason one common technique in industry is to measure high watermarks and add an engineering margin to make safety allowances for the unknown. However, the size of a suitable engineering margin is extremely difficult – if at all possible – to determine, especially when the system may exhibit discontinuous changes in timing due to pathological cache access patterns or other unanticipated timing behaviour. There is finally a further problem with measurements, which is related to the overhead introduced by the method itself: if extra instrumentation code is introduced to collect timestamps or cycle counters, produced results will be affected accordingly; fully transparent, non-intrusive measurement mechanisms are possible only using external hardware, which prevents the process from being fully automated.

HYBRID TIMING ANALYSIS. Hybrid timing analysis approaches [22] try to combine the virtues from both static and measurement-based techniques; timing for smaller program parts is obtained via measurements on real hardware, while composition of values to deduce the final WCET estimate is done statically. Determining the appropriate granularity for the elementary program blocks to be measured is an open problem, while to combine observed execution time profiles or even to predict unobserved events it has been proposed to apply probability theory [23][15]. Hybrid techniques avoid the need for constructing an expensive model of the hardware and allow the user to achieve a higher confidence in a WCET than simply measuring it end-to-end. However such techniques require a good method of measuring on-target timing information and still rely on having an adequate level of testing. Hybrid measurement-based approaches have been successfully implemented in academic [24][25] and commercial tools [26] for WCET analysis.

PROBABILISTIC TIMING ANALYSIS. Probabilistic timing analysis (PTA) [27][28][29] has recently emerged as an attractive alternative to traditional timing analysis techniques. PTA considers timing bounds in the same manner as the embedded safety-critical systems domain addresses system reliability, which is expressed as a compound function of the probabilities of hardware failures and software faults. PTA extends this probabilistic notion to timing correctness by seeking WCET bounds for arbitrarily low probabilities of occurrence, so that even if violation events may in principle occur, they would with a probability well below what is specified by a system safety requirements. The EU FP7 PROARTIS project [30][16] we were involved in is a recent initiative in this field aimed at defining a novel, probabilistic framework for timing analysis of critical real-time embedded systems.

1.2 PROBLEM DESCRIPTION

Ideally, by the adoption of a compositional approach to system design and development, incrementality should naturally emerge as a consequence of the composable behaviour of the elementary constituents (i.e., software modules) of a system. However, in practice, real-time software development can only strive to adopt such discipline, as the supporting methodology and technology are still immature. At the same time, the analysis of the timing behaviour of real-time systems, whether local to application procedures or global at system level, silently builds on the assumption that the timing bounds computed from or fed to it add compositionally. This is not true in general as

well and guaranteeing just timing compositionality on current hardware and software architectures is difficult. Although timing analysis frameworks typically characterise the timing behaviour of a system compositionally, a truly composable timing behaviour is not generally provided at the lower levels of a system. In particular, two main factors can be accounted for breaking the stability of execution times, which is a fundamental enabler of time composability [6]:

- *Sensitivity to the execution context:* the machine state influences the execution time of software running on top of it.
- *Data-dependent decisions in the control flow:* different input data activate different control flow paths in a piece of software, which normally take different execution times to complete.

Whereas the latter can be mitigated by appropriate design choices and coding styles (which are discussed in Chapter 2), sensitivity to the execution context can only be tackled by minimising the dependence of the timing behaviour from the execution history. The timing behaviour of numerous hardware and software components seen in isolation can be abstracted into simple deterministic worst-case models. This quality is much desired by state-of-the-art static timing analysis techniques, which can thus be applied with good cost-benefit ratio. The situation changes when those individual components are assembled into larger units. As long as the number of the involved components and of the interactions among them remain small, the resulting system can be analysed by STA with decent (i.e. not overly pessimistic) results. However, when the complexity increases, as it is likely the case for future systems, modelling system behaviour as it results from the interaction of multiple independent components becomes exceedingly costly and STA hits the wall [31][32]. In this case composability in the timing dimension is wrecked by history-sensitive execution-time jitter as the execution history becomes a property of the whole system and not that of a single component. Time composability may in fact be broken at any place of interaction between execution components at any level of granularity in the hardware-to-software axis of composition (Figure 3). The main obstacle to time composability is that modern hardware architectures include a score of advanced acceleration features (e.g., caches, complex pipelines, etc.) that bring an increase in performance at the cost of a highly variable timing behaviour. Since those hardware features typically exploit execution history to speed up average performance, the execution time of a software module is likely to (highly) depend on the state retained by history-dependent hardware, which in turn is affected by other modules [33],

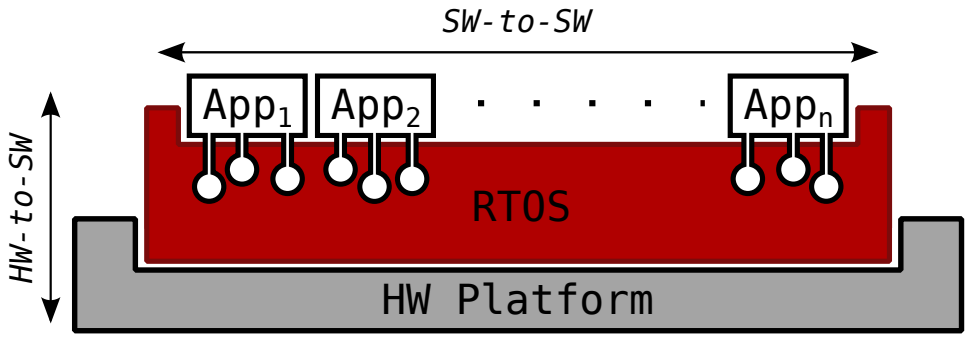


Figure 3.: Axis of composition in a reference layered architecture.

and also propagate up to the level of schedulability analysis via the occurrence of preemption. Any loss of accuracy in, or plain lack of, knowledge on the state of inner components with bearing on the timing behaviour of interest yields substantial pessimism: this reduces the guaranteed performance of the system. Unless we want to resort to a simplified (and less performing) hardware architecture, as proposed in [34], industry is presented with a lose-lose choice: either increase the number of hardware units to compensate for the lesser guaranteed performance per unit, which breaks economy of scale, or endure the possibly prohibitive costs of seeking the missing information needed to mitigate pessimism in computing bounds. In the latter case timing interferences must be taken into account in more complex forms of schedulability analysis: for example the interference of caches on task interleaving is considered in [35], whereas [36] considers the problem in a multicore setting. Yet the cumulative interference arising from all history-sensitive hardware cannot always be efficiently accounted for by state-of-the-art timing analysis techniques, which incur either steep pessimism or vast inaccuracy.

A less studied threat places along the software-to-software axis of composition, between the real-time operating system (RTOS) and the application. In [37] the author argues that, unless corrective measures are taken, the inter-dependence between the RTOS and the application is so tight that their respective timing analysis cannot be performed in reciprocal isolation. In other words, RTOS and application are not time-composable in the general case. One may wonder about the effects of this argued lack of time composability: after all we haven't seen systems reportedly fail because of that problem. In fact, the most direct consequence of applying timing analysis to a non-time-composable system is that the results are simply incorrect and we cannot easily tell whether they err toward excessive pessimism or on the opposite side. One of the reasons why we don't see serious time violations happen in non-time-

composable systems is that industrial practice injects arbitrarily large slack quantities in the system schedule to compensate for the possible occurrence of local overruns of modest amplitude. While empirically successful, this industrial practice bases on two fundamental unknowns (the ”right” amplitude of the slack and the ”largest” possible amplitude of the prediction errors) and therefore has little to save itself when the conditions change.

Tools have been developed both in industry and academia that leverage the analysis methods mentioned in Section 1.1.2 [38]; however, they all face some usability problems which prevent their actual use in real-world applications [39]. Measurement-based methods suffer from the lack of observations due to insufficient code coverage and unexercised paths, which could make analysis miss actual WCET. On the other hand, static methods are concerned with the abstract representation of the target hardware platforms, since improving model characterisation would lead to better results and tighter bounds. Understanding the program structure is therefore a cross-cutting issue, which has been attacked also with the help of dedicated programming languages and compilers. Chapter 2 shows how past and current research tries to deal with such issues.

1.3 objectives and contribution

HIRTS industry needs to take an incremental approach to software development more than ever, in order to rationalise costs and shorten time to market. A number of methodologies and frameworks currently exist to support the design and implementation of reusable and portable software systems: while many of them perfectly support the delivery of functional requirements, little progress has been made to address non-functional needs, like timeliness. Current component-based techniques are therefore doomed to failure if better control of spatial and temporal composability in-the-small are not provided. We were given the opportunity to study the issues above in the context of the EU FP7 PROARTIS project [30][16]: this initiative aimed at defining a novel, probabilistic framework for timing analysis of critical real-time embedded systems. As an industrial trait, the main project focus was set on partitioned applications, commonly encountered in avionics systems, and particularly on the IMA architecture and its ARINC 653 [40] incarnation. These industrial standards encourage the development of partitioned applications, where the concept of composability is strictly related to the fundamental requirement of guaranteeing spatial and temporal segregation of applications sharing the same computational resources within a

federated architecture. At the same time we kept an eye open on the space industry, which shares similar safety-critical and timing verification concerns with the avionics domain, though imposing different requirements.

Our research was focused on attaining time composability taking a *bottom-up* perspective on the system. As a preliminary study we identified those factors across the system architecture limiting the achievement of time composability in real-world systems: this was achieved by understanding the dependencies of execution time on execution history that are introduced along the execution stack. The analysis was carried out through the traditional structure of a layered architecture as shown in chapter 2. Breaking down the execution platform into three classic layers – the hardware layer, an operating system layer (kernel) in the middle, and, finally, a user application layer running on top – makes it possible to address history independence and thus timing composability as a bottom-up property that must be first accomplished by the underlying hardware, then preserved across the OS primitives and services, and finally exhibited by the user application. In order to give strong foundations to our approach we first drew some fundamental assumptions on the hardware platform to be considered: we identified the obstacles to timing analysis as those features specifically intended to speed up the average-case performance, but that present highly jittery behaviour. To control those effects, the execution environment must be accurately characterised from a hardware perspective, either by enforcing full determinism or with the help of probabilistic techniques, as explained in Section 2.5.2.

The core of our work aims at these two goals:

1. Propose countermeasures to remove the sources of dependency which hinder time composability and software analysability at the operating system level. We argue that the OS is the best place to inject time composability in a system due to its privileged position in between unpredictable hardware and opportunistic application code. Investigation focused on those resources, algorithms and data structures operating within the kernel which present variable service and access times. In fact, jittery OS behaviour influences application timing as much as hardware, even though this aspect is often neglected in literature: making OS primitives composable in the time domain is therefore a strong prerequisite to time analysability. To this end the execution of the OS should not degrade the machine state in a way that could interfere with the application execution, and fixed-latency kernel services should be implemented, whose invocation overhead is known.

2. Evaluate and implement solutions to counter the lack of time composability on multicore architectures. Even though it is widely recognised that the architecture of multicore processors is a completely different and more challenging setting than the single core, it is a matter of fact that current trends push towards the adoption of multicore architectures even in safety-critical systems. The very universal principle of time composability must be therefore reconsidered to study its achievement – if at all possible – in a variety of multiprocessor platforms.

Ideally, time-composable hardware and kernel layers should be able to remove all history-dependent timing variability so that state-of-the-art timing analysis approaches could be able to account for the residual variability at the application level. The latter is therefore the only place at which (application-logic related) timing variability should be allowed, and is therefore responsible for preserving system analysability and predictability by proper design and implementation choices. Experimental evidence in support of proposed solutions was provided by means of state-of-the-art timing analysis techniques and tools: hybrid timing analysis was privileged because of its wide adoption in the industry which makes it as a trustworthy approach to timing analysis. Additional support in this sense was provided by the PTA methodologies and tools developed in the PROARTIS project, which in turn make our investigation a fundamental contribution in the process of defining the applicability and requirements of the emerging probabilistic timing analysis frameworks.

1.3.1 *Contributions*

This thesis sets in the ongoing investigation on high-integrity real-time systems carried on at the University of Padova; for this reason past and current work from our group is referenced and discussed in this thesis. In this scope we attacked the problem of providing time composability to the process of timing verification of HIRTS taking on an industry-oriented perspective, thanks to our involvement in the PROARTIS project. Assuming educated behaviour of the HW layer and exploiting existing research in HW timing analysis, we focused on the OS layer and reasoned about its design with the purpose of enabling time-composable system analyisis. Alternatively, where no guarantee on the HW platform could be leveraged we investigated how the OS can attenuate its disrupting behaviour.

Figure 4 shows the reciprocal interactions among the different actors which come into play when reasoning about time composability from a system-level perspective:

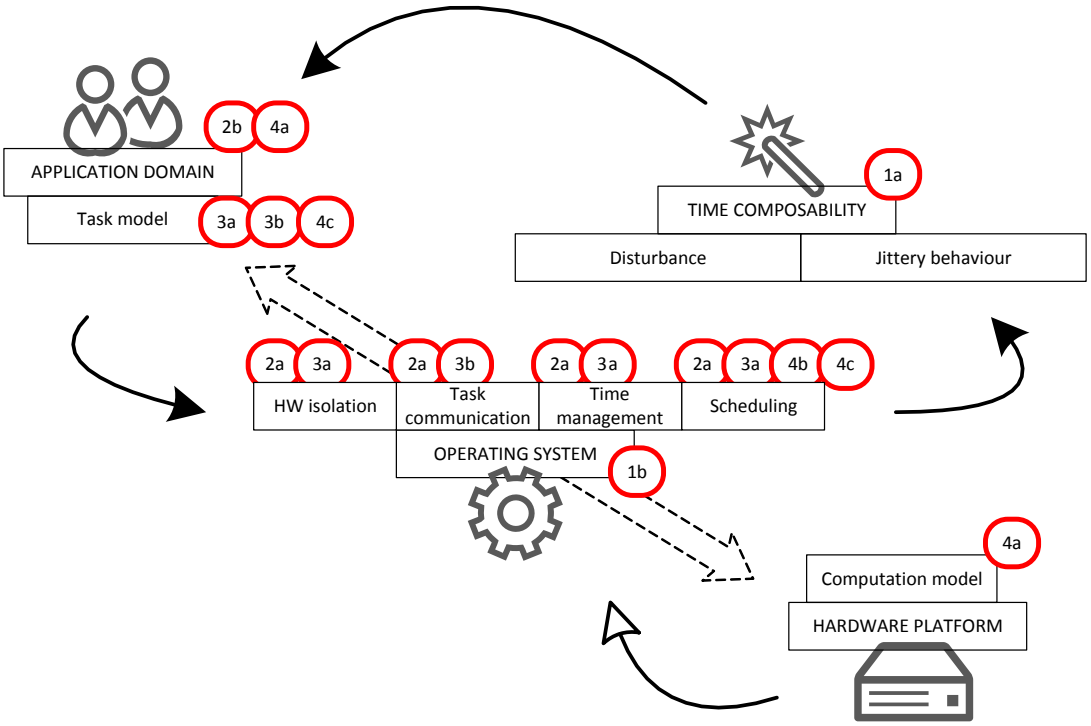


Figure 4.: Contributions of this thesis.

the layers of the system architecture we consider are arranged in diagonal, along the dashed double-arrow from the upper-left to the lower-right corner. The OS is placed in the centersince it is the key point of our discussion, together with the concept of time composability which is also included in the picture. Straight arrows highlight how decisions taken on one layer or concept influence other blocks. Solid arrows identify the main conceptual loop around which the work of this thesis revolves: according to the domain-specific application needs, which influence the decisions on the the task model to be adopted, choices have to be taken in the OS layer to support those requirements; these choices in turn impact the achievable degree of composability in the system, which eventually reflects on the behaviour of the initial application. The hardware layer, although strongly present in the problem because of its influence on the software behaviour, is not directly addressed by our dissertation, and is therefore brought into scene by an empty arrow. Figure 4 also helps spot where the main contributions of our work place themselves, as suggested by the circled numbers corresponding to the explanation below:

1. Definition of of time composability and its istantiation within the OS layer.
    a) We give a new characterisation of **time composability** in terms of disturbance and jitter, which is general enough to adapt to a broad variety

of existing real-time systems, with no assumption on the specific system architecture or deployment.

b) Based on this definition we identify the main design choices harming time composability, as reflected in common features of existing real-time operating systems.

2. Study of time composability within an ARINC 653 task model.

a) We study the **ARINC 653** software specification for space and time partitioning in safety-critical real-time operating systems for the avionics: thanks to the characterisation given at point 1 we show how to modify an existing RTOS kernel designed without the principles of time composability in mind to make it better suitable to the timing analysis verification process.

b) The results of the study are evaluated on a real-world sample avionics application used as benchmark by the avionics industry.

3. Study of time composability within a sporadic task model.

a) We implement and evaluate a **limited-preemptive scheduler** [41][42] in a Ravenscar Ada RTOS kernel; since the scheduling activities cause great interference to executing user-space applications therefore impacting its timing behaviour, we will show how this phenomenon can be mitigated to favour timing analysis.

b) Additionally, we propose a way to deal with the problem of resource sharing into the limited-preemptive scheduling framework, which has never been considered before.

4. Study of time composability in multicore architectures.

a) Time composability is much more difficult to achieve on a multiprocessor due to the run time interference naturally emerging from the parallel execution of multiple control flows. We first study different computational models and apply the composability principles in a partitioned scenario to enable probabilistic timing analyisis in an industrial case study.

b) Then, assuming that the disrupting effects of hardware interference can be mitigated probabilistically, we focus on multicore scheduling as the most influencing problem affecting compositional timing analyisis. Current schedulers take either a partitioned or a global approach to the problem: the former statically determines an assignment of tasks to processor and,

although performing quite well, cannot adapt smoothly to task set changes, nor avoid over provisioning of system resources; the latter allows full migration of tasks to processors, the performance penalty thereof being too high to be accepted in a real-world scenario. We evaluate the **RUN** algorithm [43], which is a promising solution to the optimal multiprocessor scheduling problem: RUN tries to mitigate the drawbacks of both partitioned and global approaches, taking an original approach to the problem by virtual scheduling. While not pinning tasks to cores, RUN packs them in aggregates than can be locally scheduled to significantly reduce migration while achieving optimal schedulable utilisation. RUN is especially interesting to our work here because it does not prescribe the local scheduling to be used (which must obviously be optimal if full schedulable utilisation on all cores is to be attained), and therefore allows us to reuse the solutions we devised in this work to achieve time-composable behaviour for single-core operating system.

c) We propose **SPRINT**, an extension to RUN to handle sporadic tasks, which are not contemplated in its original formulation but must be accommodated to schedule real-world applications.

1.4 THESIS ORGANISATION

This thesis is structured as follows: Chapter 2 reviews state-of-the-art methods for timing analysis of safety-critical real-time systems, with special attention given to composable WCET calculation and OS analysis; we will also describe our reference HW platform and detail the assumptions we require on it.

Chapter 3 will introduce our notion of time composability, the ARINC 653 API specification adopted by the avionics industry and the work we performed on an ARINC-compliant kernel with the goal of injecting time composability in it. We will then move to a space setting to illustrate how it was possible to port those results into a different OS kernel, the Open Ravenscar Kernel, and to extend them to implement and take benefit from a limited-preemptive scheduling approach.

The multicore setting and the RUN algorithm are covered in Chapter 4, together with the extension we propose to adapt the algorithm to handle sporadic task sets.

Finally in Chapter 5 we draw some final considerations and give some hints on the future lines of research which may be conducted on this topic.

2

BACKGROUND AND ASSUMPTIONS

In chapter 1 we discussed how time composability may be broken at multiple places in a system, from low-level hardware up to the application level. We now proceed to examine how recent research has dealt with the problem of analysing the timing behaviour of a system within the boundaries of system layers. Although considering each abstraction layer in reciprocal isolation is probably the easiest approach for the better control gained on the analysis, the composition and interaction of elementary components across layers cannot be disregarded: cross-layer interactions among heterogeneous components are in fact the place in the architecture where discontinuity in methods and technologies may occur, putting time composability in danger and potentially impairing the bounds calculated by a naïve analysis.

We now try to identify the most critical areas at hardware, OS and application levels by summarising the most significant existing research in the timing analysis landscape. In parallel we introduce the assumptions made in this thesis for each of those topics.

2.1 HARDWARE ANALYSIS

A significant amount of research on WCET calculation has been devoted so far to precisely characterising hardware behaviour. Since bottlenecks are very likely to occur in the presence of shared hardware resources, processors and memory hierarchies have been the natural candidates for optimisation: indeed, modern architectures have experienced the proliferation of new hardware features intended to boost average-case performance at the price of local non-determinism and highly variable behaviour across different executions. Additionally, these features interact with one other to further complicate execution time calculation, even when single runs are considered in isolation. Hardware manufacturers do not help in this context, as specifications tend to be incomplete or even incorrect, both because of the unquestionable hardware complexity and of the disclosure restrictions arising from intellectual property concerns. Forcing hardware to exhibit a predictable behaviour is mandatory to provide some minimal guarantees needed at the higher layers of the architecture and by the analysis process: on one hand full determinism would make architecture modeling

easier and improve the accuracy of static WCET analysis methods. However, this is not the direction taken by hardware manufacturers, whose convenience is rather bringing new features into COTS products for commercial reasons; at the same time the HIRTS industry may not always have sufficient influence to change this trend. On the other hand time randomisation would help achieve the hypotheses of independence and identical distribution of observed timing profiles required by probabilistic methods [44]. Unfortunately, similarly to the idea of deterministic hardware, this would require vendors to provide specialised hardware for specific system manufacturing. Research at the hardware level has focused on considering available COTS products and trying to characterise the state of such platforms along the execution: to this end, high-variability resources are the best candidate for improvement. An example of this formal approach is given by Kirner and Puschner [31], who identify the Timing-Relevant Processor State (TRPS) as the set of those processor components, whose values can influence the execution time of some given instructions; these components are then studied and some composition rules are given to consider their latencies, taking also into account timing anomalies.

PROCESSOR. Processor instructions may experience variable latency on output depending on the type of the involved operands: this is the case for example of the division and floating point operations. Moreover, when an interrupt is raised, the hardware may suffer variable interference to serve it. The problem has been attacked in [45] by trying to provide an architecture (SPEAR) with a constant-time, easy-to-model instruction set for a full-fledged 16-bit processor equipped with a 3-stage pipeline and caches; its design is carried on with predictability rather than high efficiency in mind. The authors interestingly outline the main sources of jitter in a processor, namely the synchronisation period between the occurrence of an interrupt and the time at which it starts to be served, the context switch overhead and the execution time of the instructions themselves. They argue that the first two operations can be performed in constant time, except for the obvious variability possibly introduced by interrupt recognition due to the clock granularity; also instructions in SPEAR have deterministic execution times, as they are single-word and out-of-order execution is not permitted within the pipeline. Unfortunately, no real hardware implementation exist at present that could guarantee such properties.

CACHES. Memory hierarchies are most exposed to variability, since their latency is significantly higher than that of processors: for this reasons fast caches have been

introduced in recent years to speed up computation on the average case, at the cost of lower predictably. Studies in this area have tried to directly account for Cache-Related Preemption Delay (CRPD) into schedulability analysis [46][47], or to restrain cache behaviour by locking [48][49] or partitioning [50]. Alternatively, caches could be replaced by fast scratchpad memories, whose behaviour can be entirely programmed by software. Either way, instruction and data cache behaviour is usually considered separately to keep the cost of the analysis affordable. More recent work attacks the problem of accounting for cache unpredictable overhead by introducing time randomisation [51][52][53], which makes them analysable by measurement-based, PTA techniques [16]. Other authors [54] propose a two-stage compositional analysis method for instruction caches: the preliminary module-level step aims at parametrizing the analysis process by collecting application-specific data flow information, which is further used to reason about potential cache evictions due to calls containing conflicting references. Subsequently, the compositional stage uses previous information and actual memory addresses incrementally to refine obtained results by composing modules bottom-up in the control flow graph. Rather than performing the analysis a posteriori, Mezzetti and Vardanega [55] advocate instead the benefits of considering different flavours of cache interference early in HIRTS development process: a cache-aware architecture is proposed, in which system-level countermeasures are taken at design time to limit undesired cache effects, by considering the code size, its concurrency layout and reuse profile.

PIPELINES AND BRANCH PREDICTION. Out-of-order, speculative execution significantly complicates WCET analysis because many potential execution states are generated which could at any time be discarded as a consequence of wrong predictions. Dynamic branch prediction intrinsically depends on execution history to collect data flow information useful to speculate on the outcome of any branch instruction [56]. Pipelines as well represent a serious obstacle for the analysis process, even when considering elementary portions of code (basic blocks), because of the variable latencies characterising different processor instructions and the presence of timing anomalies. This consequently makes it very hard to capture interferences occurring among non-contiguous basic blocks in the control flow graph. Li et al. [57] start by identifying the different forms of dependency among pipeline stages; then, in the analysis they take into account the possible initial pipeline states and the instructions surrounding the basic block under investigation. The study also integrates pipeline and instruction cache analysis, however experimental results cannot be considered

satisfactory since the problem is known to be a hard one and overestimation is needed therefore to make the solution space tractable. Betts et al. [58] adopt instead a hybrid measurement-based approach and study pipeline behaviour in relation with code coverage; the authors propose to identify pipeline hazard paths leading to potential pipeline stalls. Considering longer paths may be advantageous in theory to capture distant dependencies in the execution flow, however the number of those paths is likely to grow exponentially and no evidence is provided in the study of the validity of this method.

VIRTUAL MEMORY MANAGEMENT. Virtual memory is frequently used in HIRTS as a means to provide spatial isolation among applications sharing the same physical memory space, usually for security reasons. At each memory access a translation from a virtual to a physical memory address is performed, typically with the help of one or more intermediate data structures called Translation Lookaside Buffers (TLBs): these buffers are in charge of hashing a portion of the address space in use, in order to speed up performance similarly to what caches do. Time variability incurred by the use of virtual memory is twofold: first, one may or might not pay a penalty according to whether the required physical address is present in a TLB or not (page fault), and in the latter case the page table must be searched through. Very few studies have focused on virtual memory management and usually with the goal of obtaining tighter rather than exact WCET bounds. In [59] for example the dynamic paging behaviour is statically decided at compile time by selecting paging points, provided that some hardware or system support is available for this technique; additionally, some effort is required to identify which pages will be actually referenced dynamically.

Existing literature on hardware predictability attacks the issues affecting one hardware feature at a time: very few authors try to integrate timing analysis for two or more components and even less cope with the hardware platform in its entirety. This is because extensive analysis of complex interleavings and interactions among separate components is discouraging and probably unaffordable: wishing for deterministic hardware specifications and behaviour might be a dream as well. Nonetheless precise architecture modeling is a mandatory prerequisite to obtaining tight WCET bounds by means of static analysis. Probabilistic methods might be more effective to capture the complexity of such phenomena, but they impose unusual requirements on the platform to be satisfied for their applicability, such as the independence of instruction time profiles. Enforcing independence and composability at the finest possible grain,

i.e. at the instruction level, may look as an attractive opportunity; however, this is a very hard objective to achieve in practice due to the very heterogeneous components packed into a hardware platform, each distinguished by different execution latencies and data dependencies, which are hardly identifiable before actual execution.

2.2 OS ANALYSIS

Hard real-time systems have traditionally been deployed as embedded applications, that is without the need of general-purpose operating systems to mediate between the application and the hardware layer. However, recent industrial trends are shifting the development of complex HIRTS from a federated architecture to an integrated environment where logically segregated applications share the same physical resources, mainly for economical reasons: good examples of this trend are the Integrated Modular Avionics (IMA) with its ARINC 653 API incarnation [40], and the AUTomotive Open System ARchitecture (AUTOSAR) [60][61], which are recognised as de-facto standards in the avionics and automotive industry, respectively. For the reason above research on the role assumed by the operating system layer on WCET analysis is quite immature and much work has still to be done.

Early works on the timing analysis of real-time operating systems started by applying similar techniques to those used for WCET analysis of application code, in consideration of the software nature of the Real-Time Operating System (RTOS). Colin and Puaut [62] encountered non-trivial issues in the analysis of the RTEMS kernel with static techniques, achieving very poor results with a reported overestimation of the 86% of the real WCET. The reason for such pessimism is manifold, and its root causes have been identified in two main works [37][63]: we summarise encountered problems and proposed countermeasures in this section, according to their root cause. Most of the problems arise from the use it is made of the RTOS during execution, whose interleaved execution with the applications perturbs the state of shared HW/SW resources, preventing static analysis techniques from precisely characterising the state of the system before execution; while programmers can be asked for this additional information through annotations in the case of user applications, this opportunity is precluded at the OS level, whose code is usually considered as a black box provided by third parties and therefore out of direct control of the end user.

STATIC ANALYSIS. The traditional approach of static timing analysis has been applied to OS analysis in a variety of works. In [64] all decisions about the interactions

with the OS are taken offline: scheduling points are statically chosen and interrupts are guaranteed to be served at time instants in which no other system or task activity is in progress. This approach works in conjunction with the single-path programming approach that will be illustrated later; however, this is quite a rigid requirement for systems where performance and responsiveness to the environment are important. More recently a comprehensive analysis of the seL4 microkernel was given in [65]: authors build an integer linear programming problem with constraints derived from worst-case paths and target hardware properties; then, they compare results on interrupt latencies and OS primitives obtained with both static analysis on the constructed model and measurements on real hardware, concluding with very good ratio between computed and observed results. Unfortunately, such experiments look at the kernel services in isolation and do not consider context dependencies naturally arising in a real-world execution. Earlier, the authors of [66] tried to characterise the time profile of μC/OS-II kernel primitives by means of static analysis, which showed the difficulty of safely and precisely characterising the OS behaviour, especially in the presence of preemption. The considerations above may suggest that RTOS analysis cannot be performed in isolation as it is done sometimes for embedded application code, but rather application and OS should be analysed in conjunction, as noted also in [37]. As a solution, the same author proposes a framework [67] in which schedulability and WCET analysis are combined together: states are defined as a set of properties of interest in the system updated over time, such as task priorities, RTOS modes and privilege levels and applications states, which permit to better characterise the running system as a whole. This kind of system-wide knowledge is then exploited to perform cache-analysis, to calculate pipeline-related preemption costs and to improve WCET accuracy according to actual RTOS and application execution conditions. Similarly, Chong et al. [68] have recently proposed an integrated analysis of the microarchitecture, OS and application code runnning as a real-world robot controller; although the approach tries to combine different analysis techniques, significant overestimation still emerges from the presented results, as a consequence of relying on static timing analysis bounds used to characterise microarchitectural effects.

CODE STRUCTURE. Much of the information required for RTOS analysis is determined by the behaviour of the running system and cannot be leveraged therefore by static analysis techniques. For example, code constructs that break any locality assumption such loop bounds, actual targets of dynamic function calls and jump destinations are hardly-predictable yet mandatory parameters to be fed to static analysis.

This problem is present also at the application level and is investigated later in this work. In general, it may be argued that this is not a problem of the code alone, but also of the sensitivity of the analysis method. When speaking about the OS though, implementation as well as design should be carried with time composability in mind, which unfortunately is not always the case.

LOW-JITTER OS. One promising direction of investigation that seems to be only partially explored is making RTOS services constant-time or at least bounded by a (small) constant factor, so that a fixed overhead can be added to the calculated application WCET as a consequence of their invocation. Yang et al. [69] studied a fixed-*interrupt-latency* kernel by removing Interrupt Service Routines (ISRs), which usually execute in privileged mode and naturally present variable execution times according to the service to be provided. Their job is performed instead by preemptible kernel threads executing at the highest priority levels in the system, whose behaviour is not any different from application tasks and therefore can be easily modeled by the analysis process. Another parameter to consider and bound is *preemption latency*: as sketched in Section 2.1 this work is most challenging in architectures equipped with advanced hardware features, like caches and virtual memory, since a variable amount of cache blocks and TLB entries must be invalidated depending on the context of the preempted task. Jitter is also introduced at scheduling points by the execution of the scheduler itself, and is measured by the *scheduler latency*, whose overhead usually varies in general-purpose systems with the number of tasks present in the system and with the number of those actually ready for execution at every scheduling point. A constant-time scheduler can be found in recent Linux kernel implementations [70][71], and is preferable for real-time systems because of its predictable overhead. Finally, the *protocols* defined to provide mutual access to shared system resources and data structures, such as memory-mapped I/O and communication media may introduce latencies and increase interference: protocols devised to alleviate such problem exist, as the priority ceiling and priority inheritance, but their implementation and use in operating systems and programming languages might not be obvious and should be carefully verified.

PREEMPTION AND INTER-TASK INTERFERENCE. While traditional WCET analysis looks at the execution of application code as if its execution were isolated and uninterrupted, the interaction with an operating system layer dramatically breaks this assumption. Techniques exist in response-time analysis to account for the inter-

ference suffered from other tasks in the system; calls to system services, interrupt servicing, scheduling, preemption and all activities to respond to asynchronous events are instead highly unpredictable phenomena. Not only the time required to serve such events is naturally jittery, but also the machine state is typically modified during their execution: therefore, additional time is required to restore the execution context and resume application execution from where it was interrupted. The state of cache memories can be changed as well, with the additional complication that the overhead in this case depends on the memory accesses performed in the future execution and is therefore much more variable and inscrutable. In [72] Sandell et al. take into consideration Disable Interrupt (DI) regions and compiler optimisations concluding that some portions of a RTOS kernel are more easily analysable by static analysis than others, but this is a rather restrictive setting for the general problem. Limiting the occurrence of preemption has proven to be a more effective solution, as described next.

LIMITED PREEMPTION. The idea of limiting preemptions to reduce interference caused to a task set and increase schedulability is not new to scheduling theory. Inhibiting preemption altogether is not an option for those system whose feasibility must be strictly guaranteed, as the optimality results obtained for fixed-priority preemptive scheduling do not hold any more [73]. This observation motivates a recently increasing interest towards less restrictive approaches aiming at finding a compromise between reducing the effects of preemption and preserving the schedulability of a task set [41][42]. In deferred preemption approaches in particular emphasis is placed on the fact that preemption requests are not immediately served but may be rather deferred (i.e. postponed) for some duration of time. Preemption deferral mechanisms have the practical effect of splitting the execution of each task in preemptible and non-preemptible parts. Non-preemptive regions (NPR), within which preemption cannot take place, are either determined by fixed points in the task code or triggered by specific scheduling or timing events. The former approach is indeed exploited by one of the first attempts to introduce deferred preemptions within a fixed-priority scheduling scheme (FPDS), described by Burns in [74]: the cited work proposes a cooperative scheduling approach, where each task should explicitly request to be preempted by invoking a "yield" routine in the kernel. Although the blocking time due to preemption deferral was incorporated in refined response-time equations, this seminal work did not address the problem of computing the maximum allowable length of non-preemptive chunks. More recently, Bril et al. [75][76] improved this

model by showing that current analysis was not precise, i.e. it can be either optimistic or pessimistic according to the specific scenario; the authors proved that in the general FPDS case the worst-case response time of a task cannot be calculated by only looking at its first job, since the delay introduced by the last non-preemptive chunk of a task on higher-priority jobs may indirectly delay its subsequent job activations (self-pushing phenomenon). To address this problem the authors used the notion of *level-i active period* and refined the analysis to account for it. Yao et al. showed in [77][78] that no self-pushing can occur under the fixed preemption points (FPP) model if two favourable conditions are met: (i) the task set has constrained deadlines; and (ii) it is preemptively feasible. Unfortunately, the FPP approach suffers some drawbacks that make its adoption impractical in real-world scenarios. Preemption points should be explicitly programmed into application code, which is hard to do in the presence of legacy code, and their placement needs to be reconsidered every time a new task is added to the system, thus making the approach unsuitable for systems subject to change over time. Additionally, the choice of preemption points placement is a challenge itself, which as such has been addressed in some works with the specific goal of increasing system schedulability [79] or minimising preemption overhead [80]. Alternative and more flexible approaches instead implement non-preemptive regions without relying upon their explicit position in the code so that NPRs are said to be *floating* in the code. Since there is no explicit positioning, NPRs should be initiated at a certain time stamp or in response to some kind of event as, for example, the activation of a higher priority task (*activation-triggered* approach, see Figure 5). In floating NPR approaches the only relevant issue is thus to determine the maximum admissible length of a floating NPR that preserves feasibility of the system. These works all assume job arrival at their critical instant to ensure the safety of the analysis: however, when job phasing is introduced in the task set, the length of NPRs can be further extended with respect to the worst-case scenario of simultaneous job releases, as explained in [81]. Unfortunately, no real-world implementation of a limited-preemptive scheduler exists and the validity of published results has been proven by simulation only. Some initial considerations on implementation issues are given in [82] and [83], for the mechanism of ready-queue locking and for memory occupancy and resource sharing, respectively. Marinho et al. [84] have also investigated the beneficial effects on the cache-related preemption delay suffered by each task under a NPR scheme, as a function of the reduced number of preemptions. An alternative and valid approach for limiting the effects of preemption is the Preemption Threshold Scheduling (PTS) method introduced by Wang and Saksena [85][86]: although the authors showed that

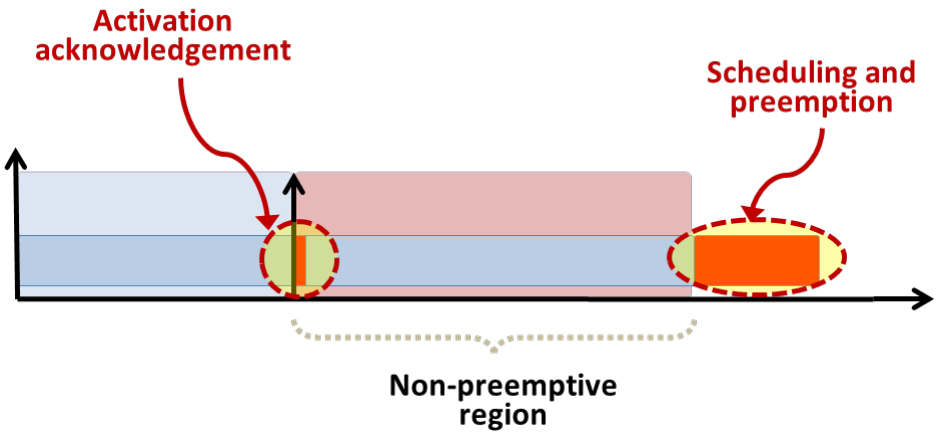


Figure 5.: The activation-triggered NPR model.

schedulability can be improved by PTS, difficulties arise in the choice of thresholds. Since preemption is still enabled in PTS, this scheduling technique can behave exactly as fully preemptive or non-preemptive scheduling according to the specific threshold tuning, which is therefore crucial to achieve the desired results. As an additional consideration, none of these works takes into account the presence of shared resources explicitly: they either ignore them or assume that no preemption can occur within them. Of course, this assumption does not hold in the general case and may be violated in the presence of non-preemptive regions whose placement is unknown at design time. A solution to this problem will be presented in Section 3.6.3.

2.3 APPLICATION ANALYSIS

In Sections 2.1 and 2.2 we identified the main obstacles in the present hardware and software architectures, which make accurate HIRTS timing analysis unaffordable or altogether impossible. Obviously, the design and implementation of the application software itself also play a central role in determining the complexity of the analysis process and, in the end, its feasibility. In the first place the Model-Driven Engineering approach can help shape the system architecture to limit and characterise component dependencies and interactions; additionally, it can significantly improve analysability by making automatic code generation from formal specifications possible. Application coding without automated assistance, more than being a costly and error-prone activity, is heavily influenced by the chosen programming language itself, whose constructs may benefit or harm timing analysability.

Literature on WCET analysis of application code is vast, although any effort in the direction of making application code analysis-friendly is of limited usefulness if not supported by a time-predictable execution stack. We now try to summarise existing research into a few main streams, highlighting the breaking points of common analysis approaches. It is certainly desirable to use programming languages that make the analysis process easier; unfortunately, this idea may be considered the counterpart at software level of using custom hardware: any new language is condemned to little penetration in industry where established knowledge, common practice and legacy code tend to oppose to change. For this reason measurement-based and hybrid techniques are dominant for WCET estimation in the industry, as they permit to look at the application as a black box.

2.3.1 *Coding Styles*

From the very first studies on application timing predictability the focus has been put on the expressiveness gap between language constructs and application needs: the smaller it is, the easier the structure of the resulting code and consequently its analysis should be. Unfortunately, the language alone cannot guarantee predictability, since the human component still present in software programming finally determines the code structure, i.e. its control flow graph. Although in principle all modern programming languages are expressive enough to build any kind of software regardless of its purpose, common experience suggests that choosing the "right" programming language is an important choice affecting both the final result and the development process itself: the adequacy of a language can be imagined in terms of the primitive constructs it provides relatively to the expressiveness needed by the specific application. Some programming paradigms in their entirety fit bad with the time requirements demanded by real-time systems: although an entire area of research is dedicated to this family [87][88][89], unrestrained *object-orientation* generally is not suited for HIRTS development, since many of its core features, like the use of dynamic binding, dynamic memory allocation/deallocation and garbage collection, intrinsically clash with the exigence of predictability of static analysis.

LANGUAGE RESTRICTIONS. Application timing analysis can benefit from constraining language expressiveness for better analysability. Kligerman and Stoyenko [90], Puschner and Koza [14] and more recently Thiele and Wilhelm [91] have spotted some language features that obstruct static code analysability, and proposed some counter-

measures: *pointers* to variables and functions can significantly decrease precision on bounds as they cannot be resolved until run time, therefore function calls and their parameters should be made explicit; *dynamic task creation* and *dynamic data structures* as well cannot provide static guarantees, since their number and dimension can grow and drop during execution. Additionally, the use of recursion and `goto` statements is not permitted, and loops can be considered only provided that the number of their iteration is statically bounded. Later on, Puschner and Schedl [92] formalised the computation of maximum execution time as an integer linear programming problem on a representation of the control flow as a graph, and Kirner [93] proposed an extension to the popular ANSI C language realising these ideas. A similar approach is taken by Carré and Garnsworthy in the formulation of SPARK [94], a subset of the Ada language to support program predictability and formal analysis.

CONTROL-FLOW RESTRICTIONS. Rather than considering programming language constructs alone, further research has been conducted on the higher-level structures of software, i.e. the basic blocks forming its control flow graph. A nice categorisation of some ordinary code patterns that hinder the analysability of a program is reported in [95] and [96]. Since every branch point appearing in the application code breaks the execution into two or more scenarios, each determining different machine states and typically presenting different execution times, the number of possible paths is exponential with the length of the given code; detecting and removing from the analysis the subset of infeasible paths is therefore fundamental to prune the state search space. The easiest way to characterise and reconstruct the control flow of a portion of code is by providing annotations to express flow facts [97][98]: some of them can be derived automatically, as they are structural and can be inferred from the program syntax, others must be explicitly provided by the programmer, as they concern the intended semantics of the program [18]. As the control flow of a program typically depends on its input, the drawback with this approach is clearly that it cannot be fully automated, which makes it costly and error-prone. A more drastic approach is presented by Puschner [99][100], in which WCET-analysable code[1] is transformed into single-path code. Conditional statements are replaced by "conditional-move" assignments where each branch is calculated beforehand; loops are also transformed in two steps, by first generating a conditional statement from the original termination condition, and then transforming it into a "conditional-move" assignment as before.

1 In this approach code is said to be WCET-analysable if the maximum number of loop iterations for every loop block is known.

Some experiments were performed on the single-path programming approach [101]: they show that precise WCET can be calculated, although some performance loss cannot be avoided for legacy code, and the guidelines for a WCET-oriented programming style are drawn. Puschner also considers the problem from an architectural point of view, observing that this approach has higher potential when working in cooperation with (i) a compiler generating code without branch instructions that may cause potential pipeline stalls at the machine level and (ii) a processor implementing branches with single machine instructions.

2.3.2 *Control Flow Analysis*

The power of the single-path approach illustrated in the preceding section is that it makes the machine state predictable, as in fact only one state exists; however, the structure of real-world and legacy code is far more complex. In this section we show how WCET bounds are calculated for the existing control flow paths in an application: static analysis methods usually derive bounds on paths bottom-up by composing finer bounds on basic blocks. This method needs to assume that the behaviour of the underlying hardware is fully controlled and does not introduce any undesirable or unexpected effect which may invalidate the considerations made at the software level; equivalently, the assumption is that the latencies of its primitives are negligible. The key point with static techniques is capturing context dependencies among basic blocks, which make their composition much more complex than a simple summation of the individual WCET bounds. Measurement-based techniques instead are directly concerned with how control flow paths can be actually traversed at run time, to ensure that the longest of them are exercised by measurements and the real WCET is captured appropriately. Different paths are determined at branching points, where data-driven decisions are taken: for this reason different paths are stimulated by providing different input sets to the software under analysis. Yet, the problem becomes finding an appropriate classification of the possibly infinite set of inputs, which guarantees sufficient accuracy of the analysis process.

COMPOSING BOUNDS ON BASIC BLOCKS. If the execution times for individual basic blocks on single control flow paths are available, three classes of methods exist to combine them into end-to-end estimates. In *structure-based* methods [102] the syntax tree built by control-flow analysis is traversed in a bottom-up fashion and WCET values for single nodes are collapsed into bounds of coarser granularity, according

to a composition rule specific for the kind of node. Because of the sensitivity of the analysis to the execution history, this step can be tricky: safeness of aggregated results can be ensured only if bounds on its individual constituents have been calculated in virtaully all possible context flows. Alternatively, it has been proposed [103] to build up the execution time of a path probabilistically. *Path-based* methods consider program paths and determine the WCET of a task by choosing the most expensive one: since the number of paths can be large and execution times for every path need to be computed explicitly, path-based approach can work only when the branching factor of the software under analysis is limited. To overcome this limitation, path clustering has been proposed to factor out paths presenting similar execution time bounds [104]. In *implicit path enumeration* techniques [105] program-flow and basic-block execution time bounds are combined together to form a set of arithmetic constraints; each of them is given a coefficient, expressing its maximum contribution to global WCET, and a multiplicative factor counting the number of times the entity is executed. The resulting global timing bound is obtained by maximising such summation of products with integer linear programming or constraint programming techniques. Elementary components have usually the granularity of a basic block, but some studies consider composition at the instruction level [106].

INPUT CLUSTERING. The issue when considering the input of a non-trivial program is that its exhaustive enumeration is infeasible, and this is even less tractable problem than control flow paths enumeration, since in principle different input sets may activate the same path. One technique to attack this problem is program clustering: Deverge and Puaut [107] propose to split paths into smaller segments to lower the complexity of test data generation, and then apply measurements on the produced segments. However, this method suffers from its naïve assumption of context-independence, thus being exposed to unpredictable latencies of hardware and to non-local interference. Fredriksson et al. [108] consider the same problem at the component level: since components are designed to be general enough for reuse in different contexts, it is the case that parts of them are only used in certain contexts. From this observation authors try to cluster inputs to according to different contexts rather than on control flow paths. The idea is promising, however some restrictive assumptions apply, such as independence of the variables used to characterise usage conditions which makes it harder to capture more complex effects. Additionally, one could argue that the problem is only shifted to clustering usage scenarios, and indeed the main problem with this approach is searching that space efficiently.

USE OF HEURISTICS. When the complexity of a problem is overwhelming, it is often tempting to adopt some heuristics to reduce the size of the search space. This principle has been applied to choose representatives of the input space to be used to perform measurements: evolutionary testing based on genetic algorithms has been proposed in [109][110][111][22]. The main problem with these techniques is usually tuning the algorithm of choice: choosing the appropriate crossover point for recombination has great impact on the convergence of the algorithm; also it is not straightforward to understand when the algorithm should stop iterating. Although providing good results in some benchmarks, the use of heuristics can lead to suboptimal solutions, which means tolerating unsafeness of calculated WCET bounds, which is of course unacceptable.

2.3.3 *Compiler Support*

When analysing the predictability of application code, it is important to bear in mind that what is written at the source level is not necessarily mapped one-to-one to machine-level instructions, which in fact determine the actual WCET value to be calculated. In between the two layers, the compiler is responsible for transforming the human-readable source code into a machine-executable flow of instructions: the introduced transformations can be so extreme that the original control flow of a program is heavily modified, especially when strong levels of optimisation are used. Not only analysis at the source level is easier for human beings, but also some constructs are better analysable at this level; conversely, analysis at the object level is capable of capturing microarchitectural effects on the real target and does not suffer from aggressive compiler optimisations. Information from both layers must be related, and since today's compilers already perform deep program analysis, they may appear a good candidate to bridge the cross-layer abstraction gap and enable the analysis. In [112] Bernat and Holsti identify some desirable features at the compiler level to express properties on the source and object code, on the mapping between them and on the code generation process itself. The support of compilers for WCET analysis could also bring some benefits by limiting the use of manual annotations [113], as many of them could be automatically generated and only a few could be provided at the source code level. An interesting example is provided by Kirner and Puschner in [114], where GCC-generated intermediate code is chosen as the place where knowledge from the user is leveraged to drive WCET calculation, and the computed results are finally used to back annotate source code in a human-readable form. A similar approach proposed by

Gustafsson et al. [115] uses abstract interpretation to automatically generate flow facts from an intermediate representation of programs written with the C programming language. More recently Curtsinger and Berger [116] have proposed to apply new transformations in the compiler with the purpose of randomising code placement at runtime to enable probabilistic analysis. Despite these efforts, some of the usability concerns mentioned for programming languages also apply here: compilers meant to be used for HIRTS development must satisfy stringent requirements to sustain certification arguments and the penetration of new technologies usually faces some difficulties.

2.3.4 *Component Analysis*

So far, we presented existing approaches to time composability at the level of code structures; however, since applications are designed and implemented by taking benefit of component-based methodologies as explained in Section 1.1.1, achieving time composability directly at the component level would be an attractive goal. Indeed, it is a recurring idea in the literature that WCET analysis should be conducted as an incremental, two-step process [117]: a first phase called abstract analysis should consider the higher-level components of the system to produce a reusable, portable WCET; a later phase called concrete analysis should then instantiate the computed results to the target platform. This would significantly ease incremental and reusable analysis of generic components, but requires an integrated approach towards WCET analysis which is seldom found in state-of-the-art techniques. Components have no knowledge of the context in which they are to be deployed and are therefore the natural candidates for composability and reuse; because their design is kept general enough, WCET analysis is even more complicated. A unifying framework is proposed in [118], where components are supposed to make some input assumptions, corresponding to expectations they have on other components and the environment, and to provide some output guarantees, symmetrically. Composition is achieved on an assumption/guarantee basis, similarly to how provided/required interfaces are used for functional behaviour in component-based development techniques. This approach is however very generic and does not explicitly address the provision of timing guarantees. Fredriksson et al. [119] propose a framework in which WCET is first calculated at the component level, and then mapped to the running tasks of a real-time system: this distinction emphasises the separation between functional and timing properties, but also shows that a conceptual model to reason about

and design for predictability is missing in traditional methods. Input clustering is adopted at the component level to handle the complexity of the input space, similarly to what is done for control flow paths. The proposed method also accounts for usage scenarios: they are described by sets of variables, whose values are associated to probability distributions. WCET bounds are finally simply added together, but it is questionable whether this is sufficient to capture more complex interactions; the authors themselves declare that WCET is potentially tightened with respect to traditional static analysis, but that at the same time it is difficult to reason about the general improvements of the method. A similar approach is taken by Lisper et al. in [120], where properties are composed by summation, assuming uninterrupted, sequential execution. Since this assumption is quite restrictive and independence of components from one another and from the execution context is required, the fractional factorial design methodology [121] is chosen to perform statistical analysis on input data. However, this framework cannot capture interactions among more than two parameters, and the use of both branches and parameters in function calls are excluded from experiments. Additionally, WCET composition can also be performed by convolution of probabilistic timing profiles rather than on single values, but the independence assumption for components is still required [122]. An interesting hybrid approach [123][124] performs timing analysis at the component level borrowing from the path analysis techniques developed by control-flow analysis; the functional behaviour of components, their WCET computed in isolation and the pre-/post-conditions in place on sub-components data flow are translated into the control, data flow and timing constraints of a constraint programming problem, respectively. The method requires that there are no loops in the component model and that the control flow is formally and completely defined; moreover, since bounds are calculated with a parametric approach there is always the issue of ensuring sufficient coverage for all possible use cases; finally, an open problem is dealing with glue code among components, that certainly influences WCET results. The merits of this framework are the achievement of better context sensitivity by including control-flow information, and the composition of tighter WCET bounds with more refined operators, rather than by naïve summation of WCET values.

2.4 COMPOSITIONAL TIMING ANALYSIS

We focused so far on techniques that try to attain time composability on the elementary components of a system architecture: the main challenge is however preserving

the properties defined at lower levels of abstraction bottom-up in the composition process. If the property of composability were satisfied, one could leverage the existing frameworks to describe the timing behaviour of complex systems top-down, by only taking a compositional view on the system. Unfortunately, these frameworks face some difficulties in their applicability to real-world scenarios as a consequence of the lack of composability at the lower levels of the architecture.

The most interesting research in this area is represented by hierarchical scheduling frameworks [125][126], which find their natural application in partitioned architectures [127]. The idea is to define a hierarchy of schedulers such that global timing properties at the system level can be derived by composing local timing properties defined at the component level. This form of schedulability analysis can be performed in a two-step process, which first abstracts resource requirements for single components and later composes them in an incremental fashion. The challenging part is again determining safe and tight execution time requirements for individual components, which is usually very difficult for non-trivial systems, as we already observed.

Another branch of research leverages the existing work on timed automata and temporal logic [128][129]. These approaches usually describe execution as a progression of discrete events, assuming a degree of accuracy which is in principle available in a digital processor, but which is then lost upwards along the execution stack. Furthermore, composition of temporal relations is defined in terms of relative input/output interfaces, while execution and communication are supposed to be synchronous. Extensions to these frameworks also exist [130], which adapt the Real-Time Calculus to the exigences of probabilistic real-time systems, and in particular of probabilistic schedulability analysis.

Research on composability is also inspired by distributed real-time systems [131][132]; in this area component-based design is a natural choice, since components may be deployed on physically separate nodes. Communication in these systems is performed across a network and is therefore subject to significant latency, when compared to inter-component communication of an application running on a single chip. For this reason work in this area usually adopts the time-triggered paradigm [133][134], in which events occur as a consequence of the passing of time, similarly to what happens in the cyclic executive execution model. These systems are known to be more predictable but less responsive to the external world, since scheduling and interrupt servicing is limited to statically defined points in time, thus increasing latency. In fact, time-triggered system seem to enforce full determinism rather than timing

predictability and for this reason they cannot be fairly compared with event-driven systems.

2.5 ASSUMPTIONS

Below we explicit the two main assumptions we need to consider in the prosecution of our thesis, respectively on the hardware platform and on the probabilistic analysis method.

2.5.1 *Hardware Platform*

The OS, which is the main focus of our investigation, provides an abstraction layer for the operating system primitives and services. From the timing analysability standpoint, the role played by this layer is possibly as important as that played by the hardware. As a fundamental enabler to compositional analysis approaches, in fact, the kernel should preserve the degree of independence from execution history exhibited by the underlying hardware and should not introduce additional sources of timing variability in the execution stack. In this thesis, we do not focus directly on HW-related composability issues, although we are perfectly aware of their relevance, especially with respect to the possible occurrence of timing anomalies [135]. We assume, instead, the availability of a hardware where interference from execution history on the timing behaviour has been proactively countered. This is not an unrealistic assumption since it can be achieved by means of simplified hardware platforms [34] or novel probabilistic approaches, as suggested by PROARTIS [16]. In the scope of our investigation, we focused on the PowerPC processor family, and the PPC 750 model [136] in particular, by reason of its widespread adoption in the avionic industry. The PPC 750 was adopted as the reference hardware by the PROARTIS project and is therefore suitable for probabilistic timing analysis.

In the following we give an insight on some representative HW platforms proposed in recent research, which have been explicitly designed with time analysability in mind and try to remove dependence on execution history. Thanks to this design choice these architectures would be the best candidates to provide a neutral HW layer which does not impair the time composability results obtained at the kernel and application layers.

PRET. The motivations for the study on the PREcision Time Machine (PRET) architecture [34] build on the observation that the timing properties of individual functions in which a large system is split by compositionality at design time may be destroyed in the integration phase; this is a consequence of the timing interference caused by unrestricted access of shared hardware resources in modern computer architectures. Edwards and Lee [137] proposed a paradigm shift in the design of computer architectures, focusing on timing predictability instead of average-case performance; later on, they proposed PRET as an architecture designed for timing predictability. PRET employs a thread-interleaved pipeline with scratchpad memories, and comes with a predictable DRAM controller. The concurrent execution of multiple (hardware) thread contexts on a shared platform is decoupled by making instructions independent of their execution context within the program; moreover, no cache nor modern memory controllers are employed so that the latency of memory operations is independent from any previous memory access. Removing execution interference enables the composable integration of different threads running on a shared platform, while preserving the temporal properties of interest. This eventually facilitates a simple and accurate architectural timing analysis of contexts associated to programs.

T-CREST. The T-CREST project [138] aims at defining a high-performance, time-predictable HW architecture for safety relevant applications, whose complexity and timing analysis costs are kept low to make it an attractive alternative in systems where robustness, availability and safety are important requirements. While the scope of the project covers the design of a complete platform equipped with time-predictable resources, such memories, a compiler [139] and an interconnection network [140], a big effort is put in the design of its processor, called Patmos [141]. This is a 5-stage, RISC processor which uses several local memories (for methods, stack, data and a scratchpad), which are guaranteed to never stall the pipeline, with the goal of reducing memory operations latency. Although the time-triggered model of computation and some of the deriving design choices (like the TDMA access policy to the on-chip network) have often been criticised for their sub-optimal performance, this choice is justified by the emphasis on time predictability for easier and tighter WCET analysis. The TiCOS kernel we developed – and which we report on in Section 3.4 – has been succesfully ported to the T-CREST architecture.

PROARTIS. The PROARTIS project [30] aimed at defining a HW/SW platform and the needed methodologies and tools to enable probabilistic timing analysis of safety-

critical, real-time embedded systems. As explained in [142], which we quote verbatim in the following, PROARTIS classifies processor resources into 3 categories, according to the jitter they exhibit: *jitterless resources* have fixed latency, with no dependency on the input or the past history of execution, and are therefore easy to model even with static timing analysis techniques. *Jittery resources* instead manifest variable behaviour which may depend on the input data they are fed with, their state as determined by the past history of execution or both. Depending on the relative degree of jitter they present and the tolerance by the analysis process, it may be acceptable to always assume their worst-case latency or, alternatively, to time randomise their behaviour. *High-jitter resources* are those for which even assuming the worst-case behaviour is unacceptable, and for this reason their behaviour is described by time randomisation. Each such resources R_i is then assigned an Execution Time Profile (ETP) defined by two vectors as $ETP(R_i) = \{\vec{t_i}, \vec{p_i}\}$, where $\vec{t_i} = \{t_i^1, t_i^2, ..., t_i^{N_i}\}$ describes all the possible execution times of R_i and $\vec{p_i} = \{p_i^1, p_i^2, ..., p_i^{N_i},\}$ the corresponding probabilities of occurrence (summing to 1). The execution time profile of a processor instruction using some HW resources is then computed by convolution of the individual ETPs of the accessed resources. Probability theory guarantees that the process is correct whenever ETP probability distributions satisfy the property of independence and identically distribution, meaning that both the execution times and the probability values associated to them must be independent of the history of execution in the system. Once HW manufacturers have provided PTA-compliant hardware and this requirement is satisfied, the analysis process can focus only on the application looking at the execution platform as a black box.

2.5.2 *Probabilistic Timing Analysis*

As described in [29] and in our SIES 2013 publication[2], the Measurement-Based approach to Probabilistic Timing Analysis follows the five steps depicted in Figure 6 and briefly recalled below:

1. *Collecting Observations.* The first step of the analysis consists in collecting the execution time observations of a number of run of the application under analysis.

2. *Grouping.* The Block Maxima method is applied, which randomly picks execution times from the set of all runs and groups them into sets of given size; from those groups the highest values are recorded.

2 Please refer to the "List of Publications" section at the beginning of this thesis.

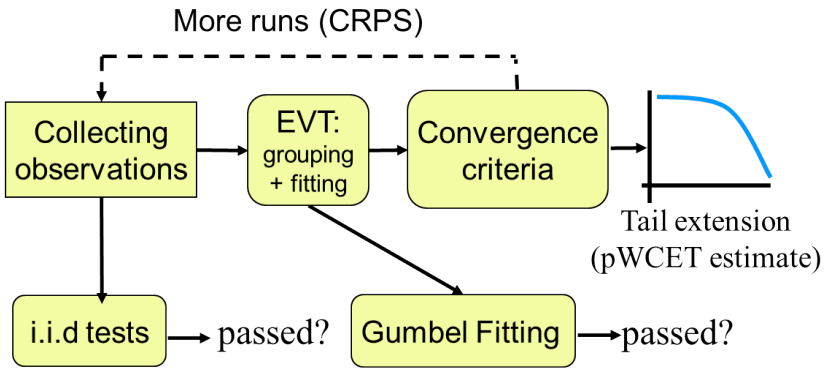


Figure 6.: Steps in the application of the MBPTA-EVT technique.

3. *Fitting.* An EVT distribution F is derived from the set of execution times generated in the previous step F is characterised by the shape (ξ), scale (σ) and location (μ) parameters as follows:

$$F_{\xi}(x) = \begin{cases} e^{-(1+\xi\frac{x-\mu}{\sigma})^{\frac{1}{\xi}}} & \xi \neq 0 \\ e^{-e^{-\frac{x-\mu}{\sigma}}} & \xi = 0 \end{cases}$$

4. *Convergence.* In this step it is decided whether enough data (i.e. minimum number of runs, MNR) have been observed to have sufficient confidence on the resulting EVT distribution. This is done by using the Continuous Rank Probability Score (CRPS) metric [143] to compare the distributions F resulting from the previous and the current iteration: fixed a confidence threshold, smaller CRPS values suggests that F is sufficiently precise and can therefore be safely accepted.

5. *Tail Extension.* Chosen an exceedance probability threshold, a pWCET estimate is extracted from F.

Two examples of the application of the PROARTIS method is given in the case studies in Sections 3.4.4 and 4.2.1.

2.6 SUMMARY

Although some analysis frameworks try to characterise the timing behaviour of a system top-down, their applicability in practice is limited by the practical impossibility of precisely characterising the behaviour of hardware and software components that

are either too complex to capture or obscured by lack of information. Most of existing research on timing analysis tries therefore to build WCET bottom-up and focuses on each layer of the system architecture separately. At the hardware level, attention has been given to those accelerating features which are intended to speed up the average execution time but whose worst case is hard to predict, like caches, pipelines, virtual memories and branch predictors. At the application level the coding style, as determined by both the programming language in use and the resulting control flow, has been recognised to largely affect analysability. The control flow itself may be considered at different abstraction levels, spanning from the interaction of components down to machine code instructions. In any case, depending on the pattern of use of shared HW resources and the invocation of OS services, computing the WCET of a non-trivial piece of code may result very cumbersome. The operating system itself has been studied to reduce its overhead on invoking applications, mainly by reducing its jitter and by limiting background activities – like scheduling and preemption – to the minimum. Finally, part of research has looked at software run times and compilers to bridge the gap between communicating layers.

In our work we assume a "well-behaved" HW layer which does not impair the time composability results obtained at the OS and application levels: this assumption is met by existing proof-of-concept platform recently proposed in top-tier research. In particular we look at the PROARTIS architecture, as one result of the FP7 research project we were involved in.

3

TIME COMPOSABILITY IN SINGLE CORE ARCHITECTURES

In previous chapters we motivated the importance of time composability in modern architectures to enable incremental development and verification of HIRTS. We also identified two places in the architecture where time composability easily risks to be broken, namely in the unaware provisioning of HW/SW features whose timing behaviour is not predictable and context-dependent and in the interactions between architectural layers. We now elaborate on these ideas to focus our investigation on time composability in single core architectures by first giving a precise characterisation of *time composability* at the OS level (Section 3.1). We assume to execute on a hardware platform which is well-behaved with respect to time composability, i.e. whose design favours analysability rather than performance: although this is seldom the case for nowadays COTS platforms, we singled out in Section 2.5.1 some potential candidates resulting from active research in this area. We then spot in Section 3.2 some areas of intervention we identified in an OS where wise implementation choices can make the difference in preserving or impairing time composability. From there on, the chapter is conceptually split into two parts, corresponding to the two industrial domains we had the opportunity to investigate in our work, avionics and space. Although they share similar concerns about timeliness and safety, the industrial process they adopt for system delivery is quite different, as will be explained in Sections 3.3 and 3.5, respectively. For both settings we then explain the work we performed on two real-world kernels, TiCOS (in Section 3.4) and ORK+ (in Section 3.6), to make them compliant with the enunciated principles of time composability. In particular, in Section 3.6.3 we discuss a limited-preemptive scheduling approach which is applicable to the space domain and provably benefits time composability in ORK+. Finally, we provide evidence in Sections 3.4.3 and 3.6.4 to demonstrate the improved degree of time composability achieved in both scenarios.

3.1 AN INTERPRETATION OF TIME COMPOSABILITY

The property of time composability postulates that the timing behaviour of any given component is not affected by the presence and activity of any other component

deployed in the system. Interestingly, this definition makes no difference between hardware and software, and thus equally applies at any level of execution granularity. For the purposes of feasibility analysis, the timing behaviour of a component is fully represented by the WCET bound given for it. Components seen from the analysis perspective include the RTOS and the application, considered as a whole or parts thereof as the analysis needs dictate.

From the application viewpoint, an application program is time-composable with the RTOS if the WCET estimate computed for that program is not affected by the presence and execution of the RTOS. When this condition holds, the internals of the RTOS might be completely replaced – while of course maintaining the functionality provided to the application – and the WCET of that application program, as determined for the specific hardware execution platform, would *not* change.

The multifaceted role of the RTOS in influencing the time composability of a system is evident from the classic layered decomposition of the execution stack (see Figure 3 in Chapter 1), which shows how the RTOS is deeply engaged in both the hardware-to-software and software-to-software axes of composition. When it comes to operating systems, the definition of time composability certainly includes the fact that the OS should exhibit the same timing behaviour independently of the number and nature of other run-time entities in the system: for example, the execution of a scheduling primitive should not vary with the number of tasks in the system. In addition, as the operating system interacts and consequently interferes with the user applications, it should also avoid to negatively affect their execution time jitter: especially in the presence of hardware features that exhibit history-dependent timing behaviour, the execution of the OS should carefully avoid disturbing effects on the application. Enabling and preserving time-composability in the kernel layer poses therefore two main requirements on the way OS services should be delivered (Figure 7):

STEADY TIMING BEHAVIOUR. RTOS services with jittery behaviour impair time composition with the application: the larger the jitter the greater the pessimism in the WCET bound for the service itself and transitively for the caller. In a measurement-based approach, in particular, it would be difficult to attribute the measured variability either to the application software or to the invoked (and triggered) OS services. RTOS whose services exhibit a constant (or at least near-constant) timing behaviour will thus reduce the jitter to be accounted for in their response time. Timing variability in the RTOS execution depends on the interaction of software and hardware factors:

(i) the hardware state retained by stateful hardware resources;

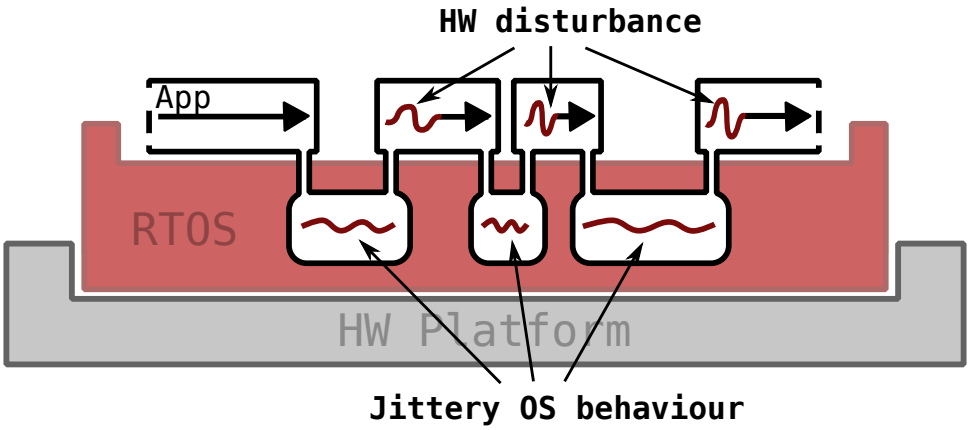


Figure 7.: Jittery OS behaviour and disturbance from history-sensitive HW propagated to application execution.

(ii) the software state as determined by the contents of the RTOS data structures and the algorithms used to access them: this includes, for example, thread queues in common scheduling algorithms, as well as the port list needed by the ARINC sampling port service which will be discussed in Section 3.3;

(iii) the input data given as parameters to the service call: this is the case, for example, of ARINC I/O services that read or write data of different size.

Factor (i) is somehow overclouded by the symmetrical issue of hardware state pollution, which is addressed by the next property. Factor (ii) instead, which vastly influences the timing behaviour, is actually determined by more or less complex data structures accessed by OS and library services and by the algorithms implemented to access and manipulate them; therefore, it needs to be addressed by re-engineering the RTOS internals to exhibit a constant-time and steady timing behaviour in the fashion of, e.g., the $O(1)$ Linux scheduler [70]). Factor (iii) is much more difficult to attenuate with general solutions, as the algorithmic behaviour of both the OS and the application logic cannot be completely detached from the input parameters, unless we do not force an overly pessimistic constant-time behaviour. In these cases, the most promising solution consists in breaking up the input-dependent part of the kernel service and either accounting for it in the application code or as a system-level background activity.

ZERO DISTURBANCE. For processor hardware that includes stateful resources whose timing behaviour suffers history-sensitive jitter, a time composable RTOS should be such that the execution of any of its services should not cause hardware-

related disturbance on application timing upon return from the RTOS routines. Since the main source of disturbance to the kernel layer and the application SW is the underlying hardware, the idea of zero-disturbance inevitably calls for some kind of separation that should be able to isolate the hardware from back propagating the effects of an OS service call. Isolation is seldom provided by means of a one-way mechanism: not being able to affect the hardware state could also mean that we are not going to benefit from it. Techniques analogous to cache partitioning [144] may serve that purpose; in cache partitioning approaches, in fact, the available cache space is split into separate partitions that can be accessed by a subset of the system tasks. In the case of the RTOS, a relatively small cache partition should be reserved for it so that the execution of its system calls would still benefit from cache acceleration without however affecting the cache state of the application. Unfortunately, implementing software cache partitioning in conjunction with a partitioned OS may be quite cumbersome in practice, as mapping the RTOS and application code to separate address spaces may be overly complex, especially for partitioned systems. An alternative (and easier to implement) approach consists in giving up any performance benefit and simply inhibiting all the history-dependent hardware at once when OS services are executed. This technique would act like a padlock that prevents any OS service from accessing (and thus altering) the state of history-dependent hardware. This comes however at the cost of relevant performance penalty that, though not being the main concern in critical real-time systems, could be still considered unacceptable. For example, disabling the cache may also transparently inhibit any kind of burst access to the main memory. In this case, cache *freezing* may be preferable to complete inhibition (Figure 8). Also the execution frequency of a service is relevant with respect to disturbance: services triggered on timer expire (such as, for example, the interrupt handler to serve a HW decrementer) or an event basis can possibly have even more disturbing effects on the application SW level, especially with respect to the soundness of timing analysis. The *deferred preemption* mechanism in combination with the selection of predetermined preemption points [145] could offer a reasonable solution for guaranteeing minimal uninterrupted executions while preserving feasibility.

Finding the most effective implementation of the RTOS features that meet the above goals is of course largely dependent on the target processor architecture as well as the adopted task model. A periodic task model suffers less preemption overheads than a sporadic one because periods can be made harmonic whereas sporadic arrivals cannot. Analogously, tick scheduling is a very intrusive style of servicing dispatching needs. It should be noted that in any case these properties should not be pursued at the cost of

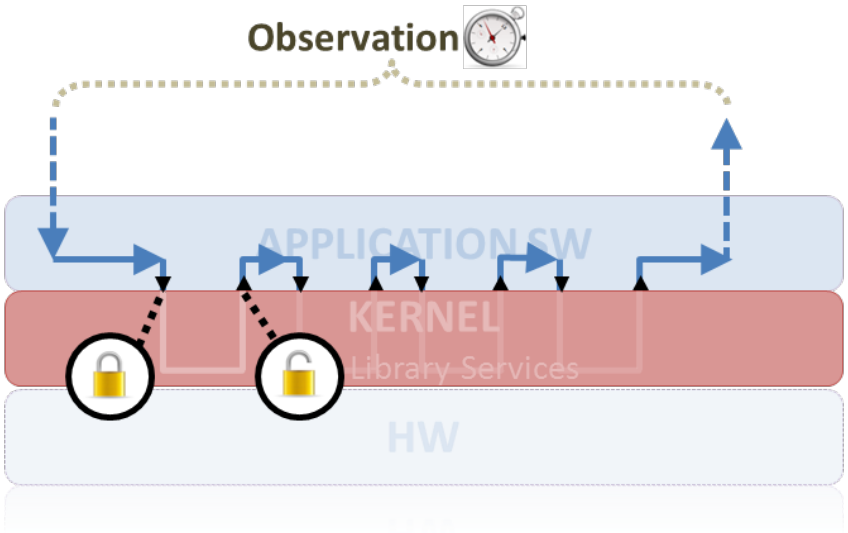


Figure 8.: Inhibition of history-dependent hardware.

performance loss: the overall performance, although not the primary concern in hard real-time systems, should not be penalised. Figure 9 summarises the steady timing behaviour and zero-disturbance requirements.

An OS layer that meets the above requirements is *time-composable* in that it can be seamlessly composed with the user application without affecting its timing behaviour. Alternative approaches could rely on a less stringent reformulation of the steady timing behaviour and zero-disturbance properties, based on the claim that timing composability between OS and application could be equally achieved if the effects of executing an OS service or HW primitive can be somehow upperbounded. These solutions range from the enforcement of a worst-case scenario (e.g., flushing the HW state) on every OS service invocation, to the computation of reliable disturbance estimates. The former approach is however unacceptably pessimistic; the latter instead is strictly dependent on the adopted timing analysis approach, and does not account for the nature itself of the operating system, which is actually separate from the user application and may not even be available to the analysis. Our proposed solution rather aims at a general reduction in the effects of the OS layer on the application code and is expected to ease the analysis process, regardless of the timing analysis technique of choice. In Sections 3.4 and 3.6 we present the implementation of a set of kernel-level services that exhibit a steady timing behaviour and cause minimal interference to the execution of user-level code.

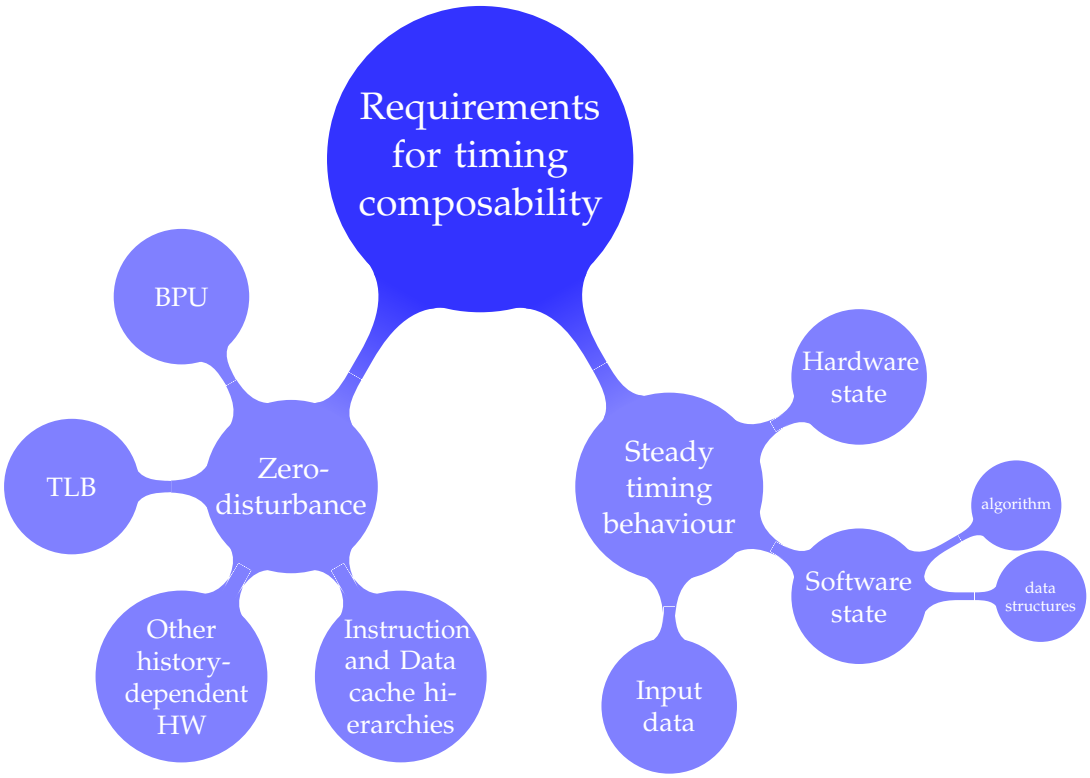


Figure 9.: Breakdown of the requirements for timing composability.

3.2 TIME-COMPOSABLE RTOS

The kernel layer, lying in between the hardware platform and the user application, accommodates the implementation of OS primitives and provides a unified access point to standard libraries and all kernel services. Whereas steadiness and zero-disturbance are universally valid principles, the practical means and the degree to which they are exhibited by a RTOS is dependent on the underlying hardware platform, the architectural specification and the task model assumed for the application. Since we consider history independence to emerge as a bottom-up property, if the HW layer were able to guarantee complete independence from the execution history then the provision of constant-time services would suffice to guarantee timing composability at the OS level. In fact, the zero-disturbance independence requirement would be automatically met: if the execution time of each processor instruction did not depend on the *hardware state* and the execution time of each kernel service did not vary on the *software state* or the input data, then the timing behaviour of the user application would not be affected by any kernel service at all. The disturbing effects on the execution of the application code can be exacerbated by the nature of a kernel service: frequent

interruptions of the user application by a hardware-triggered service (e.g., interrupt handlers) may have disruptive effects on the applicability of measurement-based approaches, as in the presence of history-dependent execution it may be extremely difficult to isolate the effects of those interference on different end-to-end runs.

An extremely restrictive software architecture resting on a simple periodic task model represents the most favourable precondition for the implementation of a time-composable RTOS. This clearly emerged during the RTOS refactoring on which we report in Section 3.4. Leaving aside exceptionally favourable circumstances (e.g., the ARINC model just mentioned), a less restrictive and more general RTOS cannot limit itself to support periodic run-time entities only. Sporadic activations, low-level interrupts and programmable timers (e.g., Ada timing events) make the quest for time composability much more complicated. The support of a more *flexible task model* and the resulting increased interactions among a number of run-time entities breaks the fairly-predictable cyclic pattern of task interleaving otherwise exhibited by a system with periodic tasks only. Moreover, mechanisms to reduce the jittery interference introduced by task preemption are required to protect time composability.

From a more general standpoint, we identify a minimal set of desirable properties, hence potential areas of intervention, that cannot be disregarded in any serious attempt to inject steadiness and zero-disturbance – hence time composability – on a full-fledged RTOS:

1. *Non-intrusive time management:* The way the progression of time is managed by the operating system should have no side effects on the timing behaviour of the applications running on top of it. Intuitively tick-based time management should be avoided in so far as it subjects the user applications to unnecessary periodic interruptions. Moreover the tick-based approach usually implies an intrusive approach to task management where many activities are performed at each tick to keep the ready queue updated.

2. *Constant-time scheduling primitives:* Most of the RTOS execution time is typically spent on scheduling activities: either task state updates (activation, suspension, resumption) or actual dispatching. Reducing the variability stemming from scheduling primitives is intuitively an essential enabler of composable RTOS. In particular, those primitives whose execution time is linearly dependent on the number of tasks in a system should be revised.

3. *Composable inter-task communication:* We intend the area of inter-task communication to cover all kinds of interactions with bearing on hardware or soft-

ware resources, including I/O, communication and synchronisation. From this perspective, the problem of providing a time-composable communication subsystems largely intersects with the problem of providing controlled access to shared resources. Uncontrolled forms of synchronisation and unbounded priority inversion manifestly clash with the steadiness and zero-disturbance principles.

4. *Selective hardware isolation:* The very concept of zero-disturbance insists on the opportunity of some form of isolation of the history-dependent hardware state as a means to preventing RTOS-induced perturbations: the provided support may vary from partitioning approaches to plain disabling of hardware resources. Clearly this is more a system-level or hardware issue than a software one, as not many solutions can be devised when the underlying hardware platform does not support any form of isolation. When necessary, however, the RTOS should be able to exploit the available features by means of ad-hoc low-level interfaces (e.g., for advanced cache management).

Although we would like our ideal time-composable RTOS to exhibit all the above properties, we are also aware that implementing such a comprehensive operating system from scratch is a challenging task. We therefore considered it worthwhile to cast time composability solutions upon existing RTOSs that would preferably already exhibit nice properties from the analysability standpoint.

We proceed in our dissertation by presenting two industrial domains, the avionics and the space, where criticality is a central concern in system development, to check how different their requirements may be and how this translates into architectural design choices. We reason about how time composability can benefit application analysability, and in turn how the specific application domain may limit the degree of achievable composability. At the same time we present the work we performed on two OS kernels, TiCOS and ORK+, in Sections 3.4 and 3.6 respectively.

3.3 THE IMA ARCHITECTURE FOR THE AVIONICS DOMAIN

In the past, conventional avionics systems were based on the *federated architecture* paradigm. In those systems each computer is a fully dedicated unit, which may undergo local optimisations. However, since most of the federated computers perform essentially the same functions (input acquisition, processing and output generation), a natural global optimisation of resources is to share the development effort by identifying common subsystems, standardising interfaces and encapsulating services,

i.e. adopting a modular approach. That is the intent of the Integrated Modular Avionics (IMA) concept [146], where multiple independent applications execute together on top of one and the same physical processor sharing logical and physical resources as needed, under the supervision of the RTOS. This contrasts with the federated architecture concept in which every application runs in total isolation on top of a distinct physical unit and fully replicated RTOS instance. The adoption of an integrated architecture opens new opportunities for software modularity, portability and reuse. In an IMA system the RTOS becomes the principal responsible for the provision of space and time isolation and the ARINC 653 specification [40] supervises how this has to be done. Space isolation is guaranteed by providing and enforcing separate address spaces for each application, typically achieved by means of different virtual memory contexts: each application, called *partition* in ARINC speak, is thus assigned a strictly distinct context on execution. Time isolation is achieved by means of partition-aware, two-level scheduling that strictly allocates all needed computational resources, including the CPU, to one partition at a time to enforce spatial isolation. The standard specifies the Application/Executive interface (APEX), a software interface between a partitioning operating system on top of the avionics computer and the application software deployed within the partitions; the ARINC architecture prescribes the semantics of intra- and inter-partition communications, and describes how to perform partition configuration. All APEX services are complemented by an API specification in a high-order description language, allowing the implementation in various programming languages and on top of diverse ARINC-compliant hardware components.

Figure 10 shows a high-level view of ARINC 653 partition-aware scheduling. Partitions execute within pre-assigned *scheduling slots* (also known as minor frames) of given duration, which may not be the same for all partitions. Within any such slot, *processes* belonging to the active partition can be scheduled for execution, typically according to fixed-priority preemptive scheduling policy. Running processes execute a sequence of *functions* specified in a process-specific schedule computed offline. Some *partition slack* may be used to fill each scheduling slot to prevent foreign activity as well to mitigate the effect of local overruns of modest magnitude. An integral number of scheduling slots is fitted into a *major frame* (MAF), which hosts the execution of all partitions within their collective hyperperiod. In addition to the primitives needed for partition scheduling, an ARINC-compliant RTOS must also provide a specialised API to regulate communications inter- and intra-partitions.

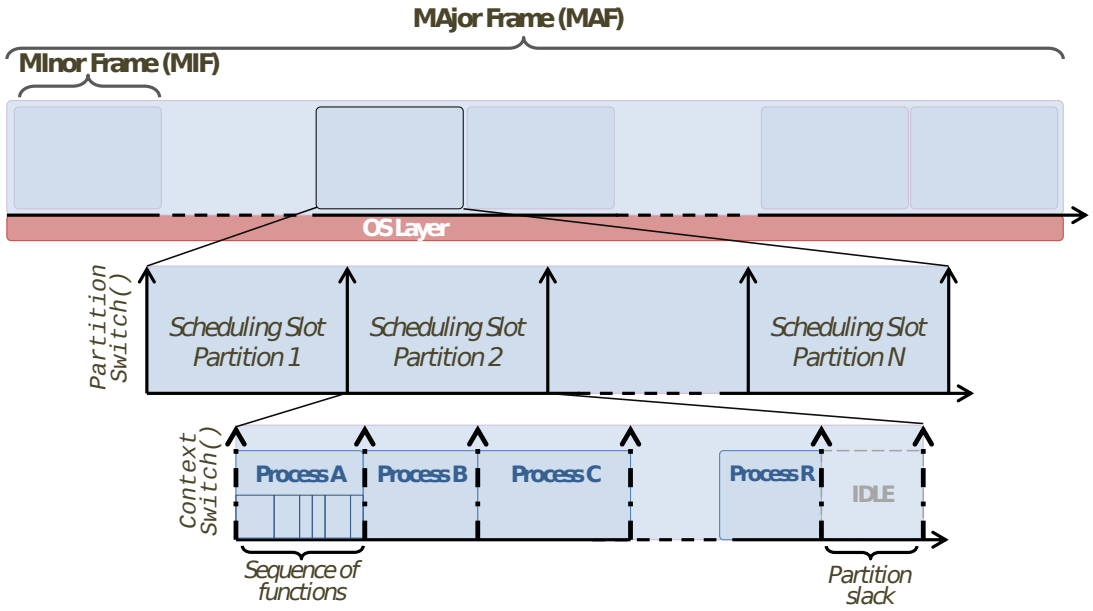


Figure 10.: Architectural decomposition of an ARINC 653 system.

A consequence that can be drawn from the architectural specification outlined above is that the ARINC task model is strictly periodic. The set of processes statically allocated to a scheduling slot is feasible if the period and duration of that slot meet the application requirements; consequently, all processes that may ever become runnable within a scheduling slot must have been statically assigned sufficient time to execute to their worst-case duration. While this is true, process dispatching within an ARINC scheduling slot is not table driven but rather takes place upon actual release and completion events.

In principle, the ARINC task model may use the partition slack to accommodate aperiodic execution, but it makes no allowances for true sporadic processes. In fact, while some ARINC primitives may give rise to event-based scheduling, ARINC itself includes no support for the enforcement of minimum inter-arrival time for release events. As a consequence, the only admissible form of sporadic process in the ARINC task model described above is assimilated to a periodic process constantly arriving at the assigned minimum inter-arrival time. This obviously causes a fair amount of wasted time in the scheduling slots hosting them.

The periodic nature of the ARINC task model and its restrictive assumptions offer several opportunities and vantage points for an RTOS to achieve time-composable behaviour, for both the specialised API services and the kernel primitives underneath

them. In particular we identified the following areas of intervention which significantly influence how time composability can be achieved in an IMA environment, and which mostly follow the categorisation presented in previous section:

TIME MANAGEMENT. The fundamentally periodic nature of the ARINC task model allows simplifying assumptions in the implementation of time management which make it possible to dispense with classic tick scheduling altogether.

SCHEDULING. In a context where processes are essentially equivalent to statically-scheduled function sequencers, preemption serves strictly no other purpose than acknowledging asynchronous events. Arguably, servicing those events can be selectively deferred to either the end of the current process or to a dedicated fraction of the partition slack, while incurring fully bounded latency and thus without intolerable performance degradation. This simplification facilitates the adoption of run-to-completion semantics [145] for processes, with obvious benefits in the terms of time composability.

SELECTIVE HARDWARE ISOLATION. The isolation of the history-dependent hardware state is an essential feature to prevent RTOS-induced perturbations. Isolation, however, would only be needed within scheduling slots at all times that partition processes execute. No isolation would be needed instead at the time of partition switches, where the RTOS is the sole entity allowed to execute: the more computationally demanding services may be scheduled for execution at those points, so long as they cause no jitter in the dispatching of partition processes and detract bounded time from the slot duration.

INTER-PARTITION COMMUNICATIONS Under the influence of their off-line computed schedule, all avionics functions fundamentally adhere to the one and same pattern of execution, which consists of three phases in succession: input acquisition, data processing, and output generation. One distinct consequence of this setting is that the RTOS processing of the inter-partition input and output calls made by functions need not be served at the point of call, thereby saving the calling process from suffering RTOS interference that endangers time composability. The obvious alternative is to position fractions of each partition slack in the scheduling slot in a manner that allows serving the pending I/O calls without causing RTOS interference on application processes while also ensuring data freshness to the consumer.

These observations, together with a few others for serving the synchronisation calls too generously allowed for by the ARINC 653 standard, form the basis to our time-composable implementation of an ARINC 653-compliant RTOS which we present in Section 3.4.

3.4 TICOS

The OS we consider is TiCOS, which we developed upon substantial refactoring of POK [147]; POK was initially chosen as part of the PROARTIS execution stack because of its lightweight dimensions, its availability in the open source and its embryonic implementation of the ARINC 653 [40] specification. POK comprised both a common interface to all the functionality offered by the operating system (inclusive of an implementation of the ARINC APEX), and a set of common scheduling primitives. Operating system primitives and ARINC services can be either executed in response to an explicit request from the application code, or implicitly triggered by a timer or scheduling event. We evaluated the services it provided from the standpoint of time composability, that is with respect to the steady timing behaviour and zero disturbance properties. The execution of these kernel services did not meet our requirements set on timing variability and inter-layer interference. The original release provided a set of kernel and ARINC services that were not developed with time composability in mind, as their implementation was clearly oriented towards an optimisation of the average-case performance. In particular, the OS timing profile depended on both the execution history and the actual values given as an input to services invocation; in turn, the execution of a kernel service itself determined new hardware/software states affecting application execution times.

To conform to the actual production environment as much as possible, we focused our study on the PowerPC processor family, which is baseline to lead avionics industry. We re-targeted the kernel to the PPC 750 processor model [136] and completed its partial implementation of the ARINC 653 Application Executive middleware layer. At that point we set on redesigning the kernel primitives and ARINC services that had the highest influence on time composability and analysability from the perspective of the partition-aware scheduling presented in Section 3.3. The ARINC-653 architectural specification allowed us to introduce several simplifying assumptions: the reduction to a strictly periodic task model was probably the most important one. In order to cause no repercussion on legacy avionics software, we made sure that our re-engineered APEX services directly exposed to the application, stayed syntactically equivalent

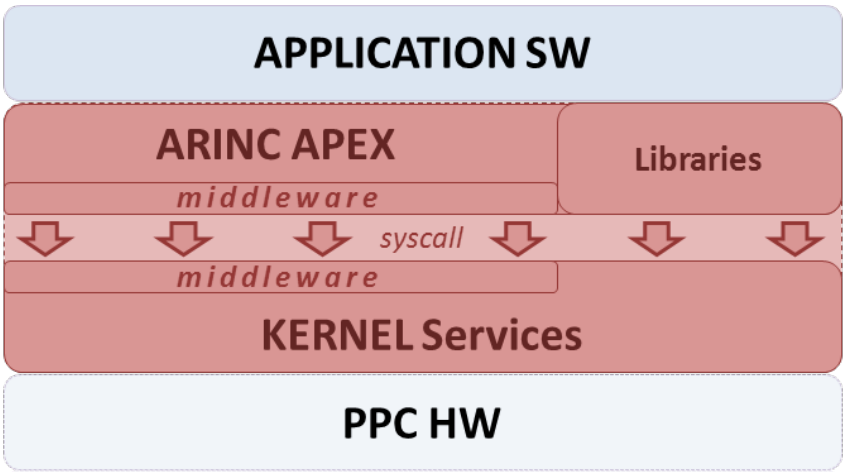


Figure 11.: Structural decomposition of TiCOS.

to and semantically consistent with the standard specification. We finally provided experimental evidence that the degree of time composability may greatly benefit from proper design choices in the implementation of the operating system, without giving up performance.

Figure 11 shows a structural breakdown of TiCOS: the kernel layer provides an interface to a set of standard libraries (e.g., C standard library) and core OS services (e.g., scheduling primitives) to the application SW layer. In addition, TiCOS also provides an implementation of a subset of the ARINC APEX. OS and ARINC services are implemented via a *system call* mechanism (often mediated by a middleware-layer redirection) that performs the switch to the supervisor mode and executes the required TiCOS kernel services and primitives.

Not all OS services, however, are required to exhibit a steady timing nor to affect the user-level code. We focused exclusively on those services that can be invoked as part of the nominal behaviour of the system or the user application. This strategy permits to exclude all those services and primitives that are invoked on system initialisation and start up, before the system enters the nominal operation state. Each kernel service can be classified by the sources of variability it may suffer from and by the degree of history dependence it may carry in the execution stack. Besides the amount of variability, however, we should also consider the way a service is invoked and its frequency of execution. The repeated execution of services on timer expire or an event basis can possibly have even more disturbing effects on the application SW level, especially with respect to how measurements should be collected in practice. The execution

of this kind of kernel primitives (such as, for example, the decrementer interrupt handler) impedes the collection of measurements on an uninterrupted execution of the measured piece of code. As long as we cannot guarantee a fully composable kernel layer this would incur even more history dependence on the execution of the application SW layer.

We continue our discussion by presenting an alternative design and implementation of the OS layer services, aimed at injecting time composability in the kernel, explaining the modifications to time management and system scheduling. We refer the reader to Annex A for the discussion on ARINC library services.

3.4.1 *Time Management*

Time management is one of the core services provided by the operating system, distinct from the front-office work of serving calls issued by the application, as it permits to control the passing of time during execution and to react accordingly. The notion of time is needed by both the operating system itself, to perform back office activities, and the application, which may intentionally program time-triggered actions. Managing task states and scheduling execution are highly dependent on how time is represented in the system: most common time-management approaches adopted in real-time systems rely on either a tick counter or on one-shot timers. When the RTOS design can make no assumption on the application task model, time management knows no better than being tick based. The tick is a constant time span chosen at initialisation by which the processor notifies the RTOS of the elapse of one step in logical time. All timing events that the RTOS can handle are an integral multiple of that step. This is the most general strategy for implementing time management in an RTOS and its requirements on the processor hardware reduce to a periodic timer that fires at constant predefined intervals. When the pairing between the RTOS and the task model is stricter, then one can consider doing away with the tick and use a programmable one-shot timer instead, which allows setting timing events at variable distances, thus being more general than the tick based solution. Unfortunately, not all processors provide one-shot timers, which makes this implementation model less general.

The original POK implementation provided a tick-based time management where a discrete counter is periodically incremented according to a frequency consistent with

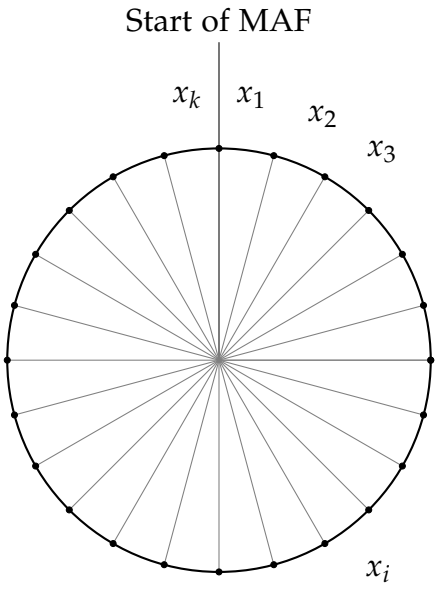


Figure 12.: Tick-based scheduling within a major frame.

the real hardware clock rate[1]. In Figure 12 execution proceeds clockwise within the Major Frame of a partitioned application: the size of every interval x_i corresponds to the time elapsed between two updates of the tick counter, and there are exactly

$$\frac{length\ of\ major\ frame}{tick\ frequency}$$

such intervals in each MAF[2]. Unfortunately, measuring time this way has the disadvantage of causing massive disturbance to executing applications, as time management operations are preemptively performed at every increment of the logical clock, typically served at the highest priority in the system. The negative effects of this RTOS activity on time composability cannot be easily attenuated because the length of the tick interval determines the latency with which the RTOS responds to time-related application needs.

However, we argue that a tick-based solution would still be acceptable if one can guarantee that the tick management (i.e., the set of kernel-level procedures triggered upon each time tick) is both lightweight and constant-time. In terms of the actual TiCOS PowerPC implementation, this means that the decrementer (DEC) interrupt handler, which is used to increment the tick counter, and the triggered

1 TiCOS in its implementation for PowerPC exploits the decrementer register and the TBU to periodically increment the tick counter.

2 A MAF is assumed to be a perfect multiple of the tick frequency.

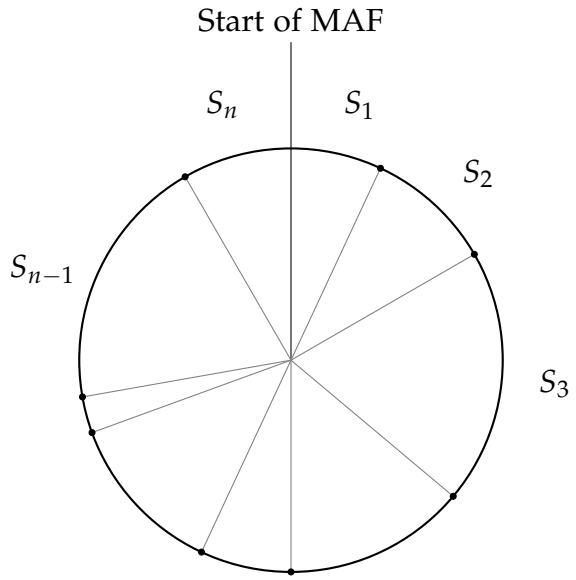


Figure 13.: Timer-based scheduling within a major frame.

scheduling (update) operations should have a minimal time duration (so as to reduce disturbance). Additionally, the behaviour of such interrupt service routines should present no jitter to satisfy the steady timing behaviour requirement in accordance with the implementation of constant-time operating system primitives. To offer a lightweight and steady behaviour we have implemented a constant-time scheduler that is illustrated in the next section. The adoption of a fully discrete tick-based time management together with constant time scheduling primitives would enable constant-time scheduling of user-level applications.

The above solution, however, suffers from severe scalability issues with respect to the required time granularity and, more importantly, prevents achieving zero-disturbance. We therefore investigated and implemented an alternative solution, based on one-shot timers. The other solution, also known as the interval-timer model, has a much milder impact on the application: in a timer-based implementation clock interrupts are not necessarily periodic and can be programmed according to the specific application needs. Intuitively, a timer-based implementation can be designed to incur less interference in the timing behaviour of the user application. The obvious reason for this reduction is that with the interval timer the RTOS may schedule dispatching points instead of polling for them.

If we consider the task model described in Section 3.3, the temporal isolation fences defined by ARINC major frames and scheduling slots naturally form the boundaries of independent, segregated and statically-sized intervals of execution. This arrangement

allows programming the interval timer to fire at the edge of each of the smallest-sized of such frames (the scheduling slot, see Figure 10) and let application execution progress undisturbed within them until the next dispatching point. In Figure 13 a timer-based representation of time is given, where each S_i corresponds to one of the n distinct scheduling slots allocated for partition execution within a MAF which is divided into n slots, regardless of how these are effectively organised into minor frames. Inside each slot alternative techniques must be devised to arbitrate task execution: we adopt a fixed-priority deferred scheduling policy [42][145], in which scheduling points are chosen at design time in a way that minimises disturbance to application code. Bringing this idea to the extreme serves our purpose very well: programming preemption only after the end of a job permits to run it to completion with no interruption from the outer world.

3.4.2 *Scheduling Primitives*

The illustrative schedule shown in Figure 10 is strongly influenced by the nature of control processing, which assumes strictly periodic processes that repeatedly execute a sequence of rigidly sequential functions each of which follows a *read-compute-write* processing pattern. For this very restricted task model and its dominant type of processes, fully preemptive scheduling is plain overkill: in general the set of processes that has to be dispatched within a scheduling slot is defined statically as part of intra-partition scheduling and the period of the corresponding processes is computed backwards as an integral multiple of the minor frame to which the scheduling slot is assigned. On this account, we may safely consider all processes within a scheduling slot to have one and the same release time, which occurs at the edge of the scheduling slot (where a partition switch occurs) and deadlines no later than the end of the scheduling slot. This simplification has no negative repercussion on system feasibility, which continues to be guaranteed exactly as in the original schedule because each process must have been reserved the time to complete its execution within the slot it is assigned to. At the same time we can draw important benefits on time composability. We may in fact contemplate using an extreme form of deferred preemption [145] that allows all processes to run to completion, thus incurring no preemption whatsoever, neither from concurrent processes nor – in force of the solution discussed in section 3.4.1 – from back-office activities of the RTOS. In this degraded form of fixed-priority scheduling within slots, the partition switch that

starts the slot is the only time-programmed scheduling point; all other dispatching points are deferred to process completion events.

Partition scheduling is easily performed in constant time on a circular list (per MAF), as it follows a recurring scheme that resembles cyclic scheduling. Ordered queues are the most common and intuitive way of managing process scheduling instead, including dispatching and status update: the main, yet deadly for us, drawback of this classic solution is that it does not lend itself to constant-time execution of process status updates. For that reason, we redesigned the fixed-priority scheduler using extremely compact representation of task states by bit arrays where the k^{th} bit holds information on the k^{th} process. Bit arrays are updated by fast, fixed-latency bitwise operations; in order to perform scheduling operations in constant time, we also assume that all processes defined in the same partition have distinct priorities. Since hard real-time operating systems typically define 255 distinct priority levels, requiring distinct priorities poses no restriction on ARINC applications, which should be capable of supporting up to 128 processes per partition [40]. It should be noted that *process* is the ARINC equivalent of a *task* in classic real-time theory.

This basic scheduling approach is illustrated in Figure 14 where the circle on the left represents the sequence of scheduling slots within a MAF, which is strictly periodic. A dispatching point (b) is triggered by the transition between two neighbouring slots – in which case it is preceded by a process status update (a) – or a process completion event.

The global data structures needed by our scheduler are the following:

- An array of the processes (say N) defined in the system, ordered by decreasing priority
- The static configuration of the major frame, i.e. the time slots defined and the allocation of partitions to them
- A set of bit masks $MASK^{state}_{current}$ of size N, one for each state a process can assume (i.e., *dormant*, *waiting*, *ready* and *running* in ARINC speak), which collectively describe the current state of all application processes
- A set of bit masks $MASK^{state}_{slot}$ of size N, one for each state a process can assume, to describe how process states are changed in the transition to every slot defined in the major frame.

At every slot (i.e. partition) switch, both scheduling and dispatching are performed:

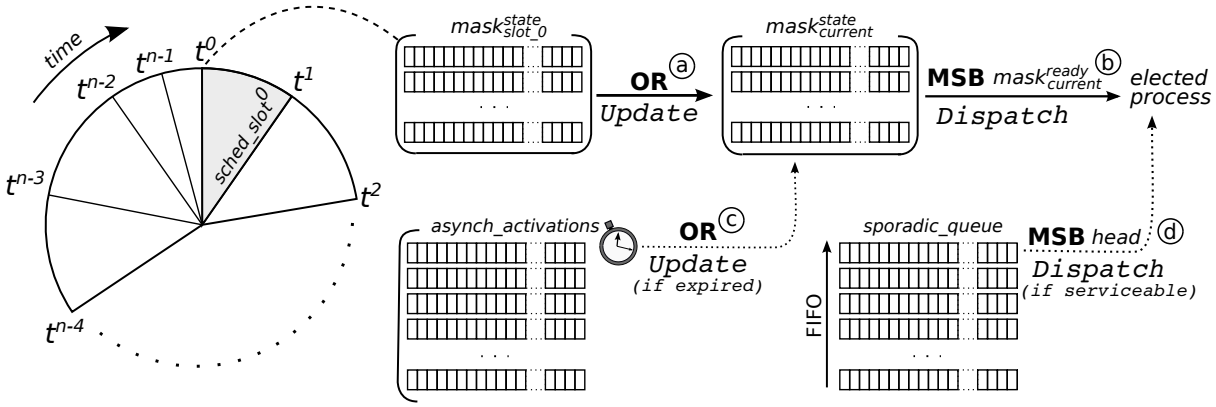


Figure 14.: Core scheduling data structures and operations.

- The status of all processes is updated by using the $MASK^{state}_{slot}$ of the upcoming slot to modify $MASK^{state}_{current}$, such that

$$MASK^{state}_{current}[\tau] = \begin{cases} 1 & \text{process } \tau \text{ should switch to } state \text{ in the next slot} \\ 0 & \text{otherwise} \end{cases}$$

  As Figure 14 (a) shows, updating the status S of the k^{th} process is done in constant time by bit-wise OR-ing the $MASK^{state}_{current}$ with the $1 \ll k$ bit-array (or its dual $\sim(1 \ll k)$.

- As a consequence of the previous step, processes that can be executed in the new slot are characterised by a "1" in the corresponding position of $MASK^{ready}_{current}$: since processes are ordered by decreasing priority, the position of the most significant bit in $MASK^{ready}_{current}$ corresponds to the position in the process array of the next process to be executed (Figure 14 (b)). Identifying such position means calculating the log_2 of the base-10 number represented by $MASK^{ready}_{current}$: this can be done for example by using the *__builtin_clz* GCC built-in function, which counts the leading "0"s in a word. However, since this operation is implemented by using the *cntlz* PowerPC assembly instruction, whose implementation is not guaranteed to take constant-time to execute, we may want to use *De Bruijn sequences*[3] for this purpose [148].

3 De Bruijn sequences allow implementing perfect hashing for binary sequences that are exact powers of two.

The implemented scheduling mechanism is better explained by a simple example. Let us consider a simplified partitioned system consisting of just two partitions P_A and P_B that are assigned two consecutive scheduling slots S_a, S_b. Each partition supports the execution of two user processes, namely τ_{a_1},τ_{a_2},τ_{b_1} and τ_{b_2}, which are all activated simultaneously at the beginning of the respective partition slot[4]. Initially (i.e. at system start-up, prior to the first major frame execution) the system is in the following state:

$$MASK^{ready}_{current} = [\overset{\tau_{b_2}\,\tau_{b_1}\,\tau_{a_2}\,\tau_{a_1}}{0\ 0\ 0\ 0}]$$

which means that no process is *ready* at this point. A state mask is associated to each scheduling slot to configure the state transitions that are to be performed when entering that same slot. Accordingly, the ready state mask associated to slot S_a is thus

$$MASK^{ready}_{S_a} = [\overset{\tau_{b_2}\,\tau_{b_1}\,\tau_{a_2}\,\tau_{a_1}}{0\ 0\ 1\ 1}]$$

since processes τ_{a_1} and τ_{a_1} are expected to become runnable at the beginning of S_a. State masks corresponding to the other process states are conveniently initialised and updated in a similar fashion, and are therefore not considered in this example. When entering slot S_a, process state mask is updated by applying a bitwise OR:

$$MASK^{ready}_{current} \mid = MASK^{ready}_{S_a} = [\overset{\tau_{b_2}\,\tau_{b_1}\,\tau_{a_2}\,\tau_{a_1}}{0\ 0\ 1\ 1}]$$

Since threads are ordered by decreasing priority within a partition and since all threads are assigned distinct priority levels, it is straightforward to determine their execution order. When a thread completes a period of execution, a "0" is placed into its corresponding position in $MASK^{ready}_{current}$.

In this context, a process may switch to a ready state only at the beginning of the current scheduling slot and never in between two scheduling slots. Therefore, according to the Fixed-Priority Preemptive Scheduling (FPPS) policy, all processes in the ready queue are guaranteed by construction to have their jobs executed without being interrupted (as shown in Figure 15 below).

We are aware, however, that this solution conforms to the strictly periodic interpretation of the ARINC task model, i.e. it relies on the strong assumption that all processes are actually activated at the beginning of a partition slot. This assumption is clearly violated when process activations events can be dynamically programmed

4 Processes are labelled by increasing priority order, e.g., $\pi(\tau_{a_1}) < \pi(\tau_{a_2})$.

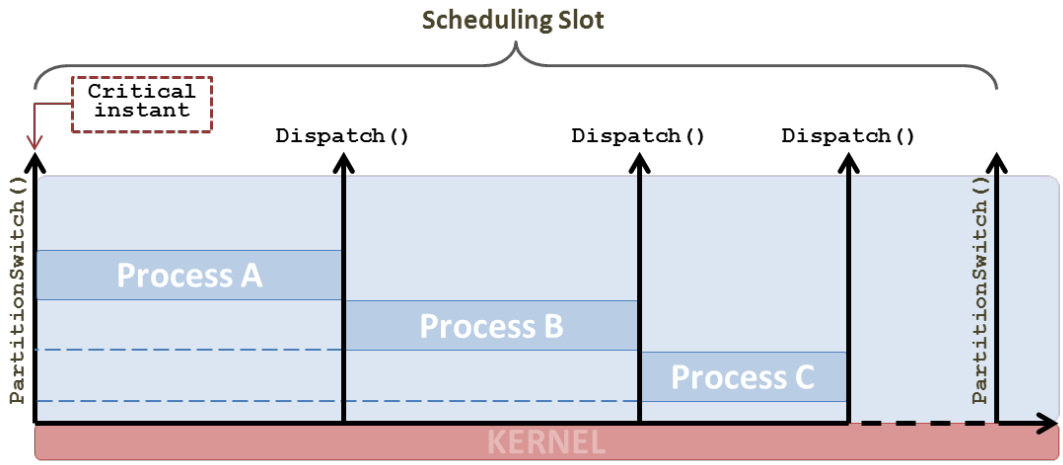


Figure 15.: Execution within a time slot.

by the user application, and thus outside of partition switches (i.e., scheduling slots). This is the case, for example, when a synchronous kernel service requires a scheduling event to be triggered as a consequence of a timeout[5]. This kind of timeout can be used to enforce, for example, a timed self-suspension (i.e., with "delay until" semantics) or a phased execution of a process.

Since we want to ensure that every process is run to completion, preemption is necessarily deferred at the end of process execution, which therefore becomes the next serviceable dispatching point, as shown in Figure 16 (a); dispatching is performed using the same method presented above. The solution we implemented in this case consists in programming our timer to fire at the time this special event occurs, thus in advance with respect to partition switching: the executing application process is interrupted for the amount of time needed to set a "1" in the current $MASK^{state}_{current}$ corresponding to the status to be changed. Timer must be then reset, according to the next event to be served, which could be either another time-triggered event or the concluding partition switch. Since we do not know in principle how many such time-triggered events have to be served within a time slot, we need to maintain them sorted in a queue, whose management cannot be performed in constant time. We can bound the jitter introduced by the use of such a queue by reasoning about the number of timeout events present in the slot: the variable timing introduced by the use of the phased (delayed) execution service can be ignored, since its invocation is supposed

5 The DELAYED_START and TIMED_WAIT ARINC services are representative examples of requests for a timed-out process activation.

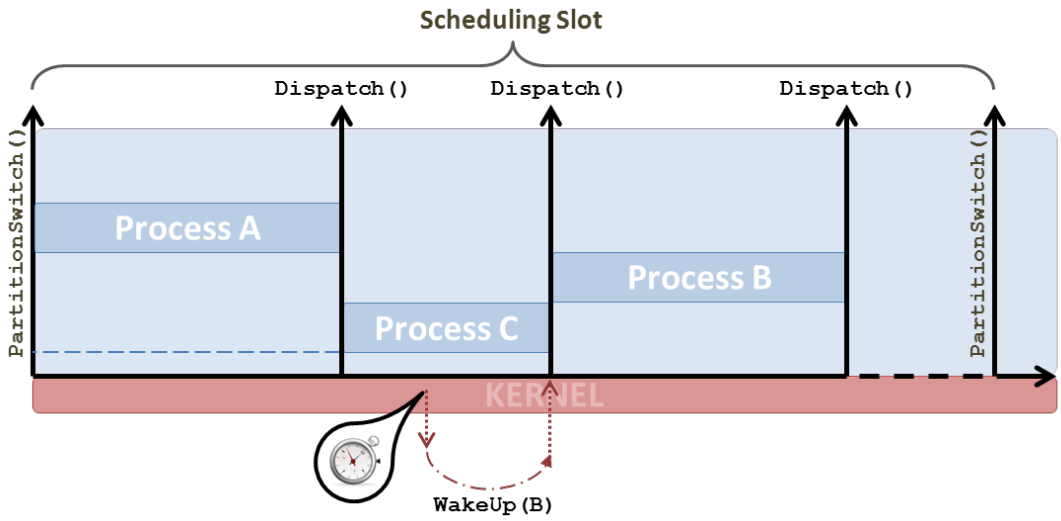
to be limited in the early start up phase of the system. For timed self-suspension, instead, variability can be bounded by looking at the differences between the best- and the worst-case scenarios: the former arises when every timeout is served before the following is set, while the latter case happens when all timeouts are set at the same time. We can conclude therefore that jitter is proportional to the number of timed self-suspension events defined in the slot, which seems to be a reasonable price to pay if having that sort of disturbing events in the system cannot be avoided. A similar mechanism is exploited to manage aperiodic processes (i.e., sporadic tasks): in this case, as reported in Figure 16 (b), the deferred scheduling event is triggered by a synchronous activation request, which does not involve the use of timers.

Before dispatching the next process to execution, the RTOS may need to check the presence of pending asynchronous activation events, whose timer may have expired in the meanwhile and therefore needs servicing. Those dynamically programmed task activations cannot be handled within the same $MASK$ mechanism and require a separate data structure to specifically collect asynchronous events (again represented as bit arrays). That data structure is checked for expired events at every dispatching point. The bit arrays for which a timer has expired are used to update $MASK^{state}_{current}$, as shown in Figure 14(c). Whereas quite good at time composability, this approach may reduce the responsiveness of the system, as it defers the servicing of time events until after completion of the running process. The resulting latency within the scheduling slot of the partition of interest (which is treated as blocking in feasibility analysis) is bounded by the longest process execution in that slot.

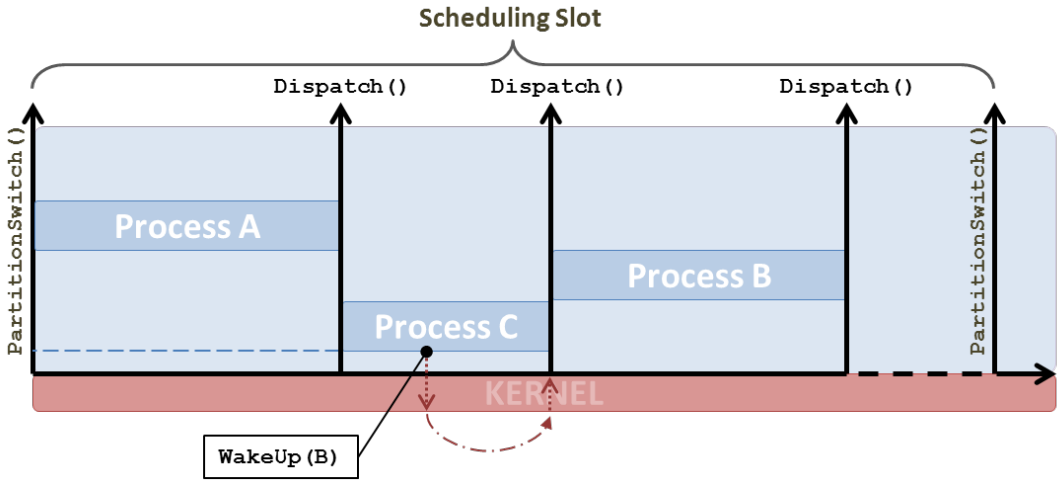
The timing composability property we claimed to subsist on the operating system services presented so far needs to be assessed and corroborated by means of selective experiments, which are reported in the next section.

3.4.3 *Evaluation of the Kernel Layer*

In this section we provide numerical evidence of how time composability has been injected into TiCOS and how end-user applications which leverage its services can benefit thereof. To this end we assessed our redesigned implementation of ARINC-compliant RTOS against the original version of POK, which is simple and perhaps exceedingly simplified, but still representative of an RTOS designed without timing composability in mind.



(a)



(b)

Figure 16.: Deferred dispatching mechanism within a time slot.

Approach and Setup

We evaluated our approach by assessing the redesigned OS services from the twofold dimension of the execution time jitter exhibited by the services themselves and the interference they may incur in the execution time of the user applications. We expected our redesigned services to have no side effect on the timing behaviour of the user application. The only allowed effect would be that resulting from a simple composition of the service timing behaviour with that of the user application. Ideally, given a unit of composition (UoC) f then its execution behaviour on a generic run r should

be computable as the *self*[6] execution time of f plus the execution time of all the OS services S that have been synchronously or asynchronously executed in the same run r:

$$OBSERVED\_TIME_r(f) = SELF\_TIME_r(f) + \sum_{s_i \in S_r} TIME(s_i)$$

This would mean that the observed behaviour of a user UoC is determined by a simple composition of the execution time of the function and the execution time of the OS services that were executing at the same time. The execution time of such services s_i would not depend on the specific run r.

We performed our analysis on the basis of the timing information collected by uninterrupted and consecutive end-to-end runs of software units at the granularity level of kernel primitives, ARINC services and process main procedures. Our experimental evaluation encompasses several target procedures under different inputs or different task workloads. Particularly, experiments were conducted over a relevant set of OS services, which we considered to be the most critical ones from the timing composability standpoint: *time management*, *scheduling primitives*, and *sampling port communication*.

TIME MANAGEMENT. Representation of time evolution and handling of timing events should incur a minimal overhead and possibly no disturbance on the user application. As for our objectives, we wanted to measure whether and to what extent the basic time management primitives may affect the timing behaviour of a generic user application: we evaluated the performance of our time management implementation, based on interval timers, against the standard tick-based approach[7], originally implemented in POK. Under tick-based time management the operations involved are periodically executed, even when there is no need to; this is likely to incur timing interference (i.e. disturbance) on user applications, depending on the tick frequency. By exploiting interval-based timers we guarantee that the execution of a user application is interrupted only when strictly required. Making a step further, it also enables to control and possibly postpone timer expires at desired points in time and possibly in a way such that user applications are not interrupted.

SCHEDULING PRIMITIVES. Implementation and design choices may affect both the latency and jitter incurred by scheduling primitives such as partition switch,

6 The self quota of the overall observed execution is the time spent in executing the UoC itself, as opposed to the cumulative execution time.

7 Tick-based approaches represents a classical option for real-time operating systems.

process dispatching or state update, which incidentally are also strictly coupled with the time management approach. The original implementation of POK came with an incarnation of the standard FPPS policy. The specific implementation, however, exhibited a variable timing behaviour, highly dependent on the task workload. We designed and implemented new kernel data structures to provide extremely fast and constant-time – O(1) – scheduling primitives for both tasks and partitions (e.g., partition switch, thread activation, suspension and selection). This provision, in conjunction with a deferred handling of timing events (preemption included), allows excluding most disturbing effects on the user-level applications. In doing so we collect timing measurements under different task and partition workloads.

ARINC SERVICES. Message-based communication between partitions is undoubtedly one of the most challenging ARINC services from the timing composability standpoint, critically impacting on the application. As observed in Section 3.4, both intra- and inter-partition communications suffer from considerable variability as their execution varies on the amount of exchanged data (i.e., message size). Moreover, the execution of such primitives as part of an OS layer service may have indirect disturbing effects on the execution context of the user application. By relocating the execution of read and write services at the beginning and end of a partition scheduling slot respectively we avoid any variable overhead on the user application. We can also benefit from the cache in executing such loop-intensive services as we have no constraint on preserving the hardware state at partition switch.

The properties we want to prove on the OS layer (steady timing behaviour and zero-disturbance) do not need to be studied with probabilistic techniques over a large volume of observations. Arguably instead, those properties can be assessed just by measuring a small number of selective examples. This claim is further corroborated by the fact that the PROARTIS Sim tool[8], which was used to collect timing traces, complies with the hardware features required by PTA techniques described in Section 2.5.2. In particular, this implies that all hardware instructions are guaranteed to execute with a fixed latency, except for memory accesses whose latency depends on the current cache state. Since caches are the only residual source of history dependence we were able to exclude, when needed, any source of interference in the execution time by simply enforcing a constant response of the cache, either always miss (i.e., inhibition) or always hit (i.e., perfect cache). For example, the interrupt handling mechanism

8 PROARTIS Sim is a highly-configurable SocLib-based PowerPC 750 simulation platform [16], graciously supplied by the PROARTIS project team.

has been modified to automatically disable caches on interrupt, thus enabling a sharp approach to zero-disturbance. This allowed us to restrain our observations to a limited number of runs. Moreover, thanks to the characteristics of the execution platform, it is not rare for the concocted test cases to exhibit a strictly deterministic execution runs that follow a periodic pattern, which allowed us to further reduce the number of required observations for scheduling primitives.

Although we identified cache inhibition to be the main countermeasure to potential disturbance[9] from the execution of kernel services, we also highlighted how several of those services can actually be postponed and executed outside the execution of the user application. The baseline PROARTIS Sim configuration in our experiments includes the perfect cache option, which corresponds to enforcing the latency of a cache hit on every memory access. In the absence of a fine-grained control over the cache behaviour, this parametrisation is meant to exclude the variability stemming from caches without incurring the performance penalty of thoroughly disabling them. According to our overall approach, in fact, the majority of our experiments addresses those services that are executed outside of the user application and there is no need to execute them with acceleration features disabled. The deferred preemption and the redesigned sampling port services perfectly fit this scenario.

Some performance loss is inevitably incurred by those services that do not lend themselves to be deferred: the enforcement of the zero-disturbance property will prevent them from taking benefit from history-dependent hardware acceleration features. However, if we exclude those ARINC services that are executed at system start-up (i.e., they are not part of the nominal behaviour) and inter- and intra- partition communication services (as we explicitly handle them), we do not expect the kernel services to greatly benefit from caches. Although the time penalty incurred by those services should be in principle assessed and evaluated against the derived benefits, we expect it not to overly penalise the overall execution of a system.

The execution traces obtained through the simulator were then interpreted on Rapitime [26], a hybrid measurement-based timing analysis tool from Rapita Systems Ltd. The information-rich tracing capabilities of PROARTIS Sim allowed us to collect execution traces without resorting to software instrumentation, thus avoiding our experiments to incur the so-called *probe effect*.

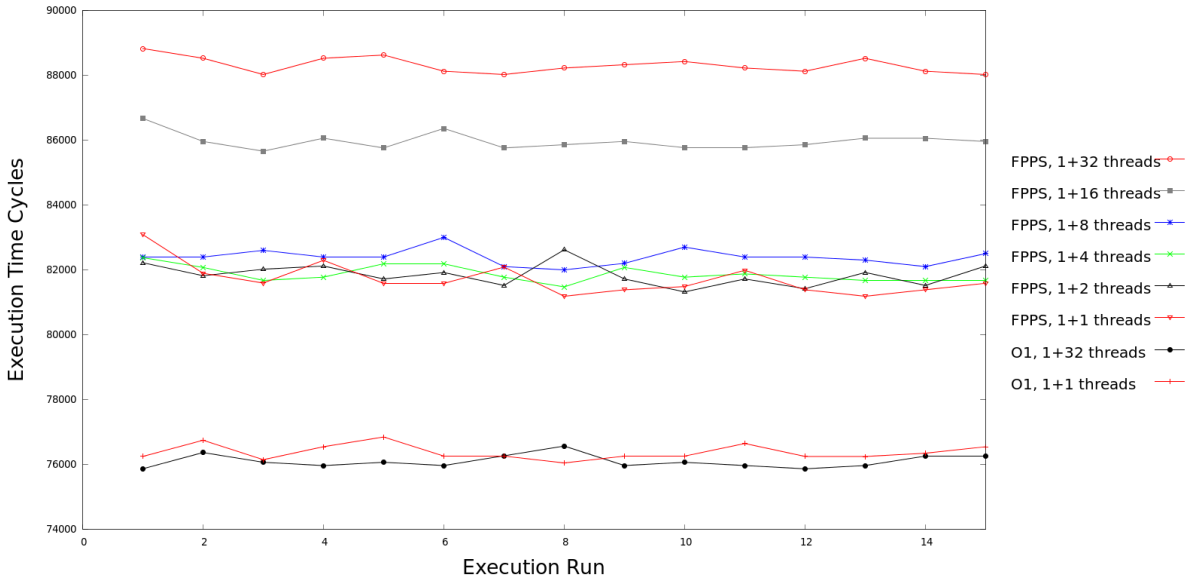


Figure 17.: Tick counter interference on the application under different workloads and scheduling policies, Fixed-Priority Preemptive Scheduling (FPPS) and constant-time (O1).

Results

As a starting point, we first wanted to evaluate and compare the interference experienced by user applications because of the different time management approach. To this end we selected the "statemate" routine from the Mälardalen benchmark [149] as our sample application and we measured it under different time management configurations and different system workloads. The main procedure of the analysed process was designed to periodically execute a loop intensive routine, as loop intensiveness is intended as a means to make the application sensible to the cache behaviour; caches have been enabled and configured with Least Recently Used (LRU) replacement policy. We first measured the example application within an implementation which uses the decrementer register as a tick counter. Successively, we set up a new scenario where no interference from the time management service is experienced, as the latter is implemented by interval timers set to fire outside the execution boundaries of the examined procedure. We observed the timing behaviour of our target application over several end-to-end runs under both settings, expecting to notice and quantify the difference between them. The system workload instead has been configured to range from a minimum of a couple of processes to a maximum of 32 processes competing with our target benchmark.

9 As caches are the only history dependent hardware feature in our execution platform.

The experimental results in Figure 17 show that an interval-timer policy (lower lines in figure) always outperforms a tick-based one, since the latter introduces significant interference on the application code, as a consequence of the time-management activities performed on every tick. Moreover, even when considering the same policy in isolation, the tick-based time management presents larger fluctuations depending on the presence of other processes in the system (i.e the pending workload); in fact, the presence of caches is responsible for amplifying the jitter suffered by the analysed process in both settings. An application scheduled with an appropriate interval-timer is instead less prone to polluting cache effects and presents therefore smaller variations in observed execution times. Interestingly, as highlighted by the relative gap between extreme lines, the cache-induced variability experienced by the application under a tick-counter policy is relatively higher than that suffered under interval-based timer; this is explained by the fact that the user application is disturbed by the tick counting system activity, which typically consists in updating the OS software state and is likely to negatively affect the cache state.

Considering the scheduling policy itself, our interest falls on the kernel primitives implemented to perform the status update of the active processes and the election of the candidate that will be actually dispatched next. We compare two scheduling policies by measuring execution times of their routines: in the FPPS setting we measure the scheduling routine end-to-end, since thread status update and selection are always performed together as an atomic operation, at every clock tick. In the constant time scheduler case instead we track the execution times of the two routines separately, since thread status updates can only occur at partition switch, whereas process election for dispatching needs to be performed every once a process has completed its execution, which is guaranteed to run to completion with no external interruption. The results prove and measure the jitter present in the original FPPS scheduler, while our new scheduler is shown to execute in constant-time, thanks to the smart representation of task and tasks states through bit masks and the efficient register operations over them. In the experiments we enforced a perfect cache behaviour so that no overhead from the hardware is accounted for in measured execution times. Moreover, exploiting the features of our execution platform and the adoption of the FPPS policy with distinct priorities, we designed our experiments to include only periodic tasks. This guarantees the execution of test cases to follow a strictly deterministic periodic pattern, which allowed us to limit our observations to a small number of runs. We first focused on the timing variability that stems from the number of run-time entities and the way

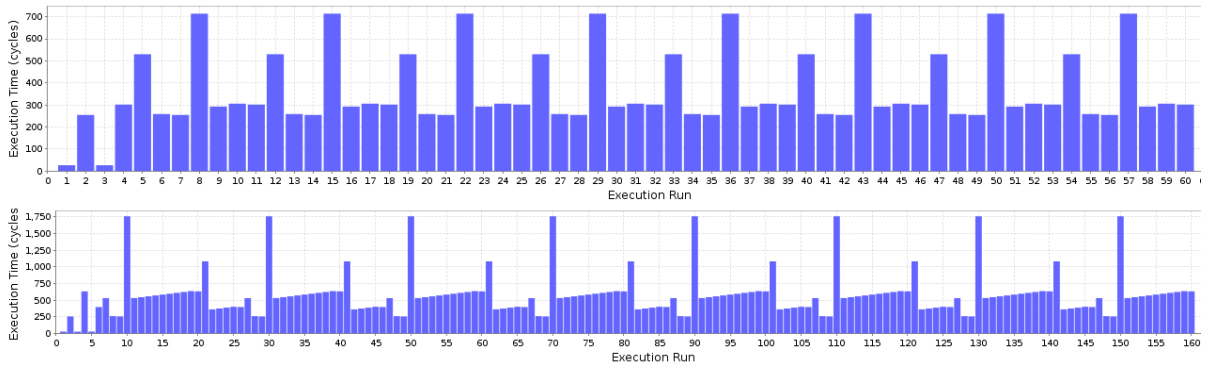


Figure 18.: FPPS thread selection under different workloads.

these are organised and managed. Figure 18 shows the observed execution times for the fixed-priority scheduler: more precisely, the values reported herein refer to the thread selection routine that is part of the larger scheduling primitive. The setting in the top chart is two partitions, with three and two threads respectively, while the bottom chart reports a larger example comprising three partitions with ten, five and two threads each.

The FPPS scheduler always performs the same operations at every clock tick, mainly checking whether the current partition and thread should be switched to the next ones; there are two sources of variability in here given by the possibility that a partition/thread actually needs to be switched at a particular scheduling point, and by the number of threads to be managed in the executing partition, respectively. The graphs above illustrate very well this situation: higher peaks correspond to partition switches, when the state of all threads in the new partition changes to ready and they must be therefore inserted in the appropriate scheduler queues. Those peaks have different heights in accordance to the number of threads defined in each such partitions. Values following the higher peaks correspond to thread switches within the same partition: here execution time is determined by the operations needed to find the next thread to be dispatched, which depends once more on the number of threads in the partition. Additionally, the number such operations is slightly increasing since ready queues are progressively emptied and their exploration requires more time[10].

For the constant-time scheduler we implemented we must distinguish two cases, since its behaviour is different at partition and thread switch. Figure 19 shows the

10 Even though this step could be performed in constant time, the reported graphs suggest that the gain on the average case would be negligible and still dominated by the worst-case time experienced at partition switch.

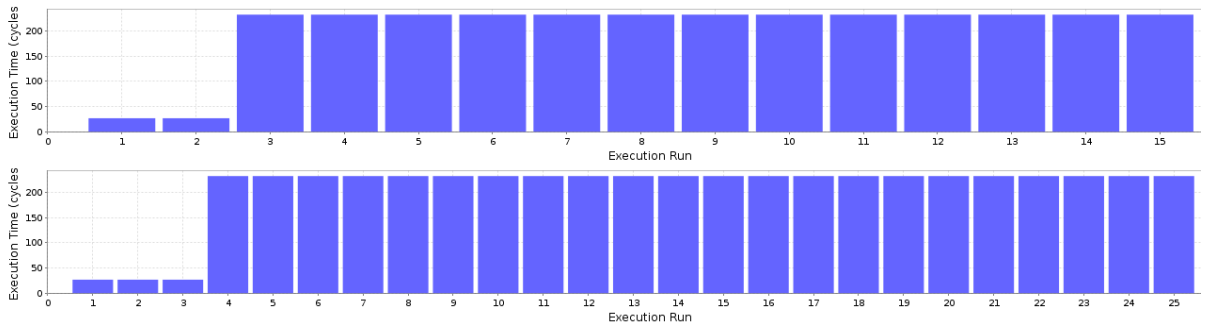


Figure 19.: Constant-time thread status update under different workloads.

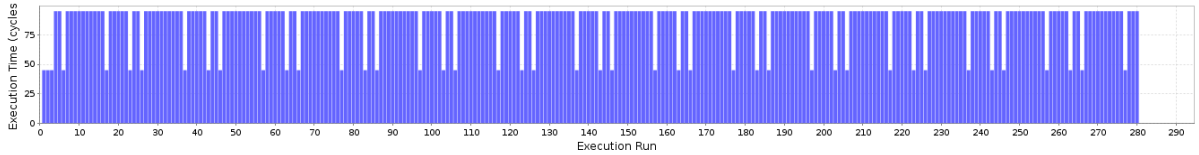


Figure 20.: Constant-time thread selection in a test case with three partitions and seventeen threads.

execution time of the routine invoked at partition switch, which only needs to update thread states[11]. Though the settings are exactly the same as Figure 18 above, status updates are performed in constant time thanks to the bitwise operations on thread masks, as explained in Section 3.4.2.

Actual thread selection is performed at every thread switch: Figure 20 shows that our constant-time scheduler is capable of detecting the highest-priority thread to be dispatched with fixed overhead regardless of the system workload, by using De Bruijn sequences as explained in Section 3.4.2. For the same reason, our constant-time scheduler performs better than FPPS even when the cumulative execution cycles and jitter of the two routines are considered. In Figure 20 lower peaks correspond to the selection of the system idle thread. The raw numbers for the illustrated experiments are reported in Table 1 below. It is worth noting that the small delta exhibited by our thread switch implementation is actually due to the difference between the selection of any thread (95) and the idle thread (45). The delta measured on the FPPS implementation instead represents real jitter.

11 This is no longer true when the system includes communication ports, as we will discuss in the next section.

Table 1.: Execution times for a user application with tick-based and interval-timer scheduling.

	FPPS (standard POK)			O(1) scheduler (partition switch)			O(1) scheduler (thread switch)		
	Min	Max	Delta	Min	Max	Delta	Min	Max	Delta
2 partitions 5 threads	255	714	459	232	232	0	N/A		
3 partitions 17 threads	259	1759	1500	232	232	0	45	95	50

We presented in Annex A the implementation of a new communication mechanism for sampling ports[12] based on posted writes and prefetched reads that permits to remove the sources of variability and disturbance from the service itself and serve them out at partition switch.

In order to assess our approach, we focused on the READ_SAMPLING_MESSAGE and WRITE_SAMPLING_MESSAGE ARINC services and compared our novel implementation against the standard one. We performed such a comparison on the maximum observed execution times, collected by triggering the execution of the above services under different inputs. As observed in Section 3.4.3, we forced the simulator to resemble a perfect cache when measuring this services as we wanted to exclude the variability stemming from the cache behaviour without incurring the performance penalty of a disabled cache: a positive effect of relocating the message management in between the execution of two partitions is in fact that of being able to exploit the caches without any side-effect on the user application.

Having excluded the cache variability, the analysed service exhibits a steady timing (actually constant) behaviour where the execution time only varies as a function over the input (message) size. We triggered the execution of the sampling services with different sizes of the data to be exchanged. Figure 21 below provides an overview of the execution times we observed in our experiments for the standard implementation (on the left) and our respective variant (on the right). The rightmost columns in each chart in Figure 21 instead show the variation in the execution time required to perform a partition switch, which is the penalty that has to be paid for relocating the message passing mechanism. As expected, the new implementation of the READ and WRITE services exhibits a major reduction in their execution times with respect to their standard counterparts. In addition, we observe that our expectations on steady behaviour are perfectly met for both services as they run in constant time

12 Queuing ports services exploit exactly the same mechanism.

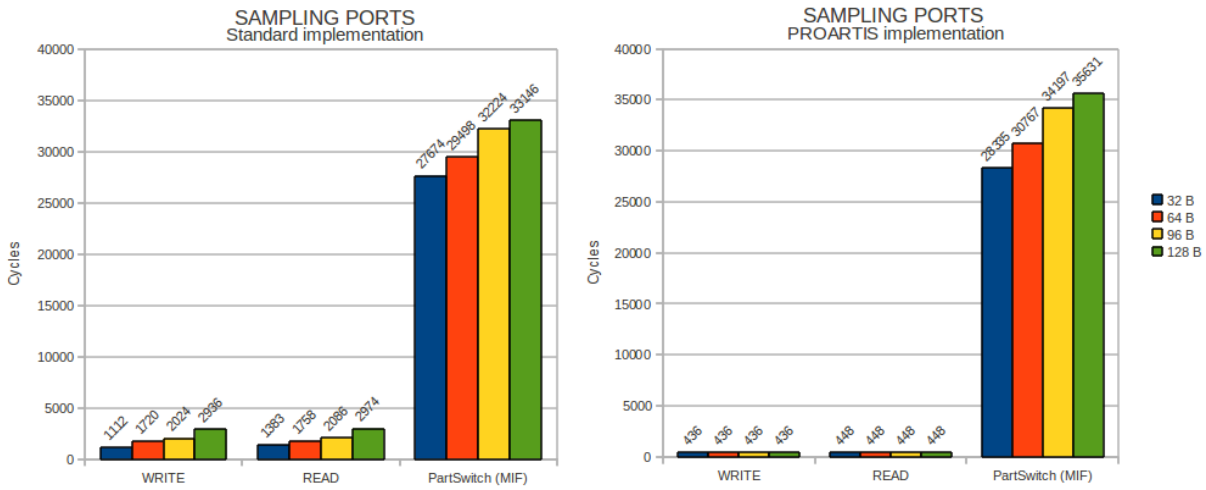


Figure 21.: Overall comparison of the two implementation variants.

regardless of input data. The reduction and stabilisation in the execution time of both services is easily explained by the fact that their most time-consuming part has been removed and postponed at partition switch. The standard implementation of the WRITE_SAMPLING_MESSAGE, for example, included the loop-intensive copy of the raw data in input to the specified source port. This code pattern inevitably exhibits a timing behaviour that varies on the amount of data that has to be written to a port location. The READ_SAMPLING_MESSAGE shares a similar but specular structure where the message data has to be retrieved from a port location.

As a further source of timing variability, the amount of data that is read (written) from (to) one and the same port is not guaranteed to be statically fixed[13]. The new mechanism based on posted writes and prefetched reads, instead, enforces a steady timing behaviour, regardless of the input data. The dependence of READ and WRITE services on the input size is shown in Figure 22 where the increase in the execution time of each service is related to an increase in the input data size, here ranging from 1 to 256 Bytes as reported in Table 2.

Our implementation of the sampling I/O services is based on the decision of postponing the actual read and write activities at partition switch. All valid reads required by a partition will cause the prefetching of a message in the partition memory space before the partition itself starts executing; similarly, all writes commanded by a partition will be performed before executing the successive partition. Thus, the kernel primitives involved in partition switching have to execute those offline

13 Only the maximum message size is always defined for sampling and queuing ports.

Table 2.: Execution times for the READ and WRITE services.

	WRITE	NewWRITE	READ	NewREAD
1B	523	436	618	448
4B	580	436	794	448
32B	1112	436	1383	448
64B	1720	436	1758	448
96B	2024	436	2086	448
128B	2936	436	2974	448
256B	5368	436	5406	448

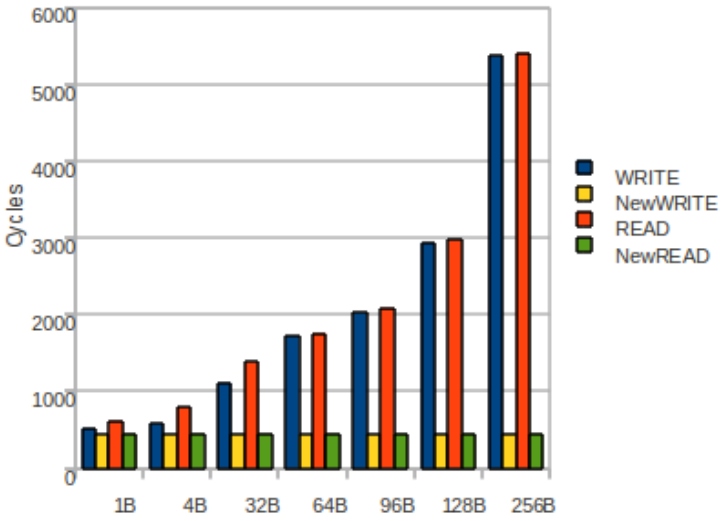


Figure 22.: Steady vs. input-dependent timing behaviour.

Table 3.: Maximum observed partition switch overhead.

	32 B	64 B	96 B	128 B	192 B	256 B	384 B
Partition Switch (standard)	27674	29498	32224	33146	37686	41619	48630
Read+Write Overhead	+ 661	+ 1269	+ 1973	+ 2485	+ 3807	+ 5118	+ 7455

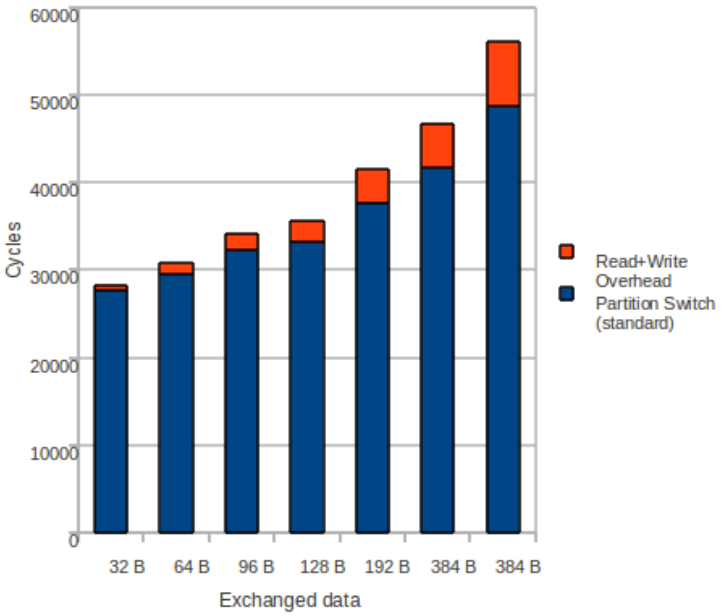


Figure 23.: Steady vs. input-dependent timing behaviour.

activities on behalf of the partitions themselves. We do not require the OS layer to be not-disturbing when executing between partitions as no expectation is made on the hardware state upon partition switch. Nonetheless, the partition switch primitives still have to exhibit a boundable timing behaviour and should not excessively suffer from the new communication-oriented activities. In our experiments we defined a small system consisting of only two communicating partitions as we just wanted to draw out some figures on the relationship between the size of the exchanged data and the increase in the partition switch time. Figure 23 and Table 3 show the partition switch overhead we observed under different input sizes. From what we observed in our experiments, the incurred time penalty is quite limited and, more importantly, when summed to the time previously spent in executing the READ or WRITE service, it does not exceed the execution time of the standard implementation (with the same input).

Finally, we wanted to measure the overhead that we had to pay in return for introducing sporadic processes in the task model. We knew from the beginning that, as a consequence of extending the task model, the complexity of the scheduling mechanism would increase. In our experiments we focused on WAIT_EVENT and SET_EVENT, the two main synchronisation services (for listening and signaling an event) that we reimplemented for the servicing of sporadic activations in accordance with the process split pattern described in Annex A. Both services have been measured

	SET_EVENT			WAIT_EVENT (end-to-end run)		
	Min	Max	δ	Min	Max	δ
ARINC-compliant	5791	9370	3579	8472	9053	581
Process Split pattern	7895	8300	405	42299	42319	20

Table 4.: Sporadic overhead.

under different workloads, ranging from 5 to 20 processes. As reported in Table 4, our SET_EVENT implementation exhibits a notably less variable timing behaviour compared to the original implementation, thanks to the use of constant-time data structures (still based on bit arrays). The residual variability is due to the cache behaviour as the cache is enabled in the part of the API that is exposed at the application level. Table 4 also reports the results observed on an end-to-end run of a process calling the WAIT_EVENT service, intended to measure the overhead introduced by our solution when the call is actually non-blocking (i.e., the guard is open). In this case, we observed a considerable execution time overhead when used in conjunction with the process split pattern. The difference is explained by the fact that the service now always generates a dispatching call.

3.4.4 *Avionics Case Study*[14]

Examining the timing behaviour of a time composable kernel in isolation or on an ad-hoc case study may be a valuable exercise; however, our effort reveals to be more significant in a more realistic scenario, where the OS manages HW resources and interacts with an application. In this respect, we had the opportunity to test our refactored kernel to support the execution of a real-world application run by Airbus in the context of the PROARTIS project. The application test bed is designed according to the IMA architecture illustrated in Section 3.3. This case study is interesting not only to show that our time composable kernel smoothly enables timing analysis, but also to give an example of how the Measurement-Based approach to Probabilistic Timing Analysis (MBPTA) works.

14 Contributions to this section are taken from PROARTIS publication at SIES 2013, jointly authored by the PROARTIS Consortium.

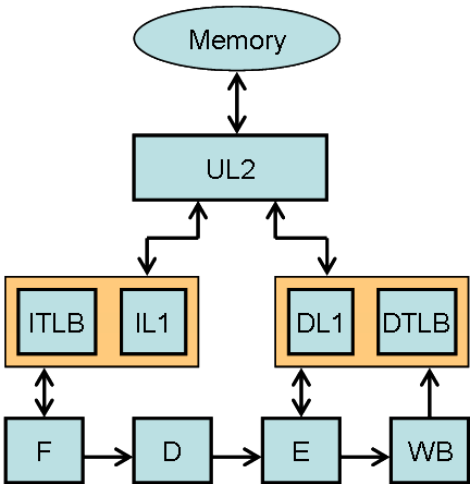


Figure 24.: Reference processor architecture

Approach and Setup

The processor architecture considered in this case study fulfils the PTA requirements which we illustrated in Section 2.5.1 and avoids timing anomalies by construction [150]. In particular, the PROARTIS processor features a 4-stage in-order pipelined core architecture with separate TLB and two levels of caches for instructions and data (see Figure 24). In-order instructions issue and finalisation avoids using resources out-of-order in the final stage of the pipeline, which could cause timing anomalies otherwise.

The cache system is composed of two separate 8-way set-associative L1 caches for instruction and data, of 32 KB in size and 32-byte cache line each, with random placement and replacement policies, operating in write-through mode [151]; a unified L2 8-ways set-associative cache of 64 KB in size and 32-byte cache line each operating in copy-back mode with random placement and replacement policies is also provided. 4 KB pages memory management is performed by two 2-way set-associative TLB (separate for instruction and data) with 128 entries each, also with random placement and replacement. TLB and caches are accessed in parallel so that instructions are stalled in the corresponding stage until both cache and TLB can serve the request. The use of random placement and replacement policies in caches and TLB attaches a distinct probability to hits and misses [27]. That is, the latency of the fetch stage depends on whether the access hits or misses in the instruction caches and TLB: for instance, a cache and TLB hit has a 1-cycle latency, whereas TLB and cache misses have a 100-cycles latency. After the decode stage, memory operations access the data cache and TLB analogously to the fetch stage. Overall, the possible latencies of a

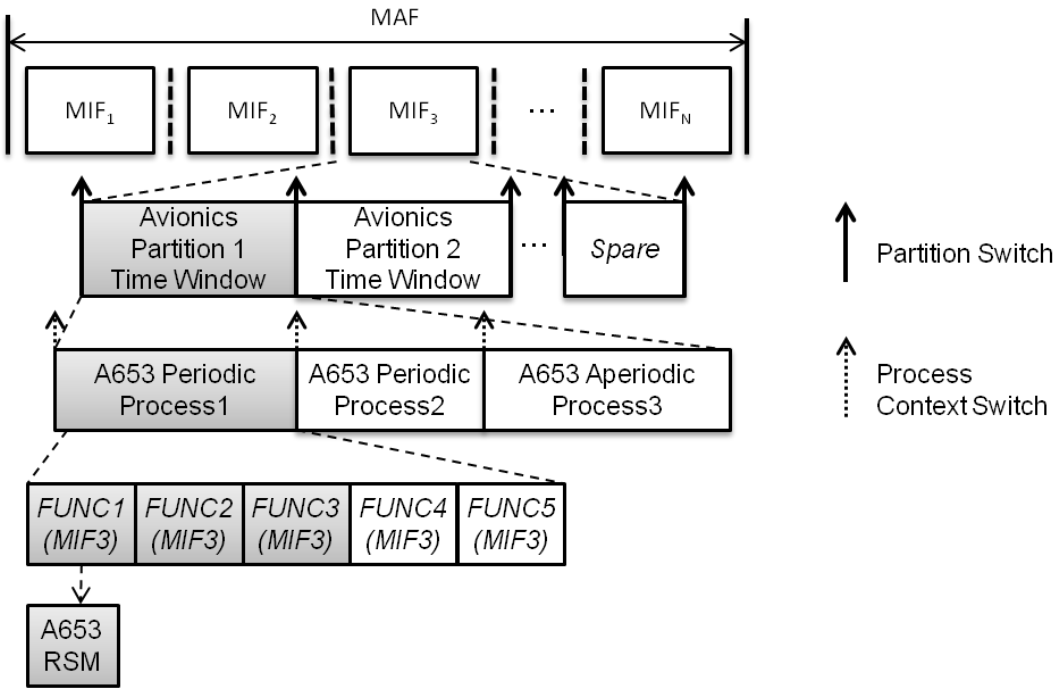


Figure 25.: Functional blocks and A653 services executed within a MIF and called by an A653 Periodic Process. Grey blocks denote the analysed programs.

non-memory instruction depend on whether it hits or not in the instruction cache and TLB, and the particular (fixed) execution latency of the operation (e.g. integer additions take 1 cycle). The latency for memory instructions depends on the hit/miss behaviour in both instruction and data caches and TLB. As a result, the observed execution times meet MBPTA requirements by construction. It is important to remark that the MBPTA approach cannot be applied to a standard deterministic architecture since the frequencies of the execution times observed during testing do not represent actual probabilities [29]. This is so because the events that affect execution time are not randomised. Hence, no probabilistic guarantees can be determined from the observed behaviour for the probability of occurrence of any given execution time in the future.

The avionics application considered in the case study executes with a period of P ms over N consecutive MIF. As illustrated in Figure 25, the application activity consists of five functional blocks, each of which is a high-level procedure, root of a finite and acyclic call graph which changes at the boundaries of every MIF:

- Functional blocks *FUNC1 (Read Sampling Port Normal, or RSPN in brief)* acquire raw data from the I/O ports. Data are acquired through the *A653 READ_SAMPLING_MESSAGE* API service.

- Functional blocks *FUNC2 (Extract Local Name Normal, or ELNN in brief)* de-serialise raw data. Raw payload data are parsed and allocated to global variables used in the subsequent steps.

- Functional blocks *FUNC3 (Normal Scade, or NS in brief)* perform data processing. These functions are automatically generated from a SCADE [152] model.

- Functional blocks *FUNC4 (Format Local Name Normal, or FLNN in brief)* serialise the processed data.

- Functional blocks *FUNC5 (WSPN, or Write Sampling Port Normal)* post the output data to external physical I/O ports.

The real application is hosted on a computer board embedding a MPC 755 processor, with L1 data cache operating in copy-back mode. Experts cannot apply state-of-the-art STA to it within acceptable bounds of engineering effort and containment of pessimism. The major difficulties are caused by the caches operating in copy-back mode, and by the massive presence of interfering Operating System code in the application space – typical for ARINC 653 implementations –, which cannot be sanely analysed by application-level state-of-the-art STA tools. The results presented thereafter were obtained by running the application on the PTA-friendly time-accurate simulator presented above. The application was run on top of our ARINC 653 compliant TiCOS that provided zero-disturbance constant-time services, which proved to be a good facilitator to application-level timing analysis.

Results

We first focus on the OS: Figure 26 shows the periodic execution pattern exhibited by the *READ_SAMPLING_MESSAGE* service called by FUNC1. As expected, the routine manifests a nearly-constant time behaviour for all invocations performed during partition execution (lower bars). Higher peaks correspond instead to invocations occurring at partition switch, as a consequence of the cache flushing activities programmed at this point and of input data pre-fetching, as needed by the next executing partition. This makes it possible to define the probabilistic WCET distribution function for the *READ_SAMPLING_MESSAGE* service shown in Figure 27, which enables probabilistic timing analysis of the avionics application described before.

We can therefore study FUNC1, FUNC2, and FUNC3 and Partition Switch time, when operating system activity deferred during the execution of the application is

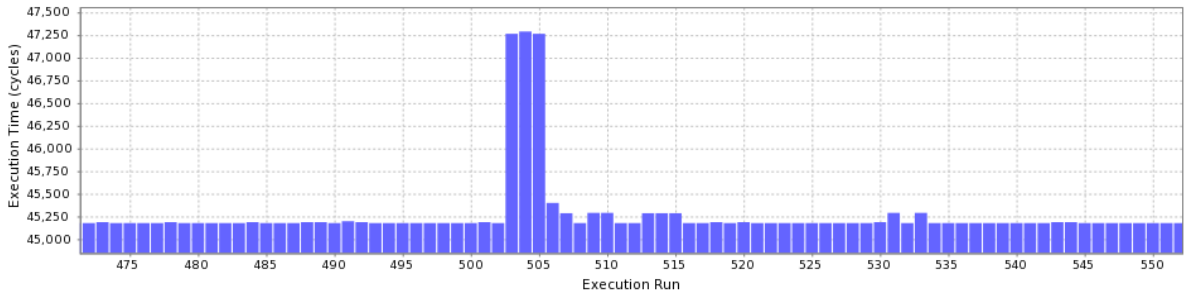


Figure 26.: Observed execution times for the *READ_SAMPLING_MESSAGE* TiCOS service.

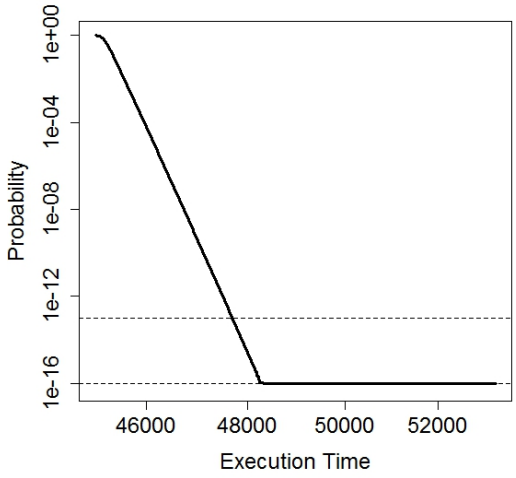


Figure 27.: pWCET distribution for the *READ_SAMPLING_MESSAGE* TiCOS service.

finally performed[15]. The five steps of the MBPTA technique depicted in Figure 6 of Section 2.5.2 are followed.

- **Collecting observations.** In our experiments we started collecting 100 execution time observations, with $N_{delta} = 50$[16]. At every collection round, we checked whether the data are described by i.i.d random variables: the two-sample Kolmogorov-Smirnov (KS) [153] test evaluates the fulfilment of the identical distribution property, and the runs-test [143] the fulfilment of the independence property. These verification steps are mandatory for MBPTA with EVT. Table 5 shows the results of the i.i.d. tests for all programs under analysis, for

15 The processing in FUNC4 and FUNC5 is analogous to that in FUNC2 and FUNC1 respectively and so is their timing behaviour.

16 N_{delta} is the number of observations fed to the next steps in case more observations were required to be able to compute the pWCET distribution

Table 5.: p-value for the identical distribution (considering two samples of $m = 50$ and $m = 100$ elements) and independence test.

	Identical Distr.		Indep	Passed?
	m=50	m=100	p	
FUNC1	0.77	0.89	0.85	Ok
FUNC2	0.69	0.90	0.08	Ok
FUNC3	0.52	0.68	0.68	Ok
PS	0.11	0.89	0.07	Ok

Table 6.: Minimum number of runs (MNR) per program.

Program	FUNC1	FUNC2	FUNC3	PS
MNR	600	250	300	250

the set of observation runs sufficient to derive the pWCET distribution (see Table 6). In order to apply the KS test, two smaller samples of $m = 50$ and $m = 100$ elements were created by randomly taking sequences of m consecutive elements from the original sample [29]. The the two-sample KS test were then applied to the two smaller samples to prove that the distribution does not change over time. The fact that in all cases the p-value obtained from the data for the KS test is higher than the required threshold indicates that data are identically distributed. All the data collected from the observation runs also pass the independence test.

- **Grouping.** A block size of 50 is used in this case as it provides a good balance between accuracy and number of executions required by the method.

- **Fitting.** EVT requires that two hypotheses are verified: (1) the n random variables are independent and identically distributed (which we checked in the *Collecting* step) and (2) the maximum of the n random variables converges to one of the three possible EVT distributions: Gumbel, Frechet or Weibull. The Gumbel distribution, with shape parameter $\xi = 0$, has been proven to fit well the problem of WCET estimation [29][154]. In our application of EVT we use the exponential tail (ET) test [155] to validate that the distribution fits a Gumbel distribution, as shown in table 7, and to determine the σ and the μ that characterise the specific Gumbel distribution.

- **Convergence.** In the experiment a difference threshold of 0.001 is considered as shown in Figure 28. If the distribution had not yet converged instead, the process

Table 7.: ET test results. *Inf Conf. Interval* and *Sup Conf. Interval* stand for inferior and superior confidence interval bounds respectively.

	Estimated q	Inf Conf. Interval	Sup Conf. Interval	passed?
FUNC1	2732099.95	2731953.71	2732199.27	Ok
FUNC2	1290199.02	1290198.36	1290441.20	Ok
FUNC3	1414051.63	1410751.90	1414588.60	Ok
PS	1110203.18	1109958.36	1110642.24	Ok

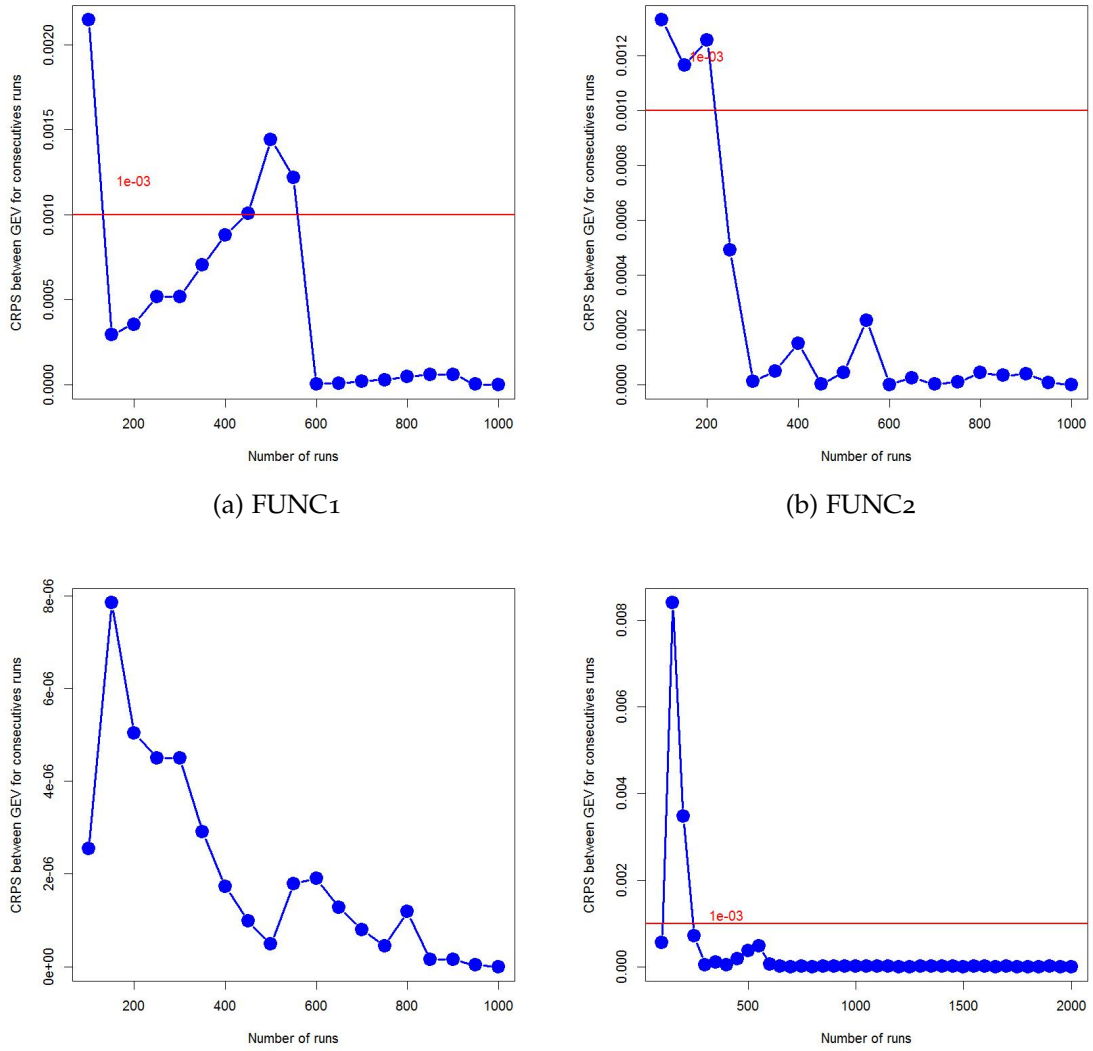
would return to the first step and collect N_{delta} more observations. Notably, passing the ET test ensures that the method always converges [29].

- **Tail extension.** Figure 29 shows the pWCET estimates obtained for the FUNC1 (a), FUNC2 (b), FUNC3 (c) and Partition Switch (d) procedures. As explained in [29], for multi-path programs (as FUNC3 in our experiments) the pWCET obtained from the MBPTA-EVT method only holds for the set of paths actually traversed in the measurement runs[17]. The path coverage we obtained for FUNC3 was deemed sufficient by the scrutiny of our industrial experts.

As MBPTA can only be meaningfully applied to programs running on a PTA-friendly architecture, the case-study application has been run on a processor simulator that supports the required features. At that point, the only practical solution for a cycle-true accurate comparison was to set the simulator cache to operate with modulo placement and least recently used (LRU) replacement, and to use the common industrial practice of taking the highest measurement for each program of interest and adding an engineering margin to it as determined by in-house knowledge on the system. The bounds obtained by running the same experiments on the simulator with caches configured in deterministic LRU mode are shown as dashed vertical lines in Figure 29. pWCET estimates are significantly below a traditional *LRU+margin%* WCET estimate, where the additional margin determined by engineering judgement is usually chosen around 20%. As shown in Figure 29, pWCET estimates are slightly above LRU execution times (between 0.1% and 3.8%), so they are between 16% and 20% below the WCET estimates for *LRU+20%*.

Currently, for commercial airborne systems at the highest integrity level (DAL-A), the maximum allowed failure rate in a system component is 10^{-9} per hour of operation

17 Note that, as opposed to MBTA methods on top of deterministic architectures, MBTPA on top of PTA-friendly architectures has *no dependence* on the particular mapping of data and instructions in memory.



(a) FUNC1

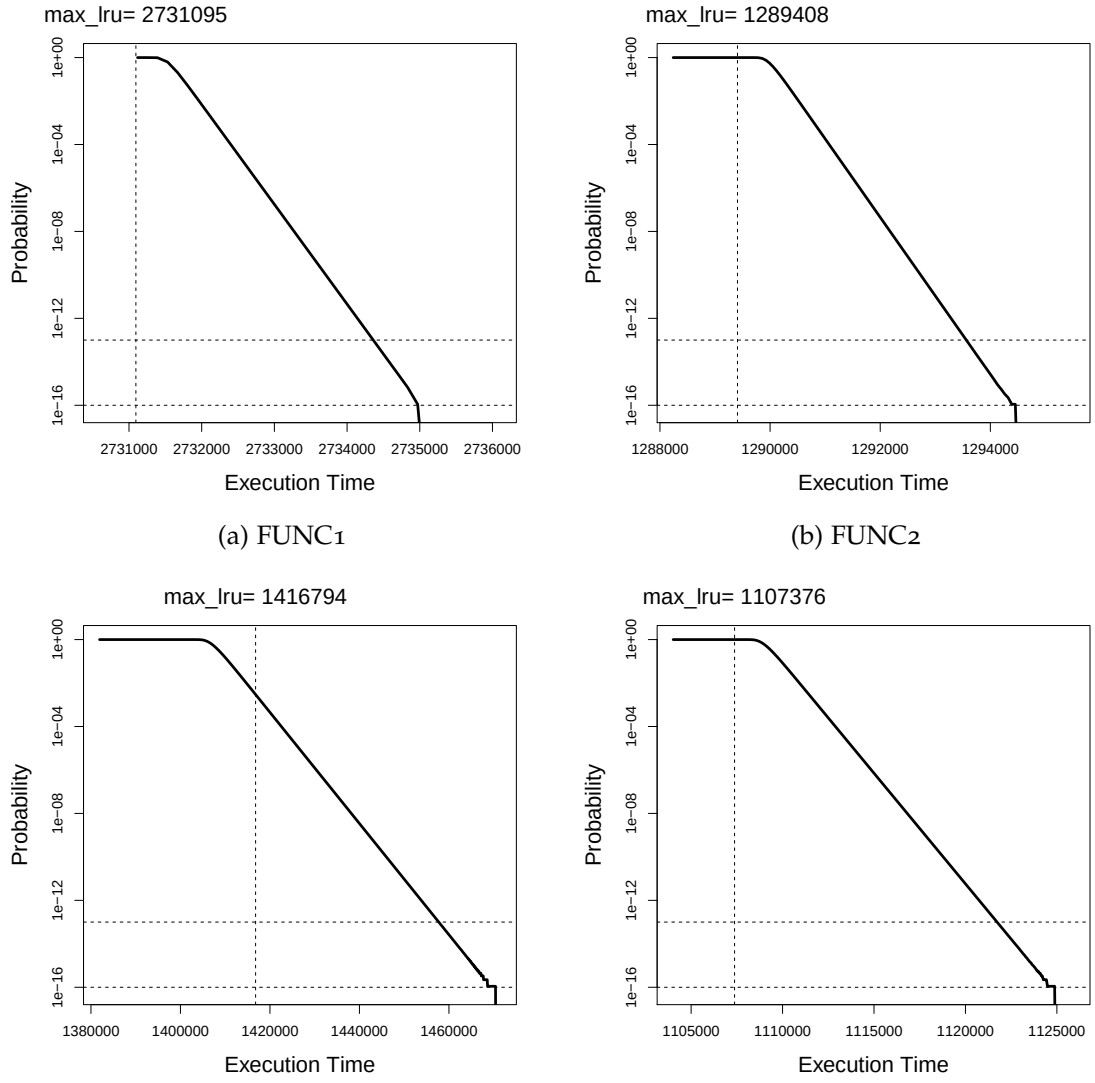
(b) FUNC2

(c) FUNC3

(d) PS

Figure 28.: CRPS variation as we increase number of collected execution times. $N_{delta} = 50$ and threshold $= 0.001$

[156]. To translate that failure rate requirement into the equivalent exceedance probability threshold, we need to know the frequency at which jobs are released. Thus, if we consider that tasks under analysis are released with a frequency of 10^{-2} seconds (i.e. 10^2 activations per second), the pWCET of that task should have an exceedance probability in the range $[10^{-14}, 10^{-15}]$: $10^{-9} \frac{failures}{hour} / \left(3600 \times 10^2 \frac{task\ activations}{hour}\right)$. Therefore, an exceedance probability threshold of 10^{-15} suffices to achieve the highest integrity level.



(a) FUNC1

(b) FUNC2

(c) FUNC3

(d) PS

Figure 29.: pWCET estimates (in processor cycles) for the programs under study. Horizontal lines stand for particular exceedance probability thresholds. Vertical lines stand for execution maximum observed time on deterministic LRU cache setup.

For the sake of illustration, we set our range of probabilities of interest to lie in the interval $[10^{-13}, 10^{-16}]$. Table 8 shows the pWCET estimates increment when increasing the exceedance probability from 10^{-10} to 10^{-13} and 10^{-16} (corresponding to a failure rate per hour of $10^{-5}, 10^{-8}$ and 10^{-11} respectively), highlighting that, as far as the functional blocks of the application under test are concerned, reasonably low extra time must be taken into account to decrease the exceedance probability threshold.

Table 8.: pWCET increase when raising the exceedance probability from 10^{-10} to 10^{-13} and 10^{-16}, which corresponds a failure rate per hour of $10^{-5}, 10^{-8}$ and 10^{-11} respectively.

	10^{-13}	10^{-16}
FUNC1	0.03%	0.03%
FUNC2	0.06%	0.07%
FUNC3	0.81%	0.87%
PS	0.26%	0.28%

Table 9.: Comparison of relative distance between max observed value wit LRU cache configuration and pWcet estimate with MBPTA

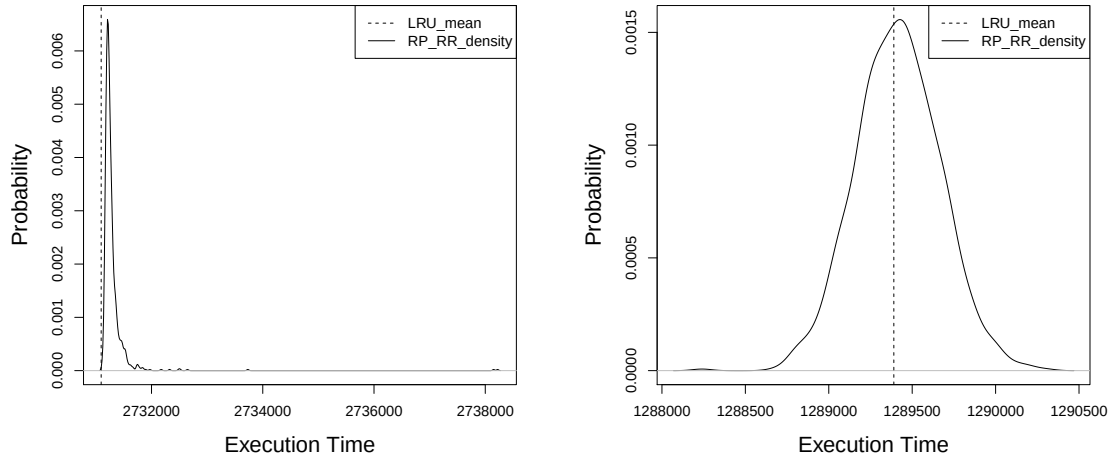
	LRU_MAX	pWCET	pWCET relative distance
FUNC1	2731095	2735109	0.14%
FUNC2	1289408	1294519	0.39%
FUNC3	1416794	1470490	3.78%
PS	1107376	1124934	1.58%

Figure 30 compares the density function of the observed execution times when running the programs under study on a processor equipped with the time-randomised cache (continuous curve) implementing random placement and replacement policies against a deterministic cache implementing modulo placement and LRU replacement policies (dashed vertical line), under exactly the same execution conditions. As shown in Table 10 and not surprisingly, the deterministic cache achieves better average performance on account of better average preservation of locality, however the overhead on average performance introduced by time-randomised cache is very low, actually below 1%. This suggests that the minor average performance loss caused by the time-randomised cache for the selected application is more than compensated by the low cost of applying probabilistic timing analysis.

Figure 29 also relates the average execution time obtained with the deterministic cache configuration against the pWCET estimates computed at an exceedance probability threshold of 10^{-16} using the random cache design. From those plots we see that

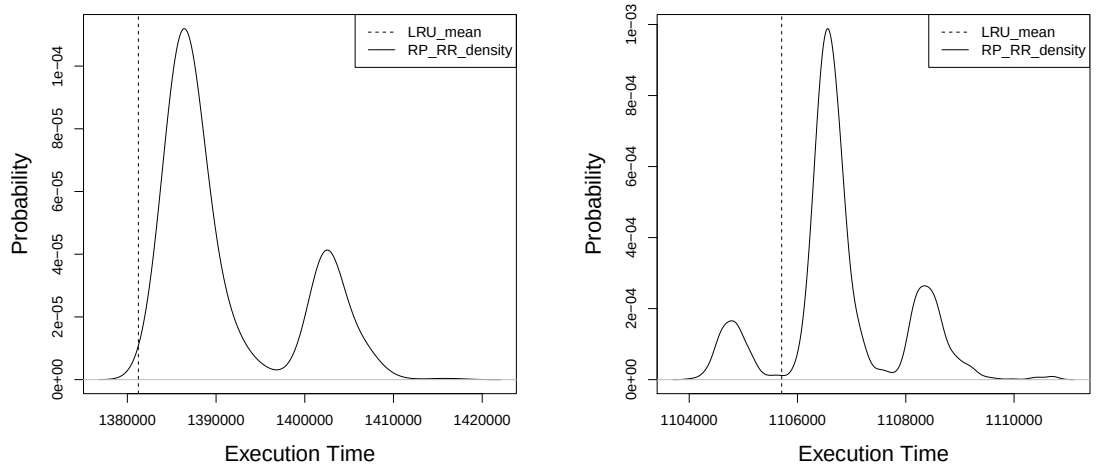
Table 10.: Comparison of relative distance between mean execution time value with deterministic cache configuration and time-randomised one.

	LRU_Mean	RP_RR_Mean	Relative distance
FUNC1	2731095	2731293	0.007%
FUNC2	1289390	1289415	0.001%
FUNC3	1381245	1391076	0.712%
PS	1105708	1106801	0.099%



(a) A653 RSM

(b) FUNC1



(c) FUNC2

(d) FUNC3

Figure 30.: Comparison of the density functions of observed execution times (in processor cycles) when running with the time-randomised cache (continuous curve) and a deterministic cache (dashed vertical line).

MBPTA produces tight bounds. As shown in Table 9 the pWCET are slightly higher but very close to the maximum observed execution time with caches configured in a deterministic way.

3.5 THE SPACE DOMAIN

While sharing some common concerns with the avionics and other high-integrity application domains, the space industry also features some very specific requirements which stem from the peculiarity of its development process. The first category comprises the traits common to the broader software development activity, such as reducing costs and time schedules, supporting incrementality of development and verification to mitigate changes and accommodate future needs. However, some other factors such as the requirement of fault isolation and recovery, and the technological limitations constraining flight maintenance operations are very domain-specific [10]. In this context, the limited size of this market and consequently the number of its contractors opened the door to the proliferation of different concepts, methodologies and technologies with respect to on-board software development, leading to distinct ways of producing the final software artefacts, whose interoperability is quite hard to guarantee. Although recent initiatives exist, mainly promoted by governative stakeholders as the European Space Agency (ESA), that aim at the definition and realisation of a software reference architecture [10] to be used in future missions, no unique framework has emerged so far as a standard practice for delivering a universally-accepted software architecture for space systems, as it is the case of IMA in the avionics. The convergence to a single, commonly agreed architecture would have the clear advantage of delivering software products which meet the intended quality, by focusing on the definition of their functional contents and avoiding unnecessary engineering effort; furthermore, that would make it possible for prime contractors to only focus competition on functionality and performance. As a consequence of this situation, when compared to the avionics world, the space industry lends itself to a more open and ambitious exploration of new design solutions which are not confined within the boundaries of a reference architecture anymore. We present next how the concepts related to time composability can be applied in this world.

3.6 ORK+

Moving within the boundaries of a restrictive specification like ARINC certainly represents the best condition to enforce time composability, thanks to the limited constructs and interactions thereof to be considered for system analysis, as discussed in Section 3.4. Unfortunately, the favourable assumptions on which that particular solution was built do not hold in general: they are instead of scarce utility outside the

very specific application domain of partitioned operating systems. For these reasons TiCOS can be considered a first attempt to inject time composability in a RTOS and the starting point towards the definition of a time-composable RTOS that is more flexible and lends itself to wider application. We want therefore to focus on understanding how and to what extent time composability, described in terms of zero disturbance and steady timing behaviour, can be achieved in a RTOS that supports a more general sporadic task model. The immediate challenge brought in by leaving the simple periodic task model is the absence of *a priori* knowledge on the release times of all tasks in the system.

In this context, the Open Ravenscar Kernel (ORK+ [157]), developed by the Technical University of Madrid, represented the perfect candidate as a reference OS on which we could try to inject time composability. ORK+ is an open-source real-time kernel of limited size and complexity, especially suited for mission-critical space applications, providing an environment supporting both Ada 2005 [158] and C applications. As the namesake suggests, ORK+ also complies with the Ravenscar profile [159, 160], a standard subset of the Ada language that specifies a reduced tasking model where all language constructs that are exposed to non-determinism or unbounded execution cost are strictly excluded[18]. As additional distinctive features, the Ravenscar profile prescribes the use of a static memory model and forces task communication and synchronisation via protected objects under the ceiling locking protocol. Ravenscar-compliant systems are expected to be amenable to static analysis by construction.

The design and implementation of a time-composable operating system of the scale and completeness of ORK+ is not an easy target however. Its extent and complexity require to approach the final solution by successive approximations. We attack the problem of injecting time composability into ORK+ in two steps, incrementally addressing what we singled out in Section 3.2 as characterising properties for a time-composable RTOS. As a first step we describe and assess some preliminary modifications we have made to ORK+ in order to improve its behaviour with respect to time composability, focusing on time management and scheduling primitives and under the simplifying assumption of a simple periodic task model; then we reason on whether and how our preliminary solution can be extended to also cover the remaining properties within a more complete task model.

18 ORK+ is therefore meant to support the restricted subset of the Ada 2005 standard, as determined by the Ravenscar profile.

3.6.1 *Time Management*

Keeping track of the passing of time in the OS kernel is heavily dependent on the underlying hardware components available to this purpose. In parallel to our effort of injecting time composability within ORK+, we also committed ourselves to porting the run time – which was originally available for the LEON2 processor (i.e., SPARC architecture) – to the PowerPC 750 board in the PowerPC family [161]. The porting effort was motivated by the poor support offered for time representation by the LEON platform, which made a zero-disturbance implementation of time management very hard: the SPARC platform does not offer any better than two 24-bit resolution timers, which alone cannot provide the clock resolution required by Annex D of the Ada Reference Manual[158]; the PowerPC 750 platform provides]fixinstead a 64-bit monotonic time base register and a 32-bit programmable one-shot timer. ORK+ cleverly copes with the LEON hardware limitation by complementing the 24-bit timer by software to obtain the accuracy of a 64-bit register [162].

As observed in Section 3.4.1, time composability may be easily disrupted by tick-based time management approaches that may inattentively interrupt and cause disturbance on the user application. Time composability instead is favoured where the hardware platform provides accurate interval timers that operating system can exploit to implement time management services with as less interference as possible. On the one hand ORK+ embraces this principle by basing its scheduling on a programmable interval timer, implemented as an ordered queue of alarms; on the other hand the kernel needs to use a periodic timer to handle the software part of its interval timer. Although the tick frequency is not likely to incur significant interference in the current ORK+ implementation a pure interval timer solution (i.e., with no periodic ticks) is still preferable as it better fits our call for zero disturbance and is facilitated by the PowerPC hardware.

3.6.2 *Scheduling Primitives*

Scheduling activities include maintaining a list of active tasks and keeping it ordered according to the applicable dispatching policy: in the restricted periodic task model we are considering for now, scheduling decisions are taken according to the fixed-priority preemptive scheduling policy and can occur only at task release and completion events. In ORK+ an activation event is managed by programming a dedicated alarm to fire at the time a task needs to be activated; ready tasks are then stored in a unique priority

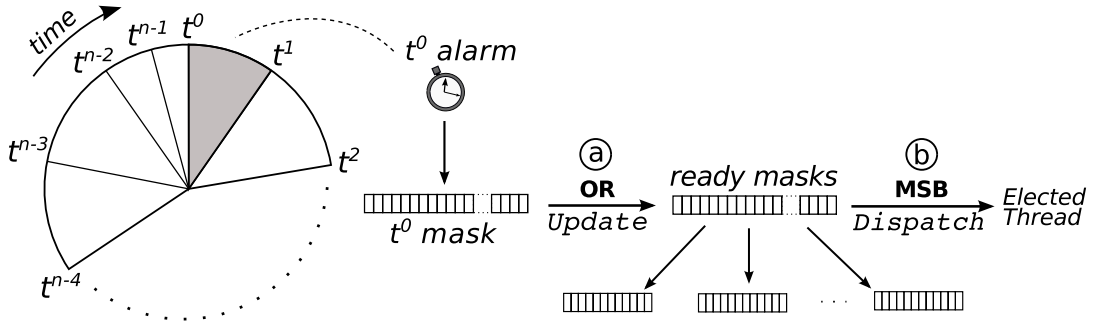


Figure 31.: Constant-time scheduling in ORK+

queue, ordered by decreasing execution priority and activation, where dispatching is performed by popping the head of such queue. At task termination nothing more than dispatching the next ready task needs to be done.

Although common practice, such an implementation has the undesirable effect of incurring highly variable execution times on accesses and updates to the involved data structures. In particular, having the alarm queue ordered by expiration time makes the insertion of a new alarm highly dependent on the elements already present in the queue (i.e. the number and value of the pending alarms at the time of insertion). This behaviour clashes with the mentioned principle of steady timing behaviour of OS primitives, with the risk for user applications to be exposed to variable – and only pessimistically boundable – interference from the execution of those primitives.

We re-designed the base ORK+ scheduling structures in a way to avoid this kind of interference. As an additional requirement we assume that tasks cannot share the same priority level. Since in our initial setting periodic tasks are not allowed to interact through protected objects, no priority inheritance mechanism is needed, and thus a task can never change the priority level it has been assigned to at the time of declaration. Our constant-time scheduler performs task management operations with fixed overhead, by leveraging on elementary operations over bit-masks, with a mechanism which is similar to that implemented by TiCOS; however in this case the scheduling operations are simplified, since we do not need to deal with the complications introduced by ARINC partitions and asynchronous services.

As shown in Figure 31, the two basic scheduling operations are (a) task insertion into the ready queue at the time of release, and (b) election of the next task to dispatch, which could either happen at the time of release of a higher priority task or upon task completion. The only required data structure we need to maintain is a 256-bit mask,

where each bit corresponds to one of the priority levels defined by ORK+: under the restriction that tasks should take distinct priorities, a 1 in position n means that the task with priority n is eligible for execution, 0 otherwise[19]. Our implementation consists of a 2-layered hierarchical data structure: the bit-mask itself is represented as a collection of eight 32-bit unsigned integers, and one additional 8-bit mask is required as root, in which a 1 is present in position k if the k-th child mask has at least one bit set to 1.

As a preliminary activity, during system start-up we initialise the alarms corresponding to all the task activations occurring during the hyper-period of the task set. Besides the timestamp of an event, each alarm also tracks which tasks need to be activated at that time, by keeping a mask with 1s set to the corresponding positions. Alarms are then organised in a circular linked list, similarly to [163], which will be looked up as execution time elapses to update the task states and set the next alarm. As soon as one alarm fires to signal the activation of a task, the following actions are performed:

(a) The mask corresponding to the fired alarm is bitwise OR-ed with the corresponding child bit-mask: in this way a 1 is set in the position corresponding to the task to be released.

(b) The task to be dispatched is identified by the most significant bit (MSB) set in the complete 256-bit-wide mask: the corresponding child mask can be identified in turn by looking at the most significant bit set in the root mask.

Performing step (b) above consists of finding the most significant bit within the base-two representation of an integer, once for the root mask to identify the leftmost non-empty child bit-mask and once to find the most significant bit within the child mask. This can be easily done by applying the constant-time de Bruijn algorithm [148] to both masks, thus guaranteeing constant-time latency in thread selection. At task completion nothing needs to be done for the terminating task, since alarms for next activations have been set at system initialisation; the election of the next task to dispatch is performed similarly.

Single-level bit-mask structures have been previously exploited by the Linux O(1) scheduler [164] (in the Linux kernel up to version 2.6.23) to provide a constant-time dispatcher. That solution, however, only applies to a strict time-slice model that acutely

19 A value of 0 is assigned also in case task at priority level n has not been defined in the considered task set.

clashes with our idea of zero disturbance. By exploiting bit-masks also in the time model we are delivering a more generic constant-time task activation mechanism that can be in principle extended to handle sporadic activations.

3.6.3 *Limited Preemption*

Modifying the kernel by addressing only a subset of the requirements we set on the RTOS, as described in the previous section, is sufficient to enforce time composability within the boundaries of a restricted periodic task model. Extending the proposed solution to support sporadic run time entities, programmable interrupts and protected objects is not straightforward as such a richer RTOS is much less amenable to time composability. Nonetheless those extensions are definitely required for a comprehensive RTOS, and especially so for Ravenscar-compliant systems where, for example, shared resources are the only permitted construct to provide task synchronisation.

Admitting sporadic task activations, the first natural extension of the task model, is difficult to accommodate in a time-composable manner. With the introduction of sporadic tasks, in fact, there is no longer static knowledge on task activation times, rather only a minimum inter-arrival time is known for them. The static activation table determined at system start-up works well for periodic tasks but can easily treat event-triggered sporadic activations by simply allowing update operations on the ready task bit-mask. Nevertheless, sporadic tasks need a prompt reaction (low latency) by the kernel when their release event occurs: prompt acknowledgement of sporadic activation events cannot be had without producing intrusive and disturbing effects on the user applications. Besides acknowledgement, the fact that each sporadic task activation may be followed by a task preemption runs against the concept of zero disturbance as the effects of preemption on a task do not follow a (more predictable and accountable) periodic pattern any more[20]. We have already mentioned how resorting to a simplified (and less performing) hardware architecture or taking interference into account in complex forms of schedulability analysis are not fully satisfactory solutions.

An alternative approach consists in trying to reduce inter-task interference by focusing on task preemption as its main root source, although it is widely acknowledged that preemption allows in the general case to improve the feasibility of a task set. In that light, a reduction in the number of preemptions experienced by each job of a task directly entails a reduction in the amount of possible interference. In fact, several approaches have been recently proposed to limit – or even altogether avoid – the

20 The same problem arises from low-level interrupts.

occurrence of preemption, while at the same time preserving the feasibility of the task set [41][75][165]. Those methods fall under the umbrella term of limited preemptive scheduling and have been recalled in Section 2.2. Most of existing approaches however rely on the simplifying assumption of task independence and seem not to make much of the fact that real systems often include shared resources which multiple tasks can access in mutual exclusion.

Limited Preemption and Shared Resources

Shared resources are often explicitly ignored in the proposed models to allow an easier mathematical formulation of the schedulability analysis problem or, when mentioned, are simplistically assumed to be encapsulated within non-preemptive chunks of execution. In real-world systems, instead, shared resources are a fundamental enabler to functional decoupling and predictable synchronisation patterns among tasks. As such, they convey design-level constraints that cannot be disregarded or handled retrospectively in a limited preemption scheme that aspires to be industrially relevant. In fully-preemptive systems shared resources typically need to be accessed in mutual exclusion so that data consistency is preserved across accesses. As a result of serialising accesses, a task may be blocked as it tries to access a resource that is already in use by a lower priority task, and thus may need to wait for the latter to relinquish the resource. In this case, the system is vulnerable to either potentially unbounded priority inversion or risk of deadlock. The use of an appropriate resource access protocol can prevent deadlock and establish a bound to the duration of priority inversion. Limited preemption scheduling arguably mitigates the problem of providing safe accesses to shared resources. At one end of the spectrum of limited preemption approaches, for totally non-preemptive systems, mutual exclusion, priority inversion and deadlock do not pose any threat as resources are not concurrently accessed anymore. While providing clear advantages in terms of time composability, disabling preemption may reduce system feasibility[21] and cannot always assure the required responsiveness for critical tasks. In less restrictive approaches, the problem of shared resource accesses is not avoided *a priori* but assumed to be solved by enforcing proper nesting of critical sections within non-preemptive execution chunks. This assumption, however, is only viable in static approaches where preemption points can be selected *ad-hoc* to include access to shared resources. The identification of fixed preemption points for all tasks in a system is a most undesirable obligation, also scarcely scalable to real-world systems.

21 Some task sets that are infeasible under rate monotonic scheduling are indeed schedulable in non-preemptive mode and vice-versa [166].

	C_i	$T_i = D_i$	U_i	q^{max}
τ_1	2	5	0.4	-
τ_2	2	9	0.222	3
τ_3	5	20	0.25	3

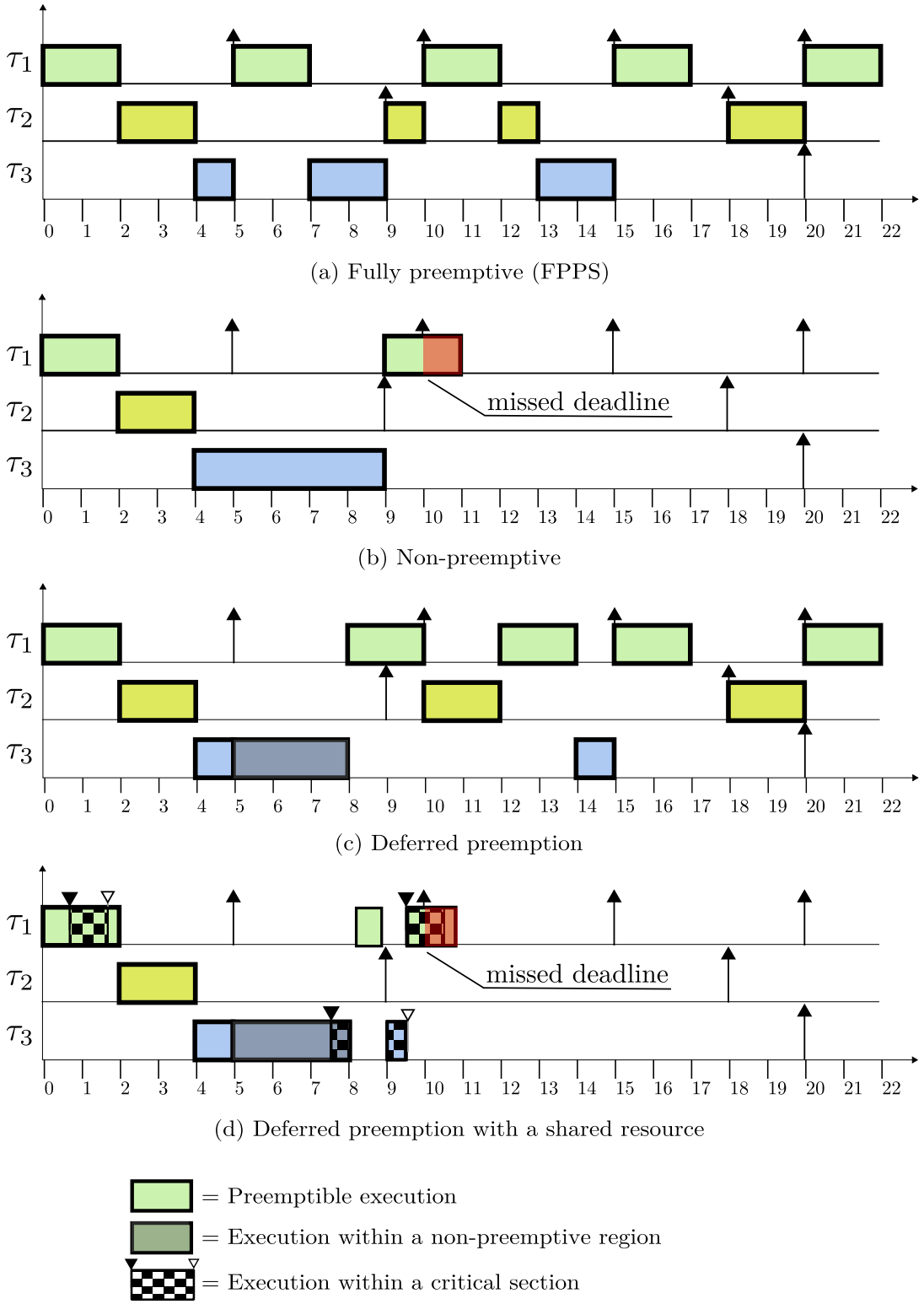
Table 11.: Example task set.

As noted in Section 2.2, the confinement of all shared resource accesses within non-preemptive regions cannot be generally taken for granted. In the activation-triggered deferred preemption approach, for example, the analysis model does not tell the exact position of preemption points, and the boundaries of a non-preemptive region depend on run-time events such as the activation of higher priority tasks. In this case the safeness of the model cannot be guaranteed and the schedulability of a task set may be compromised unless some sort of countermeasure is taken. This eventuality is confirmed by a simple motivating example involving the synthetic task set reported in Table 11, consisting in three periodic tasks $\{\tau_1, \tau_2, \tau_3\}$ with total utilisation $U = 0.872$.

As shown in Figure 32(a), the task set is schedulable under Rate Monotonic scheduling. In this particular case, non-preemptive execution is not an option: the task set simply becomes unfeasible when preemption is disabled as the uninterrupted execution of τ_3 causes τ_1 to miss its deadline, as shown in Figure 32(b).

According to the activation-triggered deferred preemption approach, the task set in Table 11 allows for NPRs of length $q_2^{max} = q_3^{max} = 3$ for both τ_2 and τ_3. Figure 32(c) shows how tasks τ_2 and τ_3 can exploit those NPRs to reduce the maximum number of preemption suffered while still preserving feasibility. Problems may arise, instead, when τ_1 and τ_3 share a resource. As shown in Figure 32(d), even in case of relatively short critical sections (just one time unit in the example) an unfortunate placement of a NPR in τ_3 may cause τ_1 to suffer more blocking than allowed, thus incurring a deadline miss. In this case, the problem is that the critical section begins but does not complete within the NPR and thus extends the non-preemptive region to some value $q_3^{max} + \epsilon$, which cannot be sustained. Notably, the presence of the shared resources alone does not compromise system feasibility as the adoption of a proper resource access protocol, even the simple priority inheritance protocol (PIP)[167], would produce a valid schedule for the example task set.

We regard the activation-triggered model as the most promising variant of the deferred preemption approach as it achieves a good compromise between effectiveness and applicability. This model in fact allows taking benefit of the slack in a system



(a) Fully preemptive (FPPS)

(b) Non-preemptive

(c) Deferred preemption

(d) Deferred preemption with a shared resource

Figure 32.: Motivating example.

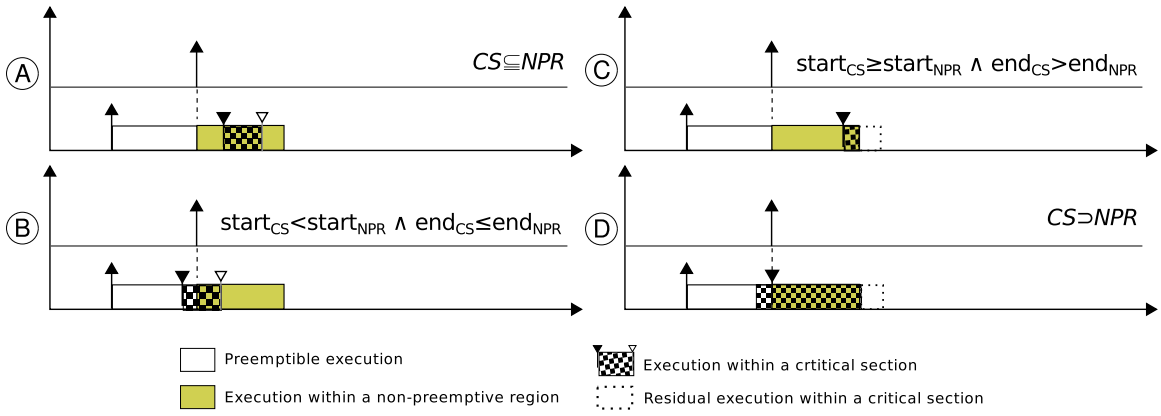


Figure 33.: Possible cases of overlapping.

without imposing any restrictive choice on the design: any decision is in fact deferred to run time. On the other hand, however, the activation-triggered deferred preemption model alone cannot prevent harmful overlap of NPRs with critical sections.

Handling Shared Resources

According to the activation-triggered model[22], an NPR of a task τ_i starts with the (first) activation of a higher priority task $\tau_j \in \mathcal{T}_i^+$ and finishes exactly after q_j^{max} units of execution. Since we cannot predict the exact starting point of the NPR, its actual placement may overlap with a critical section in any of the four cases reported in Figure 33, some of which can potentially harm the correctness and/or effectiveness of the deferred preemption model:

- *Perfect nesting*: pattern Ⓐ portrays the most convenient arrangement where a critical section is completely nested within a non-preemptive region, i.e., $cs \subseteq npr$;
- *Forerunning*: the overlapping in pattern Ⓑ is determined by the fact that a task τ_i may have already started its execution within a critical section when the activation of a higher priority task $\tau_j \in \mathcal{T}_i^+$ triggers a NPR. In this case τ_i will complete the execution within the critical section before the end of the NPR, i.e., $start_{cs} < start_{npr} \wedge end_{cs} \leq end_{npr}$;
- *Spillover*: a similar pattern in © represents the eventuality that a task τ_i enters a critical section within a commenced NPR but cannot relinquish the resource before the end of the latter, i.e., $start_{cs} \geq start_{npr} \wedge end_{cs} > end_{npr}$;

22 In this section we refer to the mathematical notation introduced in Annex B.

- *Reverse nesting*: the worst possible overlapping occurs in pattern Ⓓ, where τ_i enters a critical section before the start of a NPR and will not release the resource before the end of the latter, i.e., $cs \supset npr$.

Some preliminary observations are in order before discussing these patterns. The case when critical section and NPR do not overlap ($cs \cap npr = \emptyset$) is not accounted in the taxonomy: this is so because under the activation-triggered model it would just mean that no higher priority task can interrupt the execution within the critical section. Moreover, taking them to the extreme, patterns Ⓑ and Ⓒ may degenerate to a special occurrence of pattern Ⓓ. In addition, the activation event that is supposed to trigger the NPR in Ⓑ and Ⓓ is not easy to determine, as it may depend on the adopted resource access protocol, if any.

The perfect nesting in Ⓐ clearly shows the ideal situation that previous approaches to deferred preemption assume. In this case the NPR envelops the critical section and naturally grants mutually exclusive access to the shared resource. In fact, the form of protection guaranteed by NPRs to a task τ_i executing within a critical section is stronger than that provided by common protocols, as it inhibits all preemption requests from all tasks in the system instead of only those τ_j such that $\pi_i^b < \pi_j^b \leq \pi_i^a$, where π_i^a is the priority that a task assumes when holding critical sections in force of the applicable access control protocol.

The silent assumption in Ⓐ is that the execution within the critical section cannot exceed the duration of the NPR: $cs_{i^*}^R \leq q_i^{max}, \forall \tau_i \in \mathcal{T}$. This is a reasonable assumption as critical sections typically consist of short chunks of code. It is also true, however, that in case of nested or transactional[23] resource requests the cumulative length of a critical section becomes relevant. Interestingly, the opposite assumption is at the base of the reverse nesting pattern Ⓓ where the critical section strictly includes the NPR, implying that $\exists \tau_i, R, k | cs_{i,k}^R > q_i^{max}$.

The relationship between B_i^{CS} and B_i^{NPR} can be exploited to resolve this contradiction. Both B_i^{CS} and B_i^{NPR} contribute to the upper bound to blocking time that is included in the computation of the response time of a task. In those limited preemptive approaches where resource access protocols may coexist with NPRs (as they are limited to the final chunk of a job as in [166]) the maximum blocking suffered by a task is determined by the worst case blocking, that is

$$B_i = max\{B_i^{\text{CS}}, B_i^{\text{NPR}}\} \tag{1}$$

23 Sequential (non-cumulative) accesses to shared resources that require a protection mechanism to guarantee end-to-end consistency.

Within the activation-triggered approach, instead, blocking can only occur in conjunction with a NPR as for blocking to exist there must be an active higher priority task, which in turn is the condition for triggering a NPR.

It is therefore easy to prove that the duration of a critical section can never exceed the value of the longest non-preemptive region, thus at the same time confuting the assumption in Ⓓ (which becomes infeasible) and reinforcing the assumption in Ⓐ. We recall from Equations 5 and 8 in Annex B that both kinds of blocking are determined by a specific subset of $\mathcal{T}$. In particular B_i^{NPR} is determined by $\overline{npr}$, the longest NPR from tasks in $\mathcal{T}_i^-$, whereas B_i^{CS} is determined by $\overline{cs}$ the longest outermost critical section in $\mathcal{T}_i^B \subseteq \mathcal{T}_i^-$. Given a task $\tau_j \in \mathcal{T}_i^B$ the longest outermost critical section in τ_j that can block τ_i acts exactly as a NPR with respect to τ_i and by definition $q_j^{max} = max\{\overline{npr}, \overline{cs}\} \not< cs,\ \forall cs$ in τ_j. Consequently we may reformulate the relationships between the two kinds of blocking and observe that a safe upper bound to the amount of blocking suffered by τ_i is

$$B_i = B_i^{\text{NPR}} \geq B_i^{\text{CS}} \tag{2}$$

which is still consistent with the schedulability condition in the deferred preemption model.

Besides proving the infeasibility of Ⓓ, this observation also dispels the extreme cases in Ⓑ and Ⓒ where the end (resp. start) of NPR and critical section could in principle overlap. This notwithstanding, in contrast with the perfect nesting case, the interaction of NPR and critical section in those latter patterns does not seem to fit smoothly into the activation-triggered deferred preemption model.

The fact that the execution of a task τ_i within a critical section cs either precedes the beginning of a NPR, as in Ⓑ, or exceeds its end, like in Ⓒ, by an even small value ϵ, raises a twofold issue. On the one hand, it makes evident that, in standard activation-triggered model, NPR alone cannot guarantee serialised access to shared resources and thus a subsidiary protection mechanism (i.e., access protocol) may be required. On the other hand, enforcing mutually exclusive accesses regardless of their interaction with NPRs may incur the same effects as of extending the blocking factor B_j for all tasks $\tau_j \in \mathcal{T}_i^B$ that can be blocked because of cs. For these tasks q_i^{max} becomes $q_i^{max} + \epsilon$, which in turn invalidates the schedulability condition $B_i^{\text{NPR}} \leq \beta_i$.

In both cases we identified two possible approaches to preserve the task set schedulability. One possible direction consists in adapting the deferred preemption model to safely accommodate critical sections with a proper protection mechanism. Moreover,

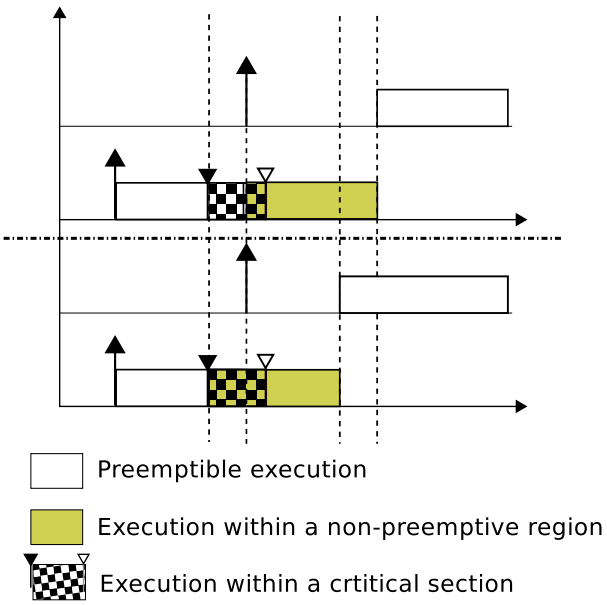


Figure 34.: Earlier start of NPR aligned with start of CS.

in consideration of the potentially negative effects of additional conservative choices in the model, we also put forward and evaluate a complimentary approach based on explicit support from the real-time kernel.

MODEL EXTENSION. When τ_i enters a critical section $cs^R_{i,k}$, an access protocol to reduce priority inversion and avoid deadlock is required, regardless of the relative position of the CS and NPR. Ideally the implementation should support ICPP, which is our protocol of choice. Unfortunately, the risk of overlap (and its destructive effects on schedulabililty) categorically excludes a naive application of ICPP.

Pattern Ⓑ is relatively simple to address. On the proved assumption that CS$\leq$ NPR, we can conservatively modify the model and trigger a NPR as soon as τ_i enters the CS, as shown in the bottom half of Figure 34. Triggering the NPR ahead of time protects the execution within the CS and preserves the upper bound on q_i^{max} as the maximum blocking caused to higher priority tasks will be bounded by $q_i^{max} - \epsilon$, where ϵ is the time elapsed between the beginning of the NPR and the first next arrival of a higher priority task. From the shared resource standpoint the positive effects are exactly the same as those of ICPP with $ceil(R) = \top$. The only drawback from bringing the NPR forward is that in the worst case we may incur one additional context switch as a result of the NPR finishing earlier.

Different issues arise when τ_i enters $cs^R_{i,k}$ whilst executing within a NPR. In the best case τ_i will release R before exiting the NPR, hence falling into the perfect nesting

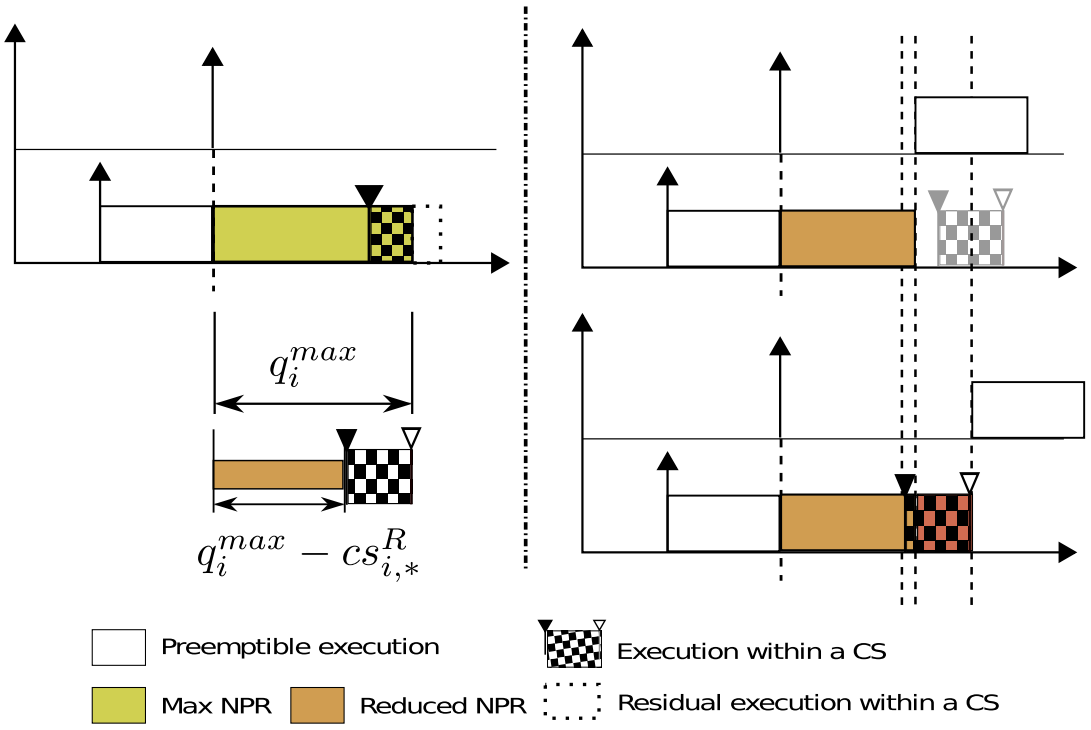


Figure 35.: NPR with safety margin.

pattern Ⓐ. In the worst case, instead, at the end of the NPR τ_i will still hold the lock on R; at this point inhibiting preemption any longer (by means of a resource access protocol) would break the schedulability condition, whereas allowing preemption would expose the system to the risks related to unprotected resource sharing.

The original model suggests in this case to deliberately reduce q_i^{max} so as to allow for an occasional spillover of τ_i over the NPR to complete its execution within $cs_{i,k}^R$. The rightmost part of Figure 35 shows the two possible outcomes of this approach: we may either acquire the CS within the reduced NPR or not even be able to try.

Conservatively adopting a more pessimistic q_i^{max} may reduce the effectiveness of the deferred preemption approach, proportionally to the length of the CS. The new $\widehat{q_i^{max}}$ should in fact be able to amortise the longest critical section in τ_i (if any), that is $\widehat{q_i^{max}} = q_i^{max} - \max_R cs_{i,*}^R$, which may result in extremely small non-preemptive regions.

To summarise, our model augmented to accommodate shared resources differs from the activation-triggered formulation in that a NPR can now be commenced in response to one of two events, either the activation of a higher priority task or the acquisition of the lock on a critical section, in case we are not already within a NPR. With regard to schedulability, since NPRs are floating within τ_i we cannot determine which overlap

will actually occur; hence $\widehat{q_i^{max}}$ should be taken as a safe value. This is clearly a minus in systems characterised by critical sections of comparatively long duration.

RUN TIME SUPPORT. The proposed extensions to the activation-triggered model are not fully satisfactory from the time composability standpoint. In cases Ⓑ and Ⓒ in Figure 33 the pessimism caused by the need to support ICPP brings in additional and not always necessary preemptions, which is against our main motivation. Better results can be obtained by exploiting run-time knowledge to reduce the pessimism in the model. To this end, low-level support from the underlying operating system is required.

We identified some room for improvement in both solutions and implemented a run-time mechanism to limit the pessimism entailed by the conservative choices made at the model level. The shortcomings in the model originate from the need to guarantee protected access of τ_i to a shared resource without violating the upper bound computed on q_i^{max}. Setting aside a safe margin from q_i^{max}, as we did, has the effect of reducing our taxonomy to pattern Ⓐ alone. In fact, all critical sections turn out to be perfectly nested within the implicit interval $\widehat{q_i^{max}} + \max_R cs_{i,*}^R$, where $\max_R cs_{i,*}^R$ is a safe bound. Over-provisioning is not always necessary: a dynamic control based on run-time knowledge allows intercepting potentially harmful effects of uninterrupted execution and prevent them from occurring. The only information we need from the underlying kernel concerns the set of active tasks and the kernel clock; and this is only relevant at the beginning and end of a critical section, where common resource access protocols operate.

With respect to pattern Ⓑ, triggering a q_i^{max} long NPR just to protect the execution of τ_i within a critical section may be an unnecessary precaution (from the deferred preemption standpoint) when no higher priority task was released in the meanwhile. We use this information when τ_i exits the CS and decide whether to execute a further NPR of length $\widehat{q_i^{max}}$, which is still safe, or to simply keep on executing, in case no higher priority task had been released yet, as depicted in Figure 36. This solution avoids any unnecessary preemption due to bringing forward the start of the NPR.

Pattern Ⓒ is a bit more thorny. Once τ_i enters a CS it is in fact just too late for any decision to be taken. As previously observed in this section, we cannot protect the execution of τ_i within the CS solely against a subset of the task set, as it would still count as a NPR for those tasks and possibly produce a cumulative NPR of unsustainable duration. Therefore accessing a resource from within a NPR is an *all-or-nothing* choice: once τ_i is granted a resource it should be also guaranteed to execute undisturbed at

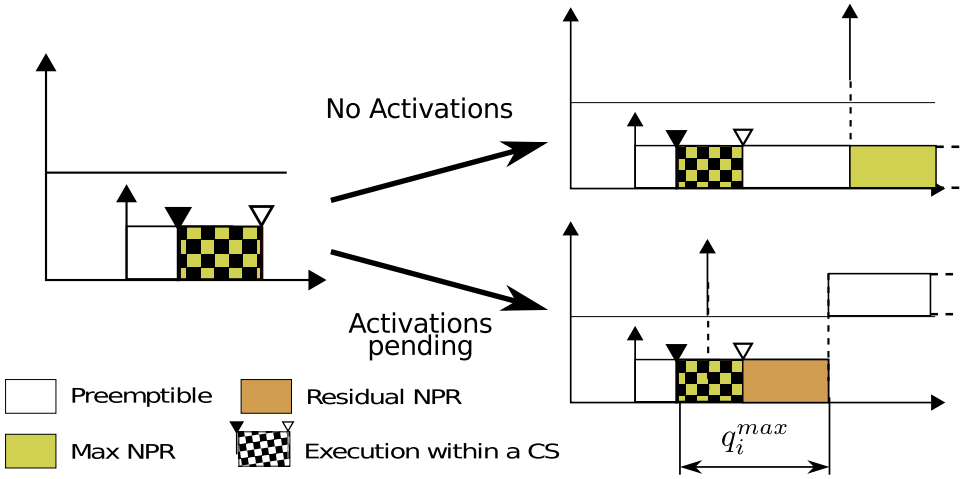


Figure 36.: Selective NPR at run time.

least until the end of the CS itself, at the unacceptable risk of exceeding q_i^{max}. Again, enforcing a curtailed upper bound $\widehat{q_i^{max}}$, though safe, is functional to having a safety margin to cover the execution if τ_i runs into a CS within the NPR.

In fact, it suffices to determine that enough residual NPR time is left to accommodate CS. Based on this information, which we are able to compute when τ_i requests CS, we decide whether to grant it or deny it. In the latter case, we also prematurely interrupt the NPR. Returning to the example in Figure 35, the blank space between the reduced NPR and the (blurry) CS in the upper-right corner hints at the improvement we can expect to the NPR duration: in the average case, τ_i is granted to execute in a NPR of duration $\widetilde{q_i^{max}}$, which improves on the conservative approach as it holds that $\widehat{q_i^{max}} \leq \widetilde{q_i^{max}} \leq q_i^{max}$.

3.6.4 *Evaluation*

The assessment of time composability achieved on the modified ORK+ has been conducted on two fronts: first we performed a preliminary evaluation of the modified kernel primitives against their original implementation in ORK+. It is worth noting that although ORK+ was not designed with time composability in mind, at least it has been inspired to timing analysability and predictability. For this reason, the original ORK+ was already well-behaved with respect to time composability and the effects of our modifications may not stand out as in the case of less educated operating systems like TiCOS. Second, we wanted to confirm the expectations on our approach and evaluate both correctness and effectiveness of the extensions we proposed to the

deferred preemption model for handling shared resources. To this end we compare our solution, which is based on a safe interaction of NPRs and CS, against a standard fully preemptive model, implementing a resource access protocol. We expect our approach to incur less preemptions, while at the same time providing the same guarantees on mutual exclusion. At the same time, we also want to compare the more conservative model solution with the respective run-time optimisation, to evaluate the effectiveness of our solution in avoiding unnecessary preemptions and size the degree of pessimism in the model. Unfortunately, no comparison is possible between our solution and the original activation-triggered model since we cannot integrate critical sections in the latter, as we already observed.

Approach and Setup

We conducted our experiments on top of the highly-configurable SocLib-based PowerPC 750 [161] simulation platform we used as TiCOS experimental setup. This allowed us to implement a zero-overhead instrumentation mechanism to gather experimental data.

For what concerns the memory layout, we configured the simulator to operate with 2 different set-ups depending on the two categories of experiments we wanted to perform. A configuration with 16 KB, 4-way set associative and 16 B line size instruction and data caches with Least Recently Used (LRU) replacement policy has been chosen for the measurement of the kernel first refactoring and its disturbance on the application. In this case we wanted to observe the timing behaviour of the task management primitives in isolation as well as the disturbing effects that the latter may have on an end-to-end run of the user applications. No measurements have been made on time management services in that, as observed in Section 3.6, both the original and our implementations, though different, incur no interference. This setup is motivated by the need of making our setting as close as possible to a real-world scenario; the LRU replacement policy has been chosen in consideration of its predictable behaviour [168]. Since zero disturbance was one of our guiding principles towards time composability, we generally took advantage of an automatic cache disabling mechanism available in the simulator as a quick workaround to isolate the effects of the RTOS on the user application cache state[24]. We then exploited measurement-based timing analysis, as we observe that the assessment of the steady timing behaviour and zero disturbance

24 We are aware of the potential implications of this solution on the overall performance and we considered implementing and evaluating a limited form of software-based cache partitioning approach to prevent that kind of side effects by reserving a piece of cache for the RTOS.

properties can be more easily conducted by means of measuring a small number of examples. On the contrary, we think that the use of a safer static analysis tool would have made it more complicated to perceive the execution time jitter. Once again, we used RapiTime [26], a hybrid measurement-based timing analysis tool from Rapita Systems Ltd., to collect and elaborate the timing information from execution traces. We performed an exhaustive set of measurements under different inputs or different workloads so as to highlight the weaknesses of the RTOS services with respect to time composability.

When evaluating our extension to the deferred-preemption framework instead, we configured the simulator to operate with both instruction and data cache disabled so as to level the costs of preemptions and avoid jittery timing behaviour in the application tasks. In this case indeed we are just interested in the raw number of preemptions incurred by each task set without caring about kernel overheads[25].

Results

CONSTANT-TIME SCHEDULER. We focused our experiments on those kernel primitives that are responsible for updating the ready queue in ORK+. The ready queue is accessed by the *Insert* and *Extract* procedures that are respectively invoked upon task activation and self-suspension (at the end of the current job). Task selection, according to the FIFO-within-priorities dispatching policy, is done by exploiting a reference (thus trivially in constant time) that is constantly updated to always point to next thread to execute. The *Insert* and *Extract* procedures are also used to access the data structures involved in our modified implementation and are thus directly comparable with their original version. In addition, thread selection in our implementation does not consist in dereferencing a pointer anymore, but uses a dedicated procedure *Get_First_Thread* to extract this information from a hierarchical bit-mask. However, it was not necessary to measure the *Get_First_Thread* in isolation, as it implements a constant-time perfect hashing function and its execution time is included in the invocation of both the *Insert* and *Extract* procedure.

Figure 37 contrasts the timing behaviour of the scheduling primitives from the original ORK+ (left) with that obtained from our modified implementation (right). The jittery execution time of the *Insert* procedure in ORK+ stems from the nature itself of the data structure: the position in the queue in which the task will be inserted depends on the number and priority of the already enqueued tasks. On

25 Considering kernel overhead in a limited-preemptive scheduling framework is part of our ongoing work.

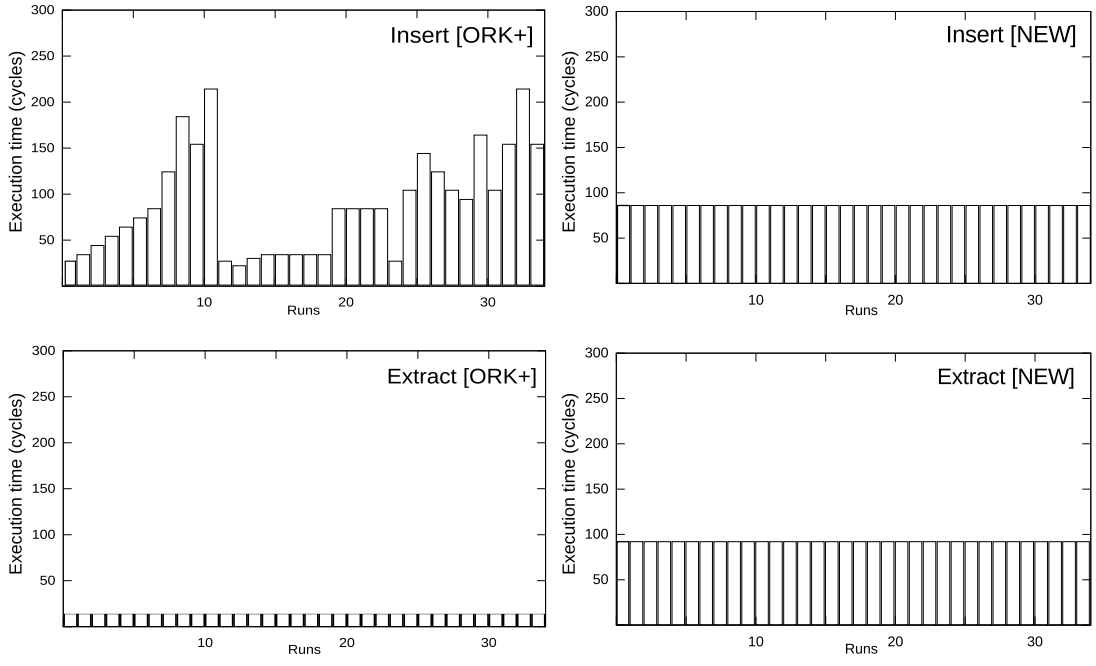


Figure 37.: Experimental results on scheduling primitives.

the contrary, the implementation based on bit-wise operations exhibits an inherently constant behaviour and a remarkably low execution bound. The *Extract* procedure instead already exhibits a constant-time behaviour in ORK+, as it simply consists in dereferencing a pointer. Our implementation, again relying on a bit-mask structure, is still constant though slightly less performing than its original counterpart. Yet, the performance loss is negligible in our reference platform, as the constant-time operation takes less than 100 cycles.

When it comes to gauging the effects of different implementations of kernel primitives on the application, we measured the same synthetic application when executing under different workloads (form 2 to 32) of strictly periodic tasks, each performing though at different rates, an identical set of constant-time operations. We measured the end-to-end execution time needed to complete each task's job and the scheduling operations, without including the cost of the low-level context switch itself.

Results are shown in Figure 38, where each point in the graph represents the difference between the maximum and minimum observed values along more than 1000 runs. The jitter, though bounded, in ORK+ grows proportionally to the number of tasks in the system, as a consequence of the variable size of its data structures, whereas our implementation performs in constant time regardless of the workload. The jittery

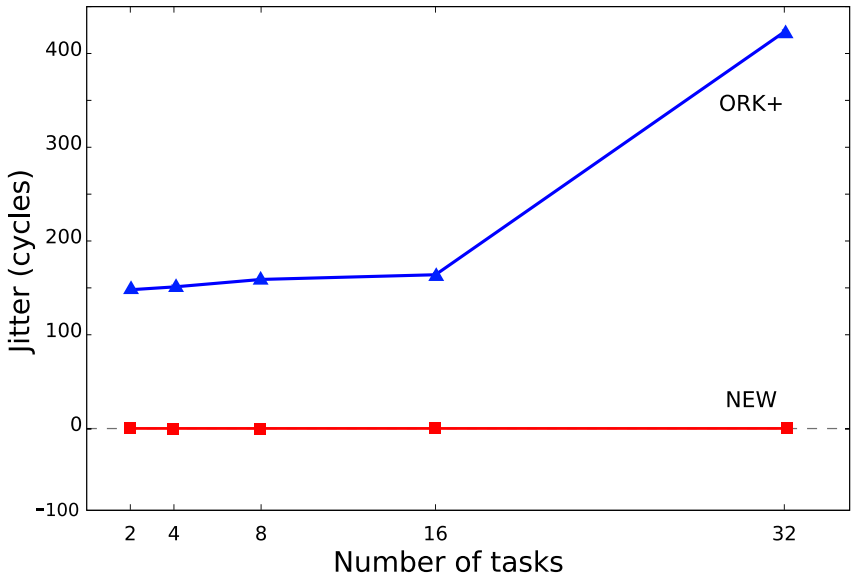


Figure 38.: Execution time jitter as a function of the task set dimension.

timing behaviour is explained by the variable execution time of the insertion of a new alarm in the ordered alarm queue, which is required to program the next activation of a periodic task.

LIMITED-PREEMPTIVE SCHEDULER. The evaluation of the limited-preemptive scheduler with shared resources was performed on randomly generated task sets with increasing overall system utilisation, ranging in between 20% and 90%[26]. Single task utilizations were generated according to a random uniform distribution within the range $[0.001, 0.1]$. Tasks were also constrained to exhibit integral harmonic periods uniformly distributed within the $[10ms, 250ms]$ range, so as to keep the experiment size manageable and to have full understanding of task interactions on shared resources. With respect to the effect of shared resources, we wanted our experiments to considerably vary also on number, duration and relative position (within the task body) of critical sections in the system. In order to avoid an explosion in the possible scenarios we focused on the worst-case setting were all tasks share the one and same resource and thus compete for the same (system) ceiling. The CS duration, instead, is expressed as a fraction of the maximum allowed blocking (which also corresponds to q^{max}) and varies within 25% and 100% of the latter. The relative position of the CS accessing the shared resource is randomly determined following a uniform distribu-

26 Task sets with utilisation lying outside the considered range were either scarcely relevant or largely infeasible.

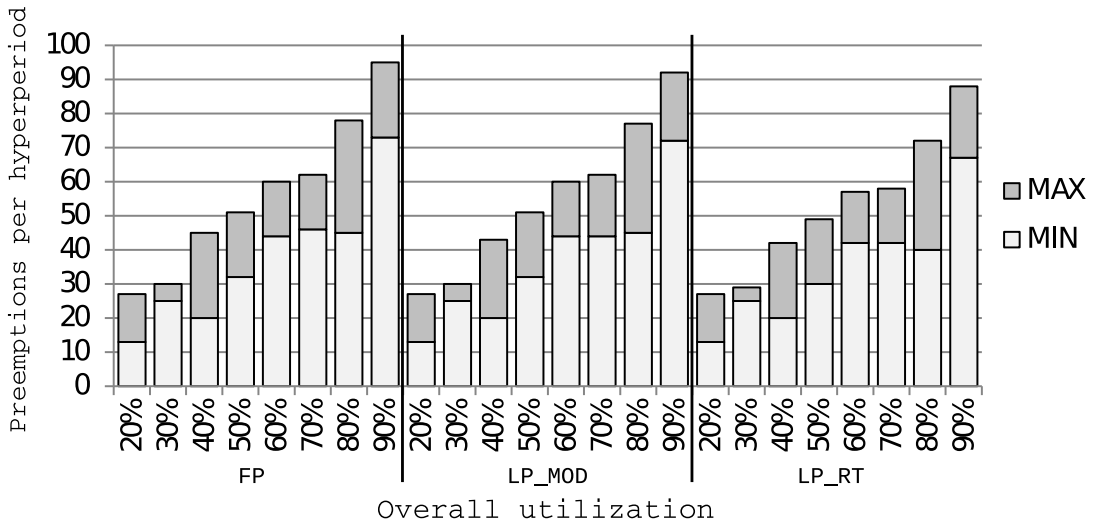


Figure 39.: Maximum and minimum cumulative preemptions for fully-preemptive and limited-preemptive scheduling in the presence of shared resources.

tion with range $[0, 1]$, the extreme ends of which stand for the beginning and the end of task body respectively. All the experimental scenarios were verified to be feasible by RTA (accounting for kernel overheads) and where confirmed to be so by run-time monitoring in the kernel.

Our first set of experiments had a twofold objective. As an initial step, we wanted to ascertain the soundness of our approach with respect to shared resource access protection. Then, we wanted to assess whether our limited-preemptive scheduling approach in the presence of shared resources still allows a reduction in the number of preemptions, as the state-of-the-art model does. We instrumented the kernel to intercept potential deadline misses and to count the number of preemptions globally incurred by all tasks in a task set during its hyperperiod. We collected the run-time traces of more than 100 randomly generated task sets per each utilisation level in the range 20-90%. Each task set was characterised by a random combination of CS durations and relative position within each task. Figure 39 compares our implementation against the fully-preemptive scheduler provided by ORK+. The graph herein reports maximum and minimum number of preemptions cumulatively incurred in a task set for each utilisation level by `FP`, `LP_MOD` and `LP_RT`, respectively the baseline deferred-preemption model accounting for shared resources and its run-time optimisation.

As a preliminary observation, we note that the absence of deadline misses confirms that our implementation of limited-preemptive scheduling provides a sound protection

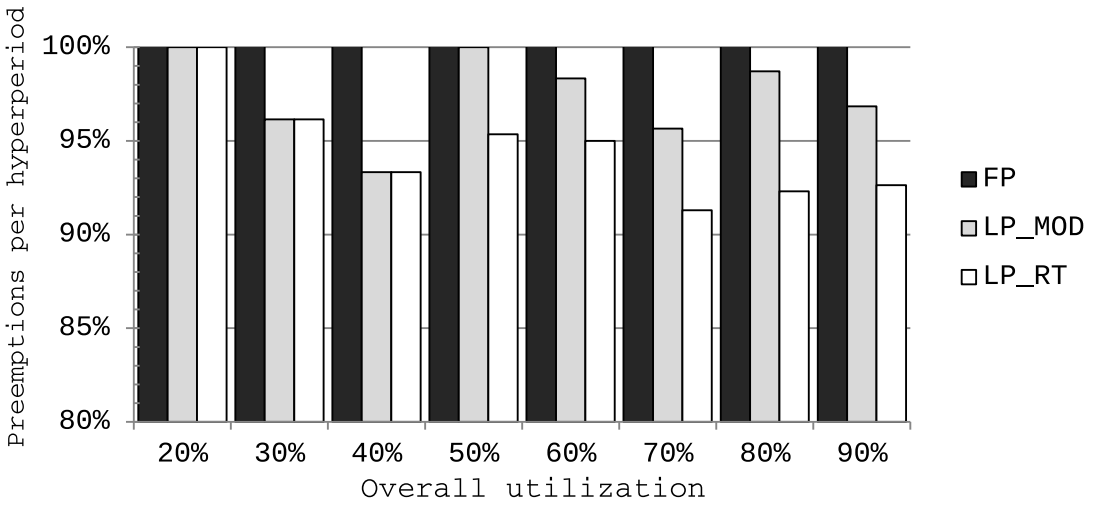


Figure 40.: Maximum number of preemptions avoided per task set.

mechanism for shared resources. Focusing on each algorithm separately, we observe that the number of preemptions in a task set simply increases with the total utilisation. This is in part due to the way the task sets were generated, as greater utilisations are obtained by increasing the size of the task set. However, higher utilisation levels (with relatively small tasks) typically entail more complex task interleaving and thus more preemptions. The most important observation that can be made from Figure 39 is that, in accordance to state-of-the-art results on limited-preemptive scheduling, limited-preemptive schedulers generally outperforms the fully preemptive one, even in the presence of shared resources. As expected, the basic `LP_MOD` scheduler performs slightly better than the `FP` algorithm in approximately achieving a 5% uniform reduction in the number of preemptions in medium- and high-utilisation task sets. No remarkable gain can be observed instead at extremely low utilisation levels (i.e., 20%). Besides proving the validity of our basic model Figure 39 also provides evidence of the improvement achievable at run time by adopting the optimisations described in Section 3.6.3. As already observed, a simple-minded extension of the limited-preemptive model to account for shared resources may incur some unnecessary preemptions. The `LP_RT` scheduler implementation, in fact, improves on the basic model extension and guarantees up to a 8% reduction in the number of preemptions globally incurred by the task set under scrutiny.

The potential benefit from applying our run-time optimisation is even more evident from the histogram in Figure 40, showing the maximum number of preemptions avoided by our LP scheduler implementations, in relation to the number of preemptions incurred by `FP` on the same task set. Data are extracted from the same set of

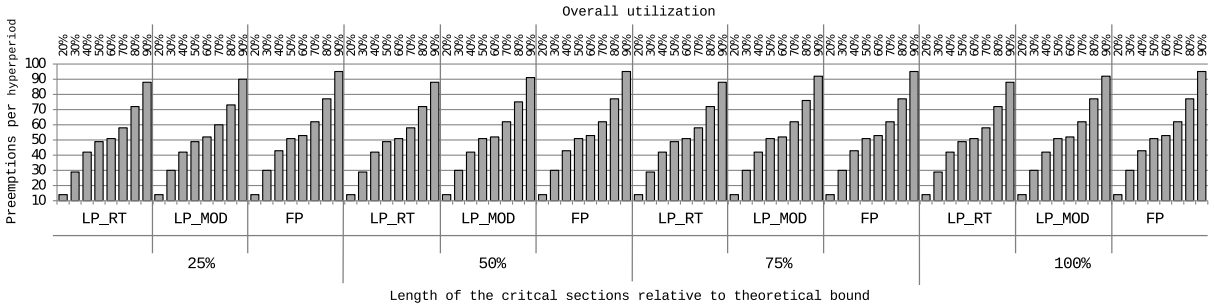


Figure 41.: Preemptions incurred by the three algorithms when the duration of critical section are stretched up to the theoretical bound.

experiments reported by Figure 39. They represent those task sets per utilisation for which we observed the maximum reduction in the number of preemptions for both variants. The number of preemptions incurred with LP_RT (the right-most bar of any triplet) is always no worse than what observed without it, for any scenario and any utilisation rate. This difference with respect FP and LP_MOD is increasingly more evident as the task set utilisation grows.

We finally investigated how the total number of preemptions incurred by the task set in a hyperperiod varies as a consequence of increasing the duration of blocking caused by resource sharing. To this end we gradually increased the duration of the critical sections (proportionally to each task duration) so as to cause larger blocking on higher-priority tasks. Figure 41 shows the results of a comparison between the fully-preemptive (*FP*) and our limited-preemptive variants LP_MOD and LP_RT on the number of preemptions incurred when the size of such critical section ranges over a representative set of equivalence classes (namely 25, 50, 75 and 100% of the theoretically admissible blocking derived from the computation of q^{max}), under increasing task set utilisation. The results confirm the goodness of the LP_RT implementation which incurs always less preemptions than those observed under the other settings. In addition, whereas the preemptions suffered by LP_MOD slightly increase with larger critical sections, the LP_RT solutions always incurs a constant number of preemptions, regardless of the duration of the critical section.

3.7 SUMMARY

In this chapter we introduced our notion of time composability within the OS layer in terms of zero disturbance and steady behaviour, the former to counter the OS-

induced perturbations on history-sensitive HW/SW resources, the latter applying to the timing behaviour of the individual OS primitives. We spotted 5 areas of intervention which should be accounted for in the process of designing or refactoring a time-composable OS: a non-intrusive time management mechanism based on interval timers together with a constant-time scheduler implementation help reduce the major part of OS-induced interference. Additionally, selective isolation of highly-unpredictable hardware resources is needed to reduce disturbance from the HW layer. Time-composable support to an expressive task model and to inter-task communication in particular is essential to provide solutions which aspire to be relevant in an industrial context.

We used those principles in the refactoring of an ARINC 653-compliant kernel for the avionics domain designed without composability in mind: the experimental evidence we provided demonstrates the reduced interference on user applications of both kernel routines and API-specific services. An additional case study developed in the scope of the EU FP7 PROARTIS project demonstrated how our time-composable kernel is crucial to enable the probabilistic timing analysis of a real-world Airbus application.

Finally, we moved to the space domain to explore a more ambitious task model which posed additional challenges in dealing with asynchronous events and inter-task communication. We implemented a limited-preemptive scheduler within the Ada Ravenscar-compliant ORK+ kernel and devised an extension to the deferred preemption framework to handle resource sharing among tasks. Experimental results confirmed the good results achieved in the ARINC setting previously investigated, and demonstrated how time composability may greatly benefit from reduced interference from task preemption.

4

TIME COMPOSABILITY IN MULTICORE ARCHITECTURES

So far we restricted the scope of our investigation on time composability to a favourable uniprocessor environment. We studied how the operating system contributes to the timing behaviour of applications, influenced by the use made of the available HW resources and the behaviour of the provided services, eventually determining the degree of achievable composability. Clearly, the task of providing a time-composable OS is greatly facilitated by the presence of a simple HW platform, whose features are designed with analysability in mind, which avoids the use of advanced, complex features for the purpose of boosting the average-case performance.

Although time composability is difficult to guarantee even on common single-core architectures and the provision of safety bounds through certification is still an extremely costly process, the interest in multicore platforms has rapidly grown in the high-criticality systems industry in recent times. Whether right or wrong, multicore processors are considered as an effective solution to cope with the increasing performance requirements of current and future real-time embedded systems. Multicores integrate multiple processing units or cores into a single chip, sharing multiple hardware resources, like caches, buses and memory controllers. By doing so, the hardware utilisation is increased, while cost, size, weight and power requirements are minimised. Moreover, multicore processors ideally enable co-hosting applications with different criticality levels. Unfortunately, even though a relatively simple core design is maintained, resulting HW architectures are usually convoluted and the interactions among their components are more difficult to capture than in a single-core setting; in this scenario the methodology and results of single-core timing are likely to be impaired once again as a consequence of the existing gap between system design and verification.

In this chapter we attack the problem of providing time composability in a multicore environment step-by-step: in Section 4.1 we look at the HW configuration to identify the sources of interference present in such platforms; then we consider two different computational models, multiprogram and multithreaded, corresponding to different ways of understanding multicore computing and of taking advantage from it (Section 4.2). In detail, in Section 4.2.1 we give an example of partitioned multicore

by looking at the avionics case study discussed in Chapter 3; this gives an insight on how the MBPTA techniques proposed by PROARTIS may be applied in a multicore scenario. More importantly, we show how time composability as discussed for single cores may apply to multicores as well, under specific conditions. In Section 4.2.2 we then focus our attention on the OS layer in a more ambitious multithreaded setup; in particular we investigate the multiprocessor scheduling problem to understand how the increased availability of HW resources affects the scheduling decisions to be taken, when matching available processors to application needs. We look at RUN as a valid algorithm to achieve a good compromise between time composability and performance (Section 4.3), and we propose an extension (called SPRINT) to handle sporadic task activations and make the algorithm more attractive from an industrial application perspective. We finally evaluate our new algorithm in Section 4.3.3 against state-of-the-art multiprocessor scheduling techniques.

4.1 HARDWARE ARCHITECTURE

Multicore processors can be broadly classified into two groups, heterogeneous and homogeneous, based on the computational power of their individual cores and on the memory access layout. Homogeneous multicore processors include only cores with the same computational power, whereas in heterogeneous multicore processors the speed of different cores can be different. Multicores can also be classified based on their design as symmetric (SMP) or asymmetric (AMP). In the first case all cores have access to a single memory address space, while in the latter cores are connected to disjoint memory address spaces.

Figure 42 [169] shows an example of a homogeneous symmetric (PTA-capable) multicore platform composed of four cores, sharing a second level cache and a memory controller through a shared bus. This is a good reference architecture for the discussion in this section, where we focus on homogeneous SMPs with a small number of cores to confine ourselves within a simple Network on Chip (NoC) design with few shared resources. The reason is that a larger number of cores complicates the interactions for access to shared hardware resources.

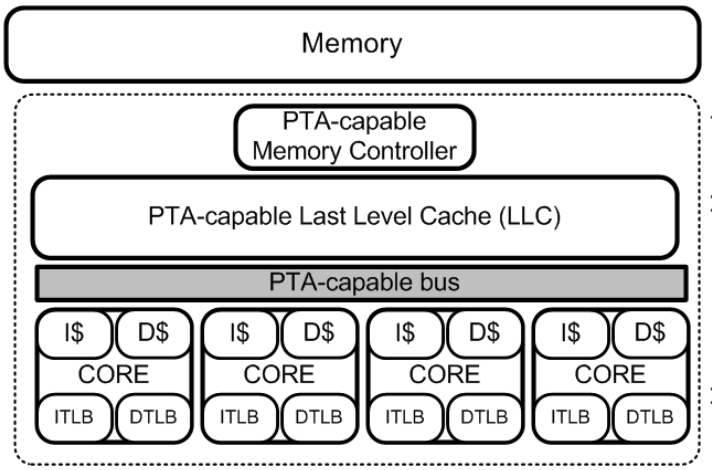


Figure 42.: Example of homogeneous symmetric multicore processor.

4.1.1 *Sources of Interference in a Multicore System*[1]

As discussed in the preceding chapters, the disturbance suffered by an executing application largely depends on the timing behaviour of the HW resources featured by the processor of choice. In a traditional single-core setting – i.e. outside the scope of probabilistic analysis techniques – shared HW resources like memory hierarchies, pipelines and TLBs are largely responsible for the jittery timing behaviour of an application, whose execution time is likely to depend on the execution history of the system, as determined by the execution pattern of active applications. Given two instructions I_x and I_y, where x and y are the cycles in which each instruction is executed, I_y may impact the execution time of I_x if it belongs to its execution history, i.e. if $y \leq x$.

Because in multicore systems many processing units execute in parallel, applications are exposed to an additional form of interference which originates from the race for gaining exclusive access to the same shared HW resources. This form of inter-task interference makes the execution time of a task dependent on the other execution threads running simultaneously. Therefore, the timing behaviour of an instruction does not only depend on the instructions previously executed in the history of the system, but also on those belonging to other tasks running simultaneously. Given two instructions $I_x^{\tau_1}$ and $I_y^{\tau_2}$ belonging to two different tasks τ_1 and τ_2, $I_y^{\tau_2}$ may impact the execution time of $I_x^{\tau_1}$ if it belongs to its execution history, i.e. if $y \leq x$. If no precedence

1 Contributions to this section are taken from [170], authored by the Barcelona Supercomputing Center.

order can be established on the execution of the tasks, then any instruction of τ_1 may affect τ_2 and vice-versa.

In fact, this means that multicores take dependence on execution history to the next level: not only the execution time of a piece of software varies as a function of the *past* history of execution, but it also depends on the *current* execution, as evolving in parallel on other cores. For this reason some studies [171] propose to desist from chasing multicore time composability, and to rather characterise applications timing behaviour as a function of their co-runners, in a degraded form of compositional (as opposed to composable) analysis.

Given that different HW resources have different impact on task execution times, interference can be classified into three types, according to its root source [170].

ARBITRATION INTERFERENCE. Arbitration occurs when two or more tasks running on different cores try to access a shared resource at the same time. To handle this kind of contention an arbitration mechanism is required, which causes one task to wait until the other completes and relinquishes the resource. Resources that suffer this type of interference are the following:

- *Buses* have fixed latency, which is the amount of time necessary for a request to cross them. When a request is granted access to the bus, no other simultaneous use is allowed and therefore interference occurs. The most common arbitration policies are priority-based, round-robin and time-division multiplexing.

- *Caches* are normally composed of multiple independent banks to enable parallel access, so that different memory operations can address different banks simultaneously. However, a bank can only handle one memory request at a time. When a bank is serving a memory request, it is inaccessible for any other operation and any arbitration choice introduces cache-bank interferences. In most processor designs, since the shared cache is connected to cores through a bus, the bus arbiter itself can be charged with resolving bank conflicts.

- *Main memory* is composed of a memory controller and one or more off-chip memory devices. The memory controller regulates the accesses to the off-chip memory, acting as a proxy between processor and memory. The memory device itself is composed of independent banks. Similarly to buses, the channel to access the memory device is shared among tasks, thus an arbitration policy is required to avoid conflicts; similarly to caches, banks can only handle one request at a time, thus the same arbiter can avoid bank conflicts as well.

STORAGE INTERFERENCE. Storage interference appears in shared-memory schemes when one task evicts valid memory data blocks used by another task. As a result the latter task requires accessing to higher memory hierarchy levels in order to re-fetch its data, increasing the overall latency. Resources that suffer this type of contention are:

- The *shared cache*, which maintains data recently accessed by all tasks. Due to its limited size, cache lines from different tasks are evicted when new space is required, producing additional conflicts that do not exist when tasks are run in isolation.

- The *main memory*, which can suffer storage contention due to page swapping similarly to the cache. However, this effect only happens when the machine runs out of memory, which is usually not allowed in real-time systems due to overhead implications.

RESOURCE-STATE INTERFERENCE. Resource-state interference occurs in resources whose response time is sensitive to their specific state at run time. As a result, the latency of such resources depends on the operations performed by other tasks that recently accessed them. This is the case for example of DRAM memory devices, whose response time may vary according to the Row-Buffer Management policy used; for example, the latency of a "read" operation may depend on the previously served requests, getting shorter if the preceding request was also a "read" than if the preceding request was a "write" operation.

The approaches to counter these forms of interference are twofold, either removing them by assuming their worst case at any time or bounding them by means of probabilistic approaches, as it was done for the single-core setting.

In the former case, to account for arbitration interference, the worst-case latency of a request on the interconnection network can be assumed. Requests on the bus are delayed, at analysis and deployment time, so that they manifest always this latency. For example, if a bus connecting all on-chip elements with a latency of 2 cycles and a round-robin arbitration policy is considered , the maximum inter-task latency for accessing the bus is $InterTaskLatency = (NumCores - 1) \times BusLatency$. To remove storage interferences instead partitioning techniques can be applied. Cache partitioning is a well-known technique that eliminates storage interferences by splitting the cache into private portions, each assigned to a different task.

In the latter case instead interference is characterised by means of probability theory. Random arbitration policies can be used to reduce arbitration interference, so that the task accessing a shared resource is randomly selected. In this case the probability of being granted access to the resource depends on the number of simultaneous requests n.

For example, considering a shared bus with a random arbitration policy with a latency of L cycles, the Execution Time Profile (ETP) can be defined as:

$$ETP_{BUS} = (\{L, 2L, 3L, \ldots, kL\}, \{1/n, (n-1/n)(1/n), (n-1/n)2(1/n), \ldots, (n-1/n)k-1(1/n)\})$$

where k is the number of tries a bus request is not granted access to the bus. This may result in an infinite ETP, in which there is an infinitesimal probability of never being granted access to the bus. For storage interference instead, random cache designs can make the address of an operation independent from the cache line in which it is actually placed. As a result, the impact due to storage interference on one task can be bounded, provided that the number of accesses from other tasks are known.

Having restricted ourselves to the simplified HW architecture in Figure 42 makes it possible to ignore additional issues related to interference in heterogeneous multicores, like in the presence of Dynamic Voltage Frequency Scaling (DVFS) techniques. Heterogeneous multicore processors differ from homogeneous processors in the way they access to shared resources. Because cores have different computational power, the frequency of access to shared resources varies among different cores and, as a result, the impact that one task may have on other tasks depends on the core assigned for execution. Similarly, if cores are allowed to dynamically shift their operation voltage (DVFS), the speed and so the frequency of access to shared resources will depend on the voltage operations of the core in which the task is run; clearly, the impact that this task will have on the execution time of other tasks will vary consequently.

4.2 MODEL OF COMPUTATION

Not only the HW configuration in use, also the way applications are deployed in a system and the use they make of the available HW resources largely concur to determining the run time behaviour of a system and eventually facilitate or impede its timing analysability. In the homogeneous multicore architecture presented in Section 4.1 the following two setups can be devised.

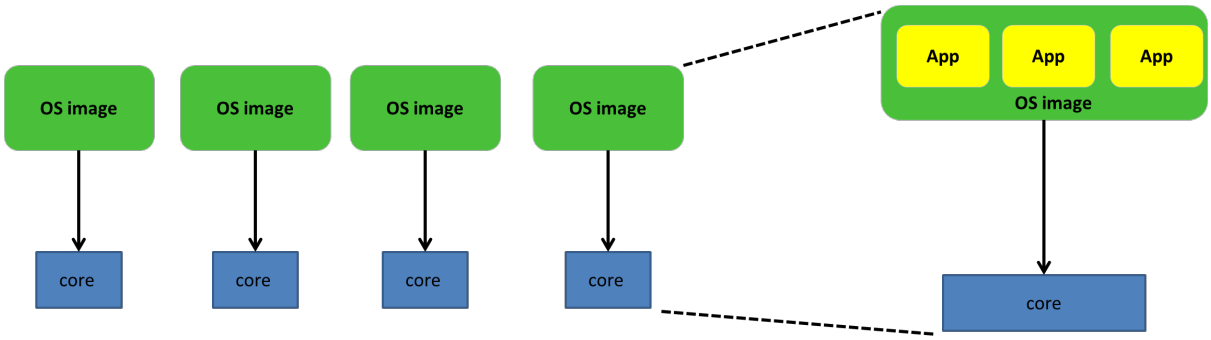


Figure 43.: Basic view of the MP setup.

MULTIPROGRAM (MP) SETUP. In this configuration a single control flow (associated to one executable image) is pinned to each core and is the only one allowed to execute on it. An image could consist of a single main procedure or an entire OS kernel with some application software running on top of it. In this case, the executable images loaded on each core are assumed to be single threaded and to exchange no data. Hence each application procedure is executed sequentially on a statically allocated core. In these conditions no direct inter-task interference can occur among applications running on different cores, defining in fact a *partitioned architecture*. In this scenario, the only focus is on bounding the effect of inter-core interferences, either by accepting their worst case or by enforcing mechanisms to guarantee the identical distribution and independence of processor instructions timing, this way enabling PTA techniques. In relation to the IMA system architecture referred to in Chapter 3 this arrangement maps to the case of having one TiCOS image per core, each containing any number of IMA partitions (Figure 43).

MULTITHREADED (MT) SETUP. In this setting, tasks are allowed to execute on any core and to exhibit data dependencies. In addition to the inter-core issues brought in by the MP multicore processor setup, this arrangement presents other challenges related to inter-task interference, like dealing with memory sharing, global scheduling of applications (including task migrations) and resource locking at the OS level. In relation to the IMA system architecture discussed in Chapter 3, this arrangement would correspond to having a single TiCOS executable containing a single IMA partition, whose threads can migrate across cores to be executed in any of them (Figure 44). However, applying this setup to an IMA architecture is just a conceptual exercise, since such a scenario would run against the space and time isolation concerns advocated by that specification.

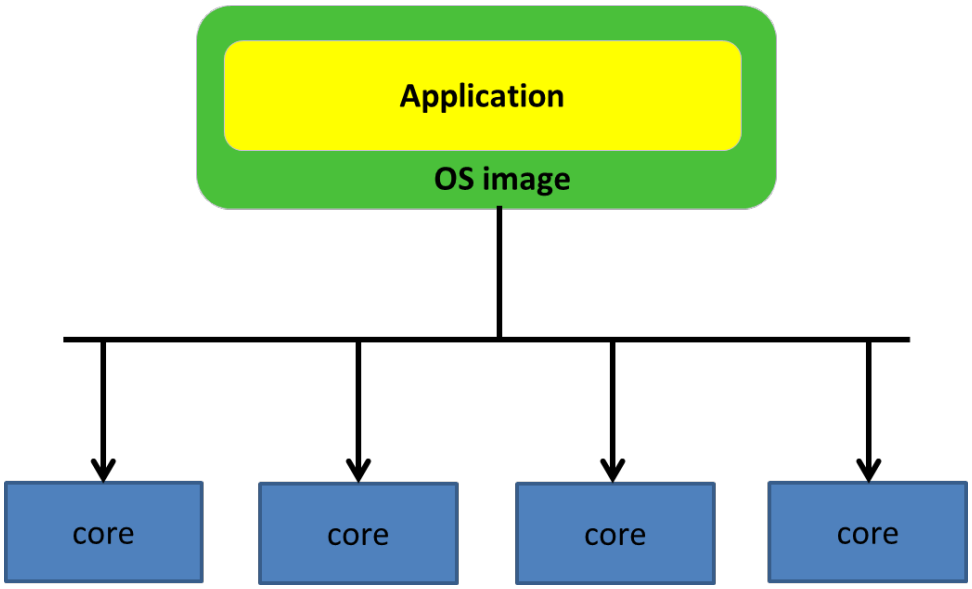


Figure 44.: Basic view of the MT setup.

The two arrangements share one main challenge, which is upper-bounding or probabilistically measuring the effects of HW inter-core interferences on application execution times. At the application level instead, when a software architecture for a multicore system needs to be designed, little-to-no help can be drawn from existing standards and reference architectures. The AUTOSAR and ARINC architectural specifications mentioned at the beginning of our dissertation conservatively deal with multicores by adopting a MP approach, thus resulting in the definition of a federated architecture. From an application perspective however this approach is no different from having a number of isolated single core computing nodes, which greatly simplifies software design, since an application is completely unaware of others possibly running on different cores. Segregation is so strong that, in the favourable conditions of the ARINC task model, run-to-completion semantics could also be adopted for the same reasons discussed in Chapter 3.

The choice of adopting the MP approach has the advantage of better controlling the computational resources embedded in a platform, but draws minimal benefit from the potential of a real multiprocessor system. The MT approach instead is certainly the most promising in terms of achievable performance as it allows, at least in theory, to fully utilise the available system resources, thus exploiting the computing power of a multicore architecture. Although this is what was originally expected when the number of processors increased, a number of issues at HW and OS level complicate real-world embrace of the MT philosophy. These issues have limited the penetration of

multicores in high-criticality industrial domains so far which explains the interest and presence of all leading manufacturers in today's cutting-edge research on multicore systems design and analysis.

The upcoming discussion conceptually follows the distinction between the MP and MT setups we gave above. In Section 4.2.1 we consider again the PROARTIS avionics case study to see how similar results to those obtained for the single-core setting can be replicated on a multiprocessor platform under a MP setup.

We then focus on the OS to study how the goal of providing time composability at this level becomes dramatically more complex in a multiprocessing environment under the multithreaded (MT) computing style. Nonetheless we look at the MT setup as a less conservative and more promising paradigm for multicore adoption. We do so for two reasons: first, the MT paradigm has arguably great potential to satisfy the increasing computational demand of real-time systems, without incurring the significant inefficiency (i.e. resource over-provisioning and low utilisation) introduced by the MP setup. Secondly, the MT paradigm poses the toughest challenges from a research point of view in terms of composability at every architectural layer: for instance, apart from the issues related to shared HW resources which undermine a system's timing behaviour, leveraging the potential of parallel execution is a promising technique to be investigated at the application level.

4.2.1 *The Avionics Case Study on Multicore*[2]

This section presents the results obtained by PROARTIS [172] for the probabilistic timing analysis of the FCDC application presented in Section 3.4.4, now running on a partitioned multicore architecture, similar to that presented in Figure 42 of Section 4.1. In this setting, multiple instances of the application run on segregated cores and take advantage of the time-composable TiCOS services, which we guarantee to cause bounded interference as in the single core scenario. Residual disturbance can arise from contention on shared HW resources, specifically the bus and the memory controller.

Approach and Setup

For the multicore case study the ARINC 653 Flight Control Data Concentrator (FCDC) application presented in Section 3.4.4 is considered. Additionally, to make the setting

2 Contributions to this section are taken from [172], authored by Barcelona Supercomputing Center.

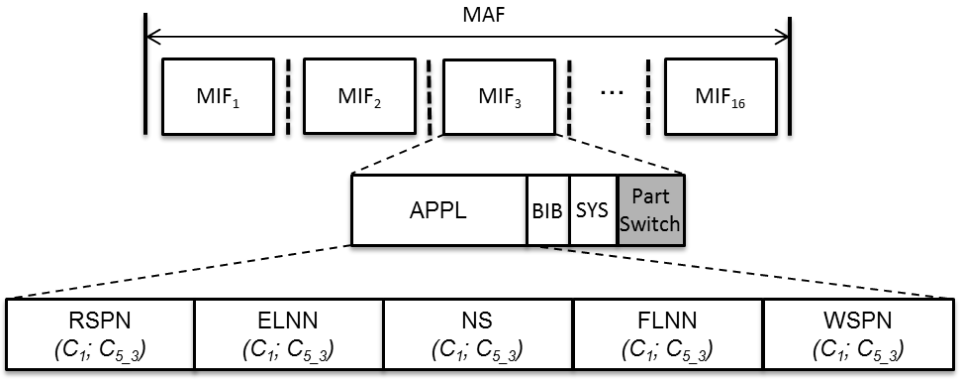


Figure 45.: WBBC A653 APPL processes (APPLICATION) executed within the third MIF.

more realistic, a second application called Weight and Balance Back-up Computation (WBBC) was selected to run in parallel with FCDC.

The WBBC application computes an independent estimation of the centre of gravity (CG) of the aircraft and generates warnings and caution functions in case anomalies are observed. Moreover, it supplies to the whole aircraft computers, weight and CG values that are used for monitoring and backup purposes. The application is composed of three main software components:

- *APPL*. Functional normal mode logic, cyclic acquisitions and transmissions of external I/Os and sequence I/Os acquisitions and emissions with pure algorithmic processing.
- *BIB*. Library of automated generated symbols used by APPLICATION A653 process.
- *SYSTEM*. It manages initialisation of WBBC software at A653 level (creation of resources such as processes, I/Os API ports, initialisation of internal WBBC variables, ...).

The APPL software component represents more than 80% of the WBBC code and is composed of the same A653 processes of FCDC application, i.e. RSPN, ELNN, NS, FLNN and WSPN. Moreover, similarly to FCDC, WBBC is scheduled within a Major Frame composed of 16 Minor Frames: Figure 45 shows the processes activated within the third MIF. At the end of WBBC application a slot is reserved for *partition switch* in which all deferred TiCOS operations are performed.

Sticking to the MP multicore setup, the two applications under study run on top of identical replicas of the PROARTIS operating system TiCOS, which provides the

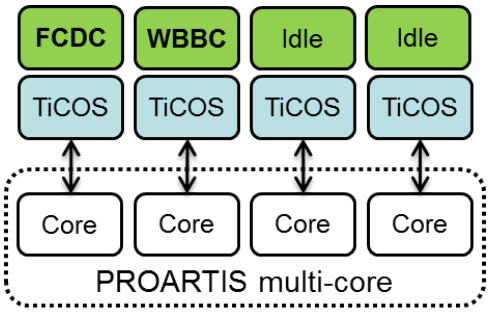


Figure 46.: Block diagram of the system configuration considered in the multicore phase.

ARINC 653 API services required by both applications with a steady timing behaviour and zero disturbance properties. Figure 46 shows a block diagram of the system configuration used for the experiments presented in this section. Each core executes an independent image of TiCOS in which a ARINC 653 partition is executed. No data is shared among cores. Because the MP design implicitly guarantees *horizontal time composability* (i.e. among applications running on different cores), the pWCET estimate of each application can be calculated in isolation, i.e. having no information about the execution on other cores, and stay valid upon the composition of those individual units into the whole system.

Results

The step of PROARTIS MBPTA procedure depicted in Figure 6 of Section 2.5.2 were applied to the FCDC NS function and the WBBC partition switch executed in the PROARTIS multicore processor. For the WBBC application, end-to-end measurements of the *partition switch* have been considered (marked in grey in Figure 45): this is mandatory to show how the I/O pre-fetch and post-write activities performed by TiCOS at these points to improve composability can be probabilistically analysed. Randomisation is introduced within the bus and memory controller to guarantee that observed execution times fulfil the properties required by MBPTA. This is confirmed by Table 12, which shows the results obtained applying the same i.i.d. tests presented in section 3.4.4 to the execution of FCDC and WBBC alone (labelled as *FCDC NS alone* and *WBBC Part alone* respectively), i.e. when all other cores are idle, and to the execution of FCDC and WBBC simultaneously on different cores (labelled as *FCDC NS + WBBC* for FCDC NS execution time measurements and *WBBC Part + FCDC* for WBBC partition switch execution time measurements). Table 13 reports instead the minimum number of runs (MNR) and the amount of time (in seconds) required to

A653 Application	Indep.	Identically Distr. ($m = 50/m = 100$)	Both test passed ?
FCDC NS Alone	0.20	0.62/0.48	Ok
FCDC NS + WBBC	0.34	0.69/0.54	Ok
WBBC Part Alone	0.25	0.75/0.60	Ok
WBBC Part + FCDC	0.37	0.57/0.55	Ok

Table 12.: Independence and identical distribution tests results.

	MNR	time
FCDC NS	300	36 s
WBBC Part	250	15 s

Table 13.: Minimum number of runs (*MNR*) and required analysis time for FCDC NS and WBBC partition switch.

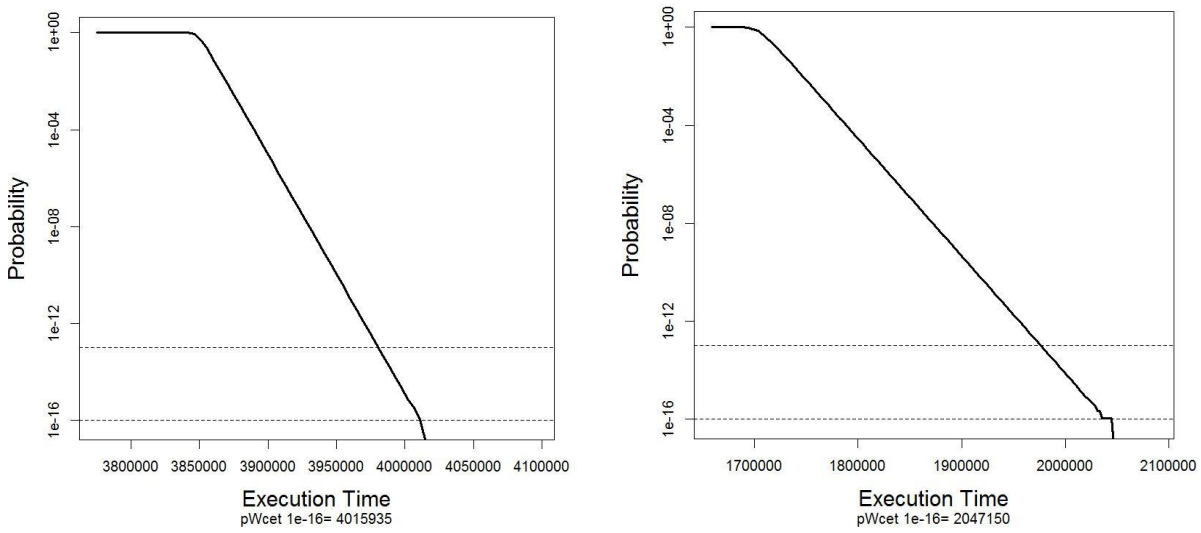
perform the timing analysis of the FCDC NS process and the WBBC partition switch (FCDC NS and WBBC part respectively).

Figure 47(a) and 47(b) show the pWCET estimates with MBPTA-EVT of FCDC NS process and WBBC partition switch respectively, when executing each application in isolation. We notice how the shape of the plot in Figure 47(a) related to the NS function is identical to the plot for the correspondin FUNC3 function in the single-core experiment shown in Figure 29(c) in Section 3.4.4; just an increasing constant factor is present, due to the contention on the shared bus and the higher levels of the memory hierarchy. Similarly, the plot in Figure 47(b) corresponds to the result reported in Figure 29(d).

The observations above support our claim that the composability results achieved on single cores can be somehow replicated on a partitioned multicore. The use of random arbitration policies in the shared bus and the memory controller, and the use of cache partitioning mechanisms makes the pWCET estimate of each application being independent of the execution on other cores. As a result, the MP multicore setup allows the use of the MBPTA-EVT techniques developed for the uniprocessor setup without any single modification, thanks to the provided horizontal time composability property.

4.2.2 *Challenges for a Multicore OS*

In Section 3.1 we characterised time composability in single cores as a combined product of the principles of *zero disturbance* and *steady timing behaviour*. In our under-



(a) (b)

Figure 47.: pWCET estimate distributions of FCDC NS process (a) and WBBC partition switch (b) when executing on the PROARTIS multicore processor.

standing, thanks to its privileged position in between hardware and applications, a real-time operating system has the responsibility of remedying to unpredictable HW behaviour by provisioning services with timing guarantees. Ideally, the goal should be no different in a multicore setting, although the challenges to address are likely to be even tougher.

In Section 4.1.1 we already showed how the sources of disturbance inevitably increase as a consequence of the growing number of shared HW resources and their reciprocal interactions. However, once undesired HW effects are characterised and under the favourable assumptions of the MP model of computation, results comparable to those obtained on a uniprocessor are achievable even on a multicore, as demonstrated in Section 4.2.1.

Steady behaviour is far more difficult to provide instead under a MT setting, where tasks are allowed to migrate across cores and to communicate to each other, even when deployed simultaneously on different processing units. Additionally, the disrupting effects of synchronous events are much amplified in a multicore environment as a consequence of inter-core dependencies. For example, the release of a sporadic event on one processor may cause tasks migrations across cores. In Section 3.2 we identified a minimal set of desirable properties, hence potential areas of intervention, that cannot be disregarded when trying to inject composability on a full-fledged single-core RTOS; we now reason on the new challenges when moving to a multicore MT setting.

1. *Non-intrusive time management:* As already mentioned, the way the progression of time is managed by the operating system should have no side effects on the timing behaviour of the applications running on top of it. For this reason we abandon the use of an intrusive tick-based time management, which periodically triggers possibly unnecessary scheduling operations, in favour of programmable timers. Moreover, the problem of correctly dispatching interrupts (and time interrupts in general) may be not trivial to attack on a multicore platform: in a MP setup it may be sufficient to replicate interrupts coming from a unique HW source on all cores, where identical replicas of the OS are executed. This is also the case of MT where the OS is statically pinned to one core. Instead, if the OS is allowed to migrate across cores as well, the processor where the OS temporarily resides must be identified to properly acknowledge the fired interrupt. This has clear implications in terms of disturbance on the core-specific HW resources involved in this process and in the responsiveness of the system.

2. *Constant-time scheduling primitives:* Most of the RTOS execution time is typically spent on task scheduling and dispatching activities, therefore reducing the variability of such primitives is essential in a composable RTOS. Aside from making the execution time of these primitives independent from the number of tasks in the system, the problem of globally assigning applications to processing units assumes great importance to achieve high system utilisation. Unfortunately, this entails the possibility of bearing task migrations among cores, which completely invalidates the working set of moved applications and causes therefore great disturbance to their timing behaviour. This is the focus of Section 4.3.

3. *Composable inter-task communication:* In our discussion on single core we defined inter-task communication as covering the broad area of interactions with hardware and software resources, including I/O, communication and synchronisation. There, the problem of providing a time-composable communication subsystems was reduced to the one of providing controlled access to shared resources. Since in a multiprocessor most resources are globally shared and tasks may be allowed to communicate across cores, a global resource locking mechanism may be needed. Its impact in terms of blocking time caused to all contenders needs to be carefully evaluated to avoid the introduction of unbounded sources of jitter in the system.

4. *Selective hardware isolation:* The idea of isolating the history-dependent hardware state as a means to preventing RTOS-induced disturbance is very similar in

multiprocessor and single-core architectures. Clearly, the emphasis placed on the former scenario is stronger as a consequence of the even higher penalty paid for HW interferences, as already discussed. Depending on the mechanisms possibly provided to the purpose, the RTOS should exploit the available features by means of ad-hoc low-level interfaces.

Whether and to what extent the OS-level solutions we adopted in the single-core setting can drive the quest for time composability in the multiprocessor case largely depends on how well they can fit with the properties of the scheduling algorithm of choice. The reason of that dependence stems from the trade-off brought forward by multiprocessor scheduling: seeking high schedulable utilisation intrinsically requires task migration, which is arch-enemy to time composability; banning task migration pollutes the system design problem with bin-packing that cannot contemplate functional aggregation. For this reason we focus the last leg of our research on the search for a multiprocessor scheduling solution that achieves a good balance in the way of time composability.

4.3 MULTIPROCESSOR SCHEDULING

4.3.1 *Background*

The problem of multiprocessor scheduling has been extensively investigated, both because of its relevance in real-time systems and of the challenges it presents. Given a task set τ and a scheduling algorithm S, the *schedulability* problem can be defined as finding a schedule for τ constructed with S that respects all the deadlines of the jobs released by the tasks in τ. Research in scheduling theory is therefore devoted to finding such an algorithm S, which is usually required to be *optimal*, i.e. to *always* find a schedule for τ whenever it exists, according to the deadline constraints imposed to its jobs (in this case the task set is said to be *feasible*). The performance of a scheduling algorithm is usually measured in terms of its worst-case *utilisation bound*, that is the minimum utilisation of any task set schedulable with it, and its *approximation ratio*, i.e. its performance when compared to a known optimal scheduling algorithm for the specific constraints on the scheduling problem.

Referring to the taxonomy given in [173], the multiprocessor scheduling problem can be intended as an attempt to solve two distinct problems, the *allocation problem* to determine on which processor a task should execute and the *priority problem* to

determine in which respective order the jobs of different tasks should be run. The latter problem is very similar to its single-core counterpart and in its respect scheduling algorithms can be classified into *fixed task priority*, where priorities are statically assigned to tasks and never change, *fixed job priority*, when the different jobs of a task may be statically assigned different priorities and *dynamic priority*, in case each job is allowed to take different priority levels during execution.

The allocation problem instead is the distinctive trait of multicore scheduling and distinguishes scheduling algorithms into *partitioned* and *global*.

PARTITIONED SCHEDULING. In partitioned approaches each task is allocated to a processor selected offline and no migration is allowed during execution (Figure 48). Hence, the task set is partitioned once by design and then a uniprocessor scheduling algorithm is used to schedule tasks within each core, possibly using different algorithms for different processors. A representative of this class is Partitioned EDF (P-EDF), which schedules tasks with EDF on each processor independently, thus on the subset of tasks assigned for execution to one specific processing unit. Although the optimality of EDF has been proven on uniprocessors, this property does not hold anymore for P-EDF in a multicore setting, essentially because the local best scheduling choice has no direct correlation with the global best decision to be taken. In fact, the problem of partitioned multicore scheduling can be seen as a bin packing problem, which is known to be NP-hard [174]; and even when a solution can be found, the process of partitioning the task set is limited by the performance of the chosen bin-packing algorithm. The focus of existing research has therefore moved to finding the best algorithm to partition a specific task set, giving the rise to plenty of algorithms, usually based on Rate Monotonic [175] or EDF [174], which are characterised by relative improvements on their utilisation bound.

Partitioned approaches have the advantage of paying no penalty in terms of migration cost, which is in fact prohibited, and of dealing with separate task queues per processor, thus incurring more lightweight overhead when accessing and updating those data structures. Partitioned scheduling perfectly fits the MP paradigm illustrated in Section 4.2 and it is indeed appealing from an industrial perspective, especially where the scarcity of best practices tested on-the-field pushes towards conservative choices.

GLOBAL SCHEDULING. With global scheduling, tasks are allowed to migrate across processors (Figure 49), thus at any point in time the m jobs with the highest priority

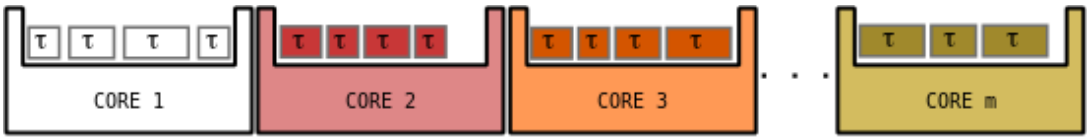


Figure 48.: Partitioned multiprocessor scheduling.

in the system are executed on the m available processors. As an example, the global version of the EDF algorithm called Global EDF (G-EDF) schedules tasks among *all* available processors according to the EDF discipline. Typically, global approaches are capable of achieving higher system utilisation, as a consequence of the possibility of migrating tasks across cores to benefit from the available spare capacity in the system, generated whenever tasks execute for less than their expected worst case.

Although from a historical perspective global scheduling has been sometimes neglected due to the so-called Dhall's effect [176], renewed interest has appeared after the observation that the necessary conditions for the Dhall's effect to occur can be precisely characterised [177][178] and therefore countered or even avoided in a real-world usage scenario. The seminal work by Baruah [179][180] gave a boost to research on global scheduling techniques by observing that optimality is in fact achievable through *fairness*, that is by dividing the timeline into equally long slices and assigning processing time to tasks within each slice according to their utilisation demand. Unfortunately, the algorithm originating from this intuition, called P-Fair, can be hardly implemented in practice, because an infinitesimally small size of time quanta needs to be considered to achieve optimality. Other algorithms trying to enforce fairness include PD [181], PD^2 [182], BF [183] and DP-Fair with its incarnation DP-Wrap [184][185]. Other algorithms instead rely on the concept of *laxity* to take their scheduling decisions, such as LLREF [186] and EDZL [187]. Recently, Nelissen et al. [188][189][190] and Regnier et al. [191][43][192] demonstrated that enforcing fairness in scheduling decisions is not a necessary requirement for optimal multicore scheduling.

From an industrial perspective the promises of global scheduling are very attractive, since achieving full system utilisation defeats over-provisioning and the incremental system development is supported with minimal requalification effort. However, the performance penalty paid in the occurrence of job migrations and the impossibility of precisely predicting these events have discouraged the adoption of global scheduling techniques. Additionally, depending on the specific case, some of the assumptions and

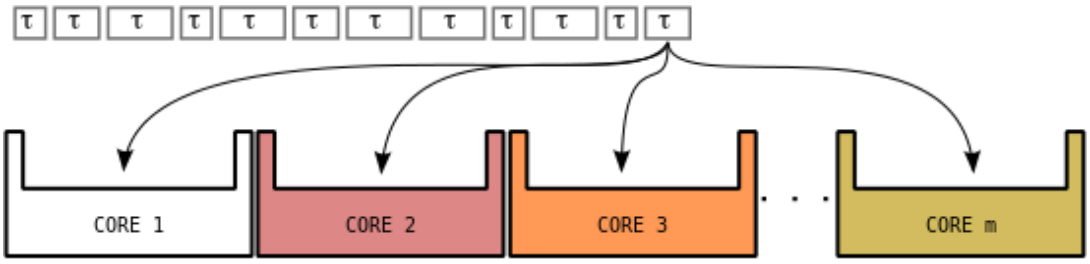


Figure 49.: Global multiprocessor scheduling.

requirements imposed by global schedulers to ensure optimality may be impractical or even impossible to implement in the real world.

As neither partitioned nor global scheduling are fully satisfactory because of practical limitations, hybrid scheduling approaches have emerged in between trying to leverage the strength of both families and mitigate their drawbacks. **Semi-partitioned** algorithms like EKG [193] adopt a partitioned approach for most of the tasks in the task set by assigning them to a specific processor, and let a smaller subset globally migrate as long as the packing heuristic fails to allocate them to one single processor. **Clustered** algorithms like Clustered EDF (C-EDF) consider instead shared HW resources to define clusters of processors where task migration is allowed within one cluster and forbidden across different clusters; considering in particular memory hierarchies, this approach permits to lower incurred migration overhead. **Hierarchical** scheduling algorithms like NPS-F [194] and RUN [43][192] partition tasks into higher-level structures (called servers or supertasks) which are scheduled globally, while a uniprocessor scheduling algorithm is used within each server. This makes it possible to solve the scheduling problem at a coarser-grained level so that an optimal solution is likely to be found more easily.

In our study on time composability in high-criticality systems, when actual implementation and deployment in a real-world industrial context cannot be ignored, we regard hybrid methods as the most promising solution to the multicore scheduling problem. In particular we focused or attention on RUN, which takes benefit from partitioning of task aggregates in a first offline step, and then allows task migrations during execution. RUN has proven to be very efficient in terms of incurred preemptions and migrations and is briefly discussed in the next section.

RUN

The Reduction to UNiprocessor (RUN) algorithm is an optimal technique for multicore scheduling which has recently attracted the attention of researchers for its out-of-the-box approach taken on the problem. Together with U-EDF [188][189][190], RUN is the only optimal algorithm to guarantee optimality without explicitly resorting to a notion of fairness. RUN consists of two steps: first a *reduction tree* for the task set is created in a preliminary offline phase. Then at run time tasks on the leaves of the reduction tree are globally scheduled across the m processors by traversing the tree downwards from the root.

The simple observation behind RUN is that scheduling a task's execution time is equivalent to scheduling its idle time. This approach named *dual scheduling* had already been investigated in a few previous works [181][195][185][196]. The dual schedule of a set of tasks $\mathcal{T}$ consists in the schedule produced for the dual set $\mathcal{T}^*$ defined as follows:

Definition 1 (Dual task)**.** *Given a task τ_i with utilization $U(\tau_i)$, the dual task τ_i^* of τ_i is a task with the same period and deadline of τ_i and utilisation $U(\tau_i^*) \stackrel{\text{def}}{=} 1 - U(\tau_i)$.*

Definition 2 (Dual task set)**.** *$\mathcal{T}^*$ is the dual task set of the task set $\mathcal{T}$ if (i) for each task $\tau_i \in \mathcal{T}$ its dual task τ_i^* is in $\mathcal{T}^*$, and (ii) for each $\tau_i^* \in \mathcal{T}^*$ there is a corresponding $\tau_i \in \mathcal{T}$.*

The definitions above suggest that scheduling the tasks in $\mathcal{T}^*$ is equivalent to schedule the idle times of the tasks in $\mathcal{T}$, therefore a schedule for $\mathcal{T}$ can be derived from a schedule for the dual $\mathcal{T}^*$. Indeed, if τ_i^* is running at time t in the dual schedule, then τ_i must stay idle in the actual schedule of τ (also called *primal* schedule). Inversely, if τ_i^* is idle in the dual schedule, then τ_i must execute in the primal schedule.

Example 1. *Figure 50 shows an example of the correspondence between the dual and the primal schedule of a set $\mathcal{T}$ composed of three tasks executed on two processors. In this example the three tasks τ_1 to τ_3 have a utilization of $\frac{2}{3}$ each, implying that their dual tasks τ_1^* to τ_3^* have utilizations of $\frac{1}{3}$. The dual task set is therefore schedulable on one processor while the primal tasks τ_1 to τ_3 need two processors. Whenever a dual task τ_i^* is running in the dual schedule, the primal task τ_i remains idle in the primal schedule; inversely, when τ_i^* is idling in the dual schedule then τ_i is running in the primal schedule. Note that the number of processors does not always diminish in the dual schedule. This actually depends on the utilization of the tasks in the particular task set.*

Lemma 1 explicits the relation existing between the utilisation of a task set and the utilisation of its dual:

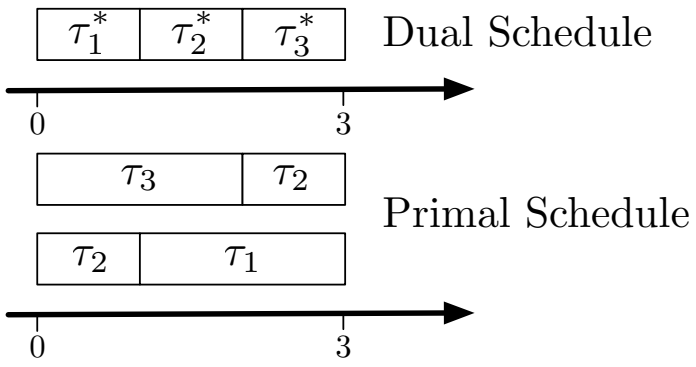


Figure 50.: Correspondence between the primal and the dual schedule on the three first time units for three tasks τ_1 to τ_3 with utilizations of $\frac{2}{3}$ and deadlines equal to 3.

Lemma 1. *Let $\mathcal{T}$ be a set of n periodic tasks. If the total utilization of $\mathcal{T}$ is $U(\mathcal{T})$, then the utilization of $\mathcal{T}^*$ is $U(\mathcal{T}^*) \stackrel{\text{def}}{=} n - U(\mathcal{T})$.*

Hence, if $U(\mathcal{T})$ is an integer then $\mathcal{T}$ and $\mathcal{T}^*$ are feasible on $m = U(\mathcal{T})$ and $m^* = n - U(\mathcal{T})$ processors, respectively. Consequently, in systems where n is small enough to get

$$(n - U(\mathcal{T})) < m \tag{3}$$

the number of processors can be reduced – and so the complexity of the scheduling problem – by scheduling the dual task set $\mathcal{T}^*$ instead of $\mathcal{T}$. In Example 1, we have $U(\mathcal{T}) = 2$ thereby implying that $\mathcal{T}$ is feasible on two processors. However, we have $n = 3$, which leads to $U(\mathcal{T}^*) = n - U(\mathcal{T}) = 1$. Therefore, the dual task set $\mathcal{T}^*$ is feasible on one processor.

RUN resorts to the concept of "server" which can be related to the supertasking approach described in [197]; as a first step to build RUN reduction tree each individual task τ_i is wrapped into a server S_i with the same utilisation and deadline. To enforce Expression 3 being true, RUN then uses a bin-packing heuristic to pack servers into as few bins as possible; each bin should have utilisation smaller than or equal to 1, to benefit from duality in further reduction steps.

Definition 3 (Pack operation/Packed server)**.** *Given a set of servers $\{S_1, S_2, \ldots, S_n\}$, the* PACK *operation defines a server S with utilisation $U(S) = \sum_{i=1}^{n} U(\tau_i)$. S is called the* packed server *of $\{S_1, S_2, \ldots, S_n\}$.*

In the following discussion we will use the notation $S \in B$ to specify that server S is packed into server B or that server S is included in the subtree rooted in B,

interchangeably. RUN then leverages the notion of duality to schedule idle times of existing servers.

Definition 4 (Dual operation/Dual server)**.** *Given a server S, the* DUAL *operation defines a server S^* with utilisation $U(S^*) = 1 - U(S)$. S^* is called the* dual server *of S.*

The process of constructing the tree then proceeds by iteratively applying the two operations above alternatively, which define new levels in the tree.

Definition 5 (Reduction operation)**.** *The operation consisting in successively applying a* DUAL *and a* PACK *operation is called* REDUCTION .

Definition 6 (Reduction level)**.** *Named $\mathcal{S}^0$ the* PACK *of the initial servers $\{S_1, S_2, \ldots, S_n\}$, the application of the* REDUCTION *operation to the set of servers $\mathcal{S}^i$ at level i defines a new set of servers $\mathcal{S}^{i+1}$ at level $i+1$ in the reduction tree. The intermediate level between i and $i+1$ (i.e. when the* DUAL *operation has been applied but the* PACK *operation has not) is indicated by i^* (see Figure 51 as an example).*

By recursively applying the reduction operation, [43] proved that the number of processors eventually reaches one. Hence, the initial multiprocessor scheduling problem can be reduced to the scheduling of a uniprocessor system. More formally, this means that $\exists l, S_k^l : |\mathcal{S}^l| = 1 \wedge U(S_k^l) = 1$, where $\mathcal{S}^l$ is the set of servers at level l and $S_k^l \in \mathcal{S}^l$. In fact, at every application of the reduction operation (i.e. at each level of the reduction tree) the number of servers is reduced by at least one half, i.e. $|\mathcal{S}^{l+1}| \leq \left\lceil \frac{|\mathcal{S}^l|}{2} \right\rceil$.

Example 2. *Figure 51 shows an illustrative reduction tree for a task set comprising 6 tasks, represented as leaves, and 17 servers, represented as internal nodes. Let $\mathcal{T}$ be composed of 6 tasks τ_1 to τ_6 such that $U(\tau_1) = U(\tau_2) = U(\tau_3) = U(\tau_4) = 0.6$ and $U(\tau_5) = U(\tau_6) = 0.3$. The total utilization of $\mathcal{T}$ is 3 and $\mathcal{T}$ is therefore feasible of 3 processors. If we directly applied duality on $\mathcal{T}$, we would have $U(\tau_1^*) = U(\tau_2^*) = U(\tau_3^*) = U(\tau_4^*) = 0.4$ and $U(\tau_5^*) = U(\tau_6^*) = 0.7$. Hence, $U(\mathcal{T}^*)$ would be equal to 3 and we would gain nothing by scheduling the dual task set $\mathcal{T}^*$ instead of $\mathcal{T}$. Therefore, as shown in Figure 51, RUN first applies the* PACK *operation which creates 5 servers S_1^0 to S_5^0 such that $S_1^0 = \{\tau_1\}$, $S_2^0 = \{\tau_2\}$, $S_3^0 = \{\tau_3\}$, $S_4^0 = \{\tau_4\}$ and $S_5^0 = \{\tau_5, \tau_6\}$. Hence, $U(S_1^0) = U(S_2^0) = U(S_3^0) = U(S_4^0) = U(S_5^0) = 0.6$ and $U(S_1^{0*}) = U(S_2^{0*}) = U(S_3^{0*}) = U(S_4^{0*}) = U(S_5^{0*}) = 0.4$ (where $U(S_i^l)$ and $U(S_i^{l*})$ are the utilizations of S_i^l, the i^{th} server of reduction level l, and its dual S_i^{l*}, respectively). The set of dual servers now has a total utilization $U(S^{0*}) = 2$ implying that this set of dual servers in feasible on 2 processors. If we apply once more the* PACK *and* DUAL *operations, we*

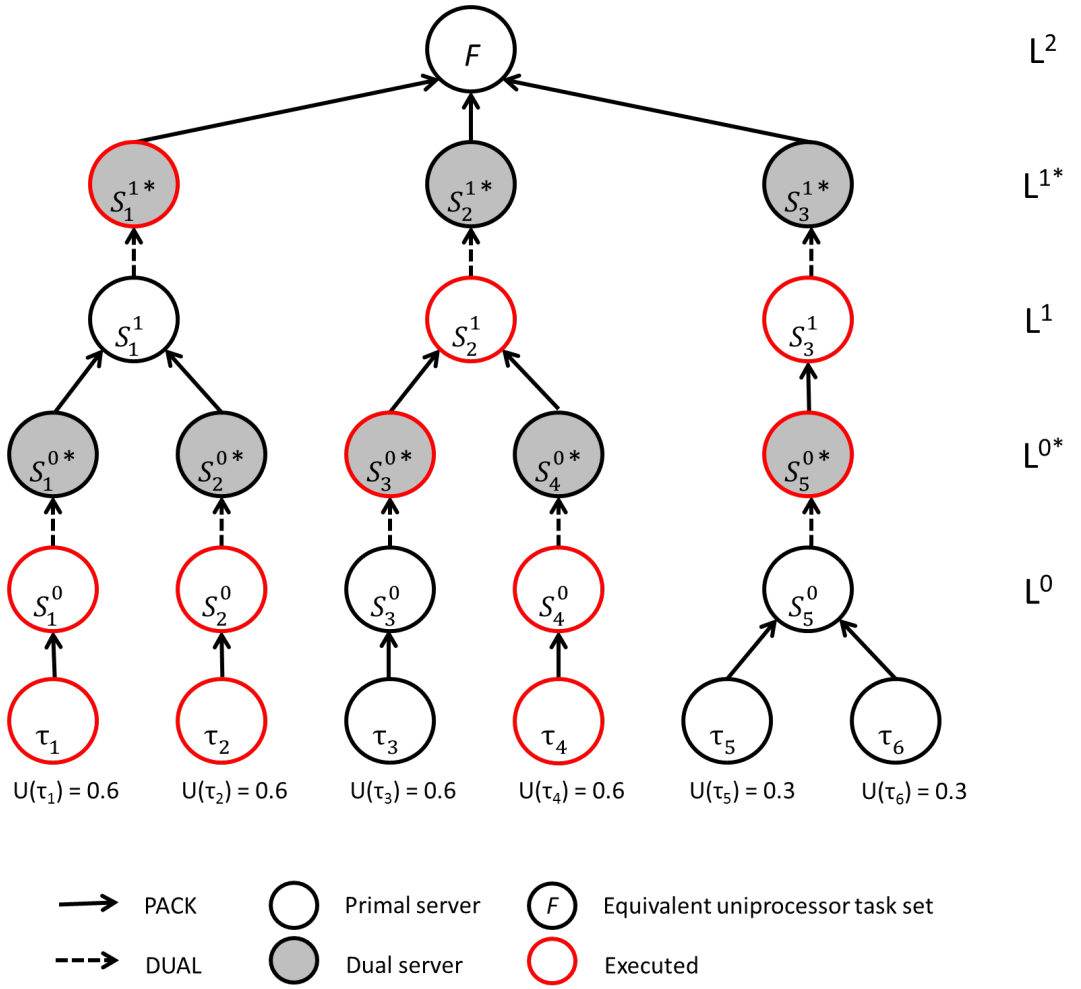


Figure 51.: RUN reduction tree for a task set of 6 tasks.

have three new servers $S_1^1 = \{S_1^{0*}, S_2^{0*}\}$, $S_2^1 = \{S_3^{0*}, S_4^{0*}\}$ *and* $S_3^1 = \{S_5^{0*}\}$ *with utilizations* $U(S_1^1) = U(S_2^1) = 0.8$ *and* $U(S_3^1) = 0.4$. *The dual servers of this second reduction level therefore have utilizations* $U(S_1^{1*}) = U(S_2^{1*}) = 0.2$ *and* $U(S_3^{1*}) = 0.6$. *Consequently, the total utilization of the dual set of servers of this second reduction level can be packed into* $S_1^2 = \{S_1^{1*}, S_2^{1*}, S_3^{1*}\}$ *with* $U(S_1^2) = 1$, *implying that they are feasible on a uniprocessor platform.*

The *online* phase of RUN consists in deriving the schedule for $\mathcal{T}$ from the schedule constructed with EDF at the uppermost reduction level (i.e. for the equivalent uniprocessor system). At any time t of execution any server S is characterized by a *current deadline* and a given *budget*.

Definition 7 (Server deadline in RUN). *At any time t, the deadline of server S_k^l on level l is given by*

$$d_k^l(t) \stackrel{\text{def}}{=} \min_{S_i^{l-1} \in S_k^l} \{d_i^{l-1}(t)\}$$

and the deadline of the dual server $S_k^{l^*}$ on level l^* is given by

$$d_k^{l^*}(t) \stackrel{\text{def}}{=} d_k^l(t)$$

Deadline and budget are related in that whenever a server S_k^l reaches its deadline, it replenishes its budget by an amount proportional to its utilisation $U(S_k)$, as follows:

Definition 8 (Budget replenishment in RUN). *Let $R(S_k^l) \stackrel{\text{def}}{=} \{r_0(S_k^l), \ldots, r_n(S_k^l), \ldots\}$ the time instants at which S_k^l replenishes its budget, with $r_0(S_k^l) = 0$ and $r_{n+1}(S_k^l) = d_k^l(r_n(S_k^l))$. At any instant $r_n(S_k^l) \in R(S_k^l)$ server S_k^l is assigned an execution budget $bdgt(S_k^l, r_n(S_k^l)) \stackrel{\text{def}}{=} U(S_k^l) \times \big(r_{n+1}(S_k^l) - r_n(S_k^l)\big)$.*

Note that the budget of a sever is decremented with its execution. That is, assuming that the budget of S_k^l was not replenished within the time interval $(t_1, t2]$,

$$bdgt(S_k^l, t2) = bdgt(S_k^l, t1) - exec(S_k^l, t_1, t_2) \tag{4}$$

where $exec(S_k^l, t_1, t_2)$ is the time S_k^l executed during $(t_1, t2]$.

The schedule for $\mathcal{T}$ is finally built by applying the two following rules at each reduction level:

> **Rule 1.** *Each server of reduction level l which is running in the primal schedule at time t executes its component server (or task) with the earliest deadline in the dual schedule at reduction level $(l-1)^*$;*

> **Rule 2.** *Each server of reduction level $(l-1)^*$ which is not running in the dual schedule is executed in the primal schedule at level $l-1$ and inversely, each server which is running in the dual schedule is kept idle in the primal schedule.*

Example 3. *Let us take the same task set $\mathcal{T}$ from Example 2 depicted in Figure 51 and let us assume that each of the six tasks $\tau_i \in \mathcal{T}$ has an active job at time t such that $d_1(t) < d_2(t) < d_3(t) < d_4(t) < d_5(t) < d_6(t)$. Since a server inherits the deadlines of its component tasks,*

each server S_i^0 (with $1 \leq i \leq 5$) on the first reduction level L^0 inherits the deadline $d_i(t)$ of the corresponding task τ_i. At the second reduction level L^1, the deadline of S_1^1 is equal to $d_1(t)$ while the deadline of S_2^1 is $d_3(t)$ and S_3^1 has the same deadline as τ_5. Because a dual server has the same deadline as the corresponding primal server, if we execute EDF on the set of dual servers at level 1^*, the dual server S_1^{1*} is chosen to be executed at time t (see Figure 51). This means that both S_2^1 and S_3^1 should be running in the primal schedule of the second reduction level. Applying EDF in each of these servers, we get that S_3^{0*} and S_5^{0*} must be running in the dual schedule of the first reduction level. Therefore, S_3^0 and S_5^0 must stay idle while S_1^0, S_2^0, S_4^0 must be executed in the primal schedule of the first reduction level. Consequently, it results that τ_1, τ_2 and τ_4 must execute on the platform at time t.

It was proven in [43, 191] that the online part of RUN has a complexity of $O(jn\log(m))$ when there is a total of j jobs released by n tasks on m processors within any interval of time.

Although its authors classified it as semi-partitioned, the algorithm presents the traits of different categories: the use of supertasks to reduce the size of the problem certainly recalls hierarchical algorithms. The idea of packing tasks into servers borrows instead from purely partitioned approaches, even though partitioning is not performed per processor but rather among servers and is not as critical for the performance of the algorithm as in other partitioned approaches (e.g. P-EDF). Finally, tasks are not pinned to processors and are free of migrating to any core when scheduling decisions are taken, as in global scheduling techniques. As mentioned earlier, RUN does not explicitly deal with the notion of *fairness* in a strong sense, i.e. it does not try to assign proportional execution to each task in the task set to generate a fluid schedule; rather, some kind of "partitioned" proportional fairness is enforced among servers on higher levels of the reduction tree. This results in a relaxed flavour of fairness, which is enforced only between consecutive time boundaries, as in BF^2 [188], represented by job deadlines. This causes significantly less scheduling overhead when compared to global scheduling algorithms like P-Fair [179][180], which also translates into a smaller number of preemptions and migrations[3].

More than just reducing the number of incurred preemptions and migrations, RUN takes on a divide-et-impera approach to the multiprocessor scheduling problem, which parallels current best practices of system development, and makes therefore

3 The authors claim to have observed an average of less than 3 preemptions per job in their simulation results and proved that the average number of preemptions per job is upper bounded by $\left\lceil \frac{3p+1}{2} \right\rceil$ when p reduction opperations are required.

its application appealing from an industrial perspective. First of all, the algorithm can gracefully adapt to changes in the task set, since just the computation of a new reduction tree is required in that event, which is however performed offline. This feature is highly desirable in settings where incrementality is a necessary requirement of the development and qualification processes, as it is the case of real-time systems. In this context incrementality is indeed the key enabler to achieve *compositional* design, which permits to express the behaviour of a system as a function of its elementary components. In this scenario RUN servers mimic functionally cohesive and independent subsystems, enclosing part of the overall system functionality, which may be allocated to a dedicated subset of system resources and analysed in isolation. While this is usually achieved by strict partitioning and not being a partitioned technique in the strong sense, RUN is capable of achieving high schedulable utilisation at run time, thus avoiding over provisioning of system resources.

Unfortunately, in its original formulation, RUN presents two major limitations that hinder its applicability in a real-world scenario:

- **Missing support for sporadic tasks.** One big issue with RUN is that it is intended only for periodic task sets. Clearly, for a scheduling algorithm to be industrially relevant, the support of asynchronous events like interrupts and sporadic task activations is a mandatory requirement.

- **Missing support for resource sharing.** The problem of resource sharing in a multicore environment, although highly critical from a real-world point of view, is often ignored in the first place when a new scheduling algorithm is defined. The reason thereof is that various forms of blocking among tasks are introduced when exclusive access to critical sections needs to be guaranteed, and this needs to be taken into account by the analysis. This is also the case of RUN, whose tasks are considered to be independent from each other. Recently, Burns and Wellings proposed a locking protocol [198] for partitioned multiprocessor systems whose response-time analysis is as easy as the analysis of the ceiling protocol in the single core case. A viable solution to this problem recently investigated at the University of Padova consists in packing those tasks sharing the same resource together within the same server, which is then made non-migrating.

In [199] it has been demonstrated that a real implementation of RUN[4] is indeed possible with reasonable performance when compared to other existing partitioned

4 The mentioned implementation has been realised on the LITMUS^{RT} extension to the Linux kernel developed by UNC [200][201].

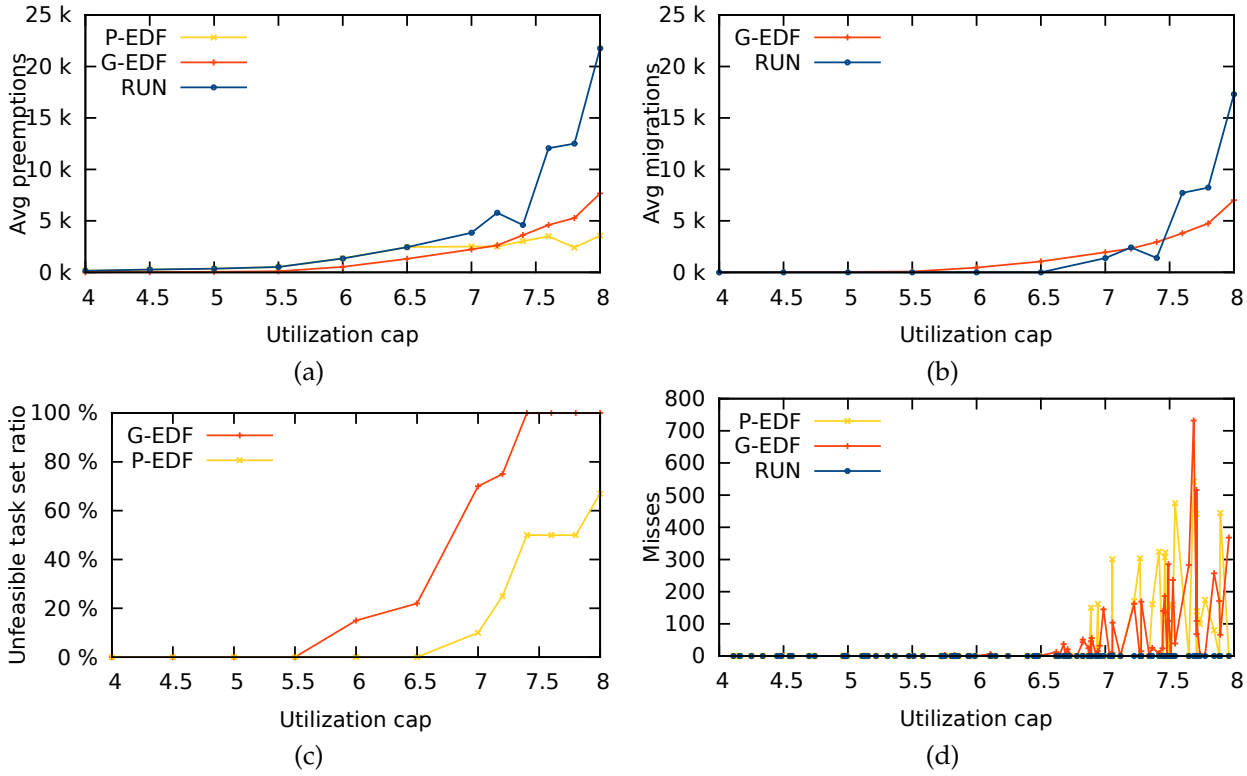


Figure 52.: Evaluation of RUN on a real implementation.

and global algorithms. Figures 52(a) and 52(b) from that work report the number of preemptions and migrations, respectively, incurred by task sets schdeuled under different policies: RUN shows comparable performance to G-EDF and P-EDF until 85% utilisation. After this threshold the number of unfeasible task sets for those algorithms increases (Figures 52(c)) causing therefore more deadline misses (Figures 52(d)), whereas RUN is capable of scheduling them up to an utilisation bound of 100% as a consequence of its optimality.

RUN represents therefore a promising solution to real-world multicore scheduling, and is worth being more deeply investigated to the purpose of countering the mentioned limitations. In particular, we are interested in extending the periodic task model considered by RUN to support sporadic task sets, which would make the algorithm more appealing to real industrial adoption. The next section presents this work.

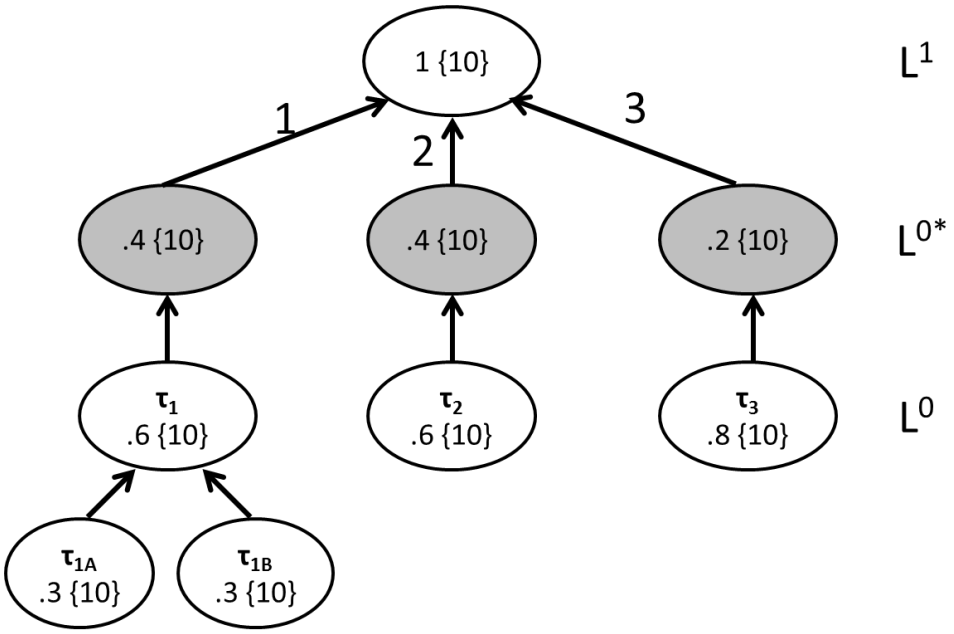


Figure 53.: Reduction tree for the task set in Example 4.

4.3.2 *SPRINT*

Preliminaries

The impossibility of scheduling sporadic task sets with RUN is a serious limitation which confines its validity to academic research and poses a serious obstacle to its adoption in a realistic context. Consider the next motivating example, where a feasible task set cannot be scheduled with RUN as a consequence of the the sporadic nature of some of its tasks.

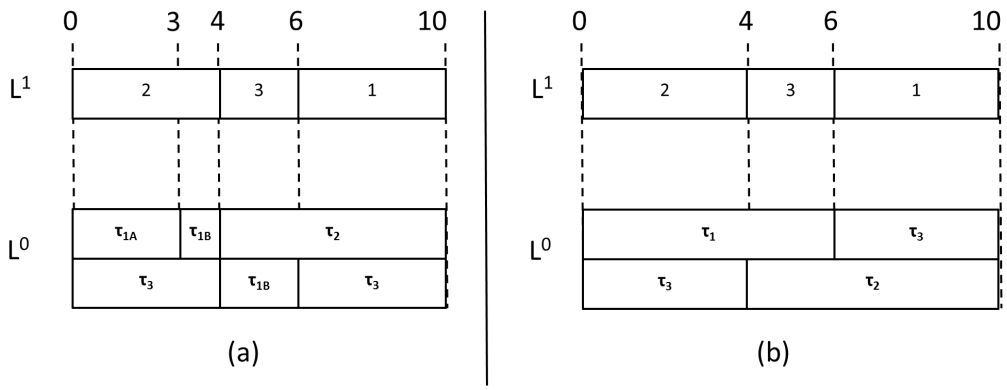


Figure 54.: Possible schedules for the task set in Example 4.

Example 4. *Consider the following task set composed by 4 tasks with implicit deadlines,* $\mathcal{T} = \{\tau_{1A} = \langle 3, 10\rangle, \tau_{1B} = \langle 3, 10\rangle, \tau_2 = \langle 6, 10\rangle, \tau_3 = \langle 8, 10\rangle\}$*, whose reduction tree is shown in Figure 53. A possible schedule constructed by RUN (Figure 54(a)) allocates tasks* τ_{1A} *to processor* π_1 *at time* $t = 0$*; however, if* τ_{1A} *were a sporadic task with arrival time* $t_{\tau_{1A}} = 1$*, then it would be impossible for RUN to execute it for the 3 units of time required, therefore missing deadline* $d_{1A} = 10$ *for task* τ_{1A}*. In this case the problem could be solved with RUN by choosing the schedule where the allocation of* τ_{1A} *and* τ_{1B} *are switched, although there is no way to capture this requirement in the original algorithm. Moreover, this patch would not solve the general problem, as evident in the case of task set* $\mathcal{T}'$ *in Figure 54(b), which is identical to task set* $\mathcal{T}$ *with the only exception that tasks* τ_{1A} *and* τ_{1B} *are now collapsed into a single task* τ_1 *with the same deadline and utilisation equal to their sum. In this case, if* τ_1 *and* τ_2 *were sporadic tasks no valid RUN schedule exists in which a late arrival of both* τ_1 *and* τ_2 *can be accommodated.*

Example 4 above highlights the main problem with RUN: the algorithm assumes that new jobs for tasks in $\mathcal{T}$ can be only released at fixed points in time, i.e. at boundaries d_i, corresponding to multiples of task periods. After jobs are released and new deadlines are propagated up to the root, a new schedule for $\mathcal{T}$ is computed by traversing the reduction tree top-down, i.e. by first finding a solution to the uniprocessor problem of scheduling jobs released by servers at level n. By applying Rules 1 and 2 the process proceeds downwards eventually producing a schedule for the actual tasks on the leaves. However, no clue is given to the scheduling process on how to select servers at intermediate levels and the corresponding branches in the tree to favour/ignore the scheduling of specific tasks, which may not have been activated yet. Additionally, we observe that RUN needs to assume full system utilisation when the reduction tree is created to guarantee the convergence of the reduction process. This means that in a system with m processors the cumulative utilisation of the task set T to be scheduled must be $U_{TS} = \sum_{\tau_i \in TS} \frac{C_i}{T_i} = m$. Even in case this condition is not met, it can be easily achieved by adding some dummy tasks to fill system utilisation up to m. We might equally consider those tasks to be sporadic and inactive in the system until their activation event is fired.

It is therefore both interesting and promising to investigate a possible generalization of the algorithm, which could account for dynamic arrival times and handle sporadic task sets. Since it was proven in [43] that the structure of RUN reduction tree based on the notion of dual schedule is the key enabler for the low number of observed preemptions and migrations, we decided to maintain this core idea while relaxing any

assumption on job arrival times. This gave rise to SPRINT, an algorithm for scheduling sporadic task sets, which is presented next. Only the online phase of RUN is affected in SPRINT. The ofline part — i.e., the construction of the reduction tree — is identical in SPRINT and RUN.

The Algorithm

The algorithm we present is no different from RUN in its basic mechanism: servers are characterized by *deadlines* and at the firing of any of them the *execution budget* of servers is replenished to provide additional computational resources for new jobs execution; however, the way those quantities (i.e., deadlines and budgets) are computed changes to accommodate possible sporadic releases. Additionally, we need to enforce a new *server priority mechanism* for the scheduling of component servers to avoid the waste of processor time when the transient load of the system is low. However, differently from RUN, SPRINT currently only works on reduction trees up to 2 levels: although all the task sets we generated for our experimental evaluation (Section 4.3.3) satisfy this property, part of our future work is making SPRINT more general to schedule any given task set. We now show how these key modifications the differentiate SPRINT from RUN.

SERVER DEADLINES. Server deadlines play a central role in influencing the scheduling process of RUN, as they directly map to priorities under the EDF scheduling policy (Rule 1). It is therefore critical to carefully assign deadlines so that proper scheduling decisions are taken online. This principle still holds in SPRINT, with the further complication that the deadlines of servers at level 0 should account for the possible sporadic nature of the individual tasks wrapped into them.

Let $r_n(S_k^0)$ be the release of a new job of server S_k^0 at level $l = 0$. In RUN the job released by S_k^0 would be assigned a deadline equal to the minimum deadline of the tasks packed in S_k^0 [43]. However, in SPRINT, because of the sporadic nature of the tasks, some task τ_i may exist in S_k^0 which is not active at time $r_n(S_k^0)$ and therefore has no defined deadline. Nonetheless, we need to compute a deadline $d_k^0(r_n(S_k^0))$ for the job released by server S_k^0 at time $r_n(S_k^0)$. While computing this deadline, we want to preserve an important property of RUN: the firing of a deadline of any job released by any task τ_i always corresponds to the firing of a deadline in any server S_p^l such that $\tau_i \in S_p^l$. Furthermore, for the sake of simplicity, we do not want to update the deadline of S_k^0 at any instant other than the release of one of its job.

In order to fulfil those two requirements, for all the tasks $\tau_i \in S_k^0$ that are inactive at time $r_n(S_k^0)$, we consider their earliest possible deadline assuming that they release a job right after $r_n(S_k^0)$. That is, for each inactive task $\tau_i \in S_k^0$, we assume an artificial deadline $r_n(S_k^0) + D_i$. Thanks to the discussion above, the deadline of a server in SPRINT can be defined as follows:

Definition 9 (Server deadline in SPRINT)**.** *At any time t the deadline of a server S_k^0 on level $l = 0$ is given by*

$$d_k^0(t) \stackrel{\text{def}}{=} \min_{\tau_i \in S_k^0} \{ d_i(r_k^0(t)) | \text{ if } \tau_i \in \mathcal{A}(r_k^0(t)); \\ r_k^0(t) + D_i | \text{ if } \tau_i \notin \mathcal{A}(r_k^0(t)) \}$$

where $r_k^0(t)$ is the latest arrival time of a job of server S_k^0, i.e. $r_k^0(t) \stackrel{\text{def}}{=} \max_{r_n(S_k^0) \in \mathcal{R}(S_k^0)} \{ r_n(S_k^0) \mid r_n(S_k^0) \leq t \}$. At any time t the deadline of any other server S_k^l on a level $l > 0$ is defined as in RUN. That is,

$$d_k^l(t) \stackrel{\text{def}}{=} \min_{S_i^{l-1} \in S_k^l} \{ d_i^{l-1}(t) \}$$

It is worth noticing that, as expected, this definition preserves the property of RUN saying that the firing of a deadline of any job released by any task τ_i always corresponds to the firing of a deadline in any server S_k^l such that $\tau_i \in S_k^l$. Note however that the converse although true in RUN is not valid in SPRINT anymore, i.e. a deadline may be fired by a job of server S_k^l without any corresponding deadline in the task set $\mathcal{T}$.

REDUCTION AT LEVEL 0. As a more challenging modification to RUN, SPRINT must redefine the budget replenishment rules for servers. This is needed because the execution budget assigned to a server S_k^l at any level up to the root must reflect the execution time demand of the component tasks on the leaves placed below the subtree rooted in S_k^l. This time demand may now vary at any point in time as a consequence of sporadic releases and must be preserved upon the reduction process performed along the tree. While in RUN just one rule is needed to compute the budget of any server in the reduction tree, in SPRINT we need to distinguish different budget replenishment rules corresponding to the different levels at which a server is located in the reduction tree.

Let $R(S_k^0) \stackrel{\text{def}}{=} \{r_0(S_k^0), \ldots, r_n(S_k^0), \ldots\}$ be the sequence of time instants at which the budget of S_k^0 is replenished, where $r_0(S_k^0) \stackrel{\text{def}}{=} 0$ and $r_{n+1}(S_k^0) \stackrel{\text{def}}{=} d_k^0(r_n(S_k^0))$. Similarly to RUN, the budget allocated to S_k^0 at time $r_n(S_k^0)$ is a function of the utilisation of the active task in S_k^0 at time $r_n(S_k^0)$ and is given by

$$bdgt(S_k^0, r_n(S_k^0)) \stackrel{\text{def}}{=} \sum_{\tau_j \in S_k^0 \cap \mathcal{A}(r_n(S_k^0))} U(\tau_j) \times \Big(r_{n+1}(S_k^0) - r_n(S_k^0)\Big)$$

where $\mathcal{A}(r_n(S_k^0))$ is the set of active tasks at time $r_n(S_k^0)$.

However, in SPRINT, because of their sporadic nature, tasks packed in S_k^0 can also release jobs at any time t between two replenishment instants $r_n(S_k^0)$ and $r_{n+1}(S_k^0)$. At such a job release, the budget of S_k^0 should then be incremented to account for the new workload to be executed. More generally, if a set of tasks becomes active in a server S_k^0 at time $t_a \in \big(r_n(S_k^0), r_{n+1}(S_k^0)\big)$, the budget of S_k^0 should be incremented of an amount proportional to the cumulative utilisation of all released jobs. Formally,

$$bdgt(S_k^0, t_a) \stackrel{\text{def}}{=} bdgt(S_k^0, t_a^-) + \sum_{\tau_i \in S_k^0 \cap Rel(t_a)} U(\tau_i) \times (d_k^0(t_a) - t_a)$$

where $Rel(t_a)$ is the set of tasks releasing a job at time t_a and $bdgt(S_k^0, t_a^-)$ is the remaining execution budget of S_k^0 right before the arrival of those jobs at time t_a.

Considering the dual server $S_k^{0^*}$, we need to ensure that the budget assigned to $S_k^{0^*}$ is sufficiently small to allow the execution of the primal packed server S_k^0. Indeed, by definition of a dual server, the primal server S_k^0 executes only when $S_k^{0^*}$ is idle, and conversely S_k^0 is kept idle when $S_k^{0^*}$ executes. Therefore, as a minimal requirement we need to ensure that $bdgt(S_k^{0^*}, t) \leq (d_k^0(t) - t) - bdgt(S_k^0, t)$ at any time t. Since in RUN the budget assigned to a server S_k^0 may only vary at instants $r_n(S_k^l) \in R(S_k^l)$, it is sufficient to respect the equality $bdgt(S^{0^*}, t) \stackrel{\text{def}}{=} (d_k^0(t) - t) - bdgt(S_k^0, t)$ at any time t. In SPRINT instead the budget of S_k^0 may increase as soon as an inactive task $\tau_i \in S_k^0$ releases a new job (Rule 3). Therefore whenever the budget $bdgt(S_k^0, t)$ increases at any time t due to the release of a new job of S_k^0 or a job of an inactive task $\tau_i \in S_k^0$ the budget of $S_k^{0^*}$ needs to be computed according to the following equation:

$$bdgt(S_k^{0^*}, t) = (d_k^0(t) - t) - bdgt(S_k^0, t) - \sum_{\tau_i \in S_k^0, \tau_i \notin \mathcal{A}(t)} U(\tau_i) \times (d_k^0(t) - t)$$

where the last term accounts for the maximum workload by which the budget of S_k^0 could be inflated as a consequence of potential future job releases by the inactive tasks in S_k^0.

To summarize, in SPRINT the computation of the budgets of any servers S_k^0 and $S_k^{0^*}$ at level $l = 0$ must comply with the two following rules:

Rule 3 (Budget replenishment at level 0). *At any instant $r_n(S_k^0) \in R(S_k^0)$ servers S_k^0 and $S_k^{0^*}$ are assigned execution budgets*

$$\begin{cases} bdgt(S_k^0, r_n(S_k^0)) = \\ \qquad \sum_{\tau_j \in S_k^0 \cap \mathcal{A}(r_n(S_k^0))} U(\tau_j) \times \left(r_{n+1}(S_k^0) - r_n(S_k^0)\right) \\ bdgt(S_k^{0^*}, r_n(S_k^0)) = (d_k^0(t) - r_n(S_k^0)) - bdgt(S_k^0, r_n(S_k^0)) \\ \qquad - \sum_{\tau_i \in S_k^0, \tau_i \notin \mathcal{A}(r_n(S_k^0))} U(\tau_i) \times (d_k^0(t) - r_n(S_k^0)) \end{cases}$$

where $\mathcal{A}(r_n(S_k^0))$ is the set of active tasks at time $r_n(S_k^0)$.

Rule 4 (Budget update at level 0). *At any instant t (such that $r_n(S_k^0) < t < r_{n+1}(S_k^0)$) corresponding to the release of one or more jobs from one or more tasks τ_i in server S_k^0, the execution budgets of servers S_k^0 and $S_k^{0^*}$ are updated as follows:*

$$\begin{cases} bdgt(S_k^0, t) = bdgt(S_k^0, t^-) \\ \qquad + \sum_{\tau_j \in S_k^0 \cap Rel(t)} U(\tau_j) \times (d_k^0(t) - t) \\ bdgt(S_k^{0^*}, t) = (d_k^0(t) - t) - bdgt(S_k^0, t) \\ \qquad - \sum_{\tau_i \in S_k^0, \tau_i \notin \mathcal{A}(t)} U(\tau_i) \times (d_k^0(t) - t) \end{cases}$$

where $Rel(t)$ is the set of tasks releasing a job at time t and $bdgt(S_k^0, t^-)$ is the remaining execution budget of S_k^0 right before the arrival of those jobs at time t.

The following example shows how server budget can be computed in presence of both active and inactive tasks, resuming from our initial motivating example.

Example 5. *Let us consider level $l = 0$ in the reduction tree, i.e. the result of applying the* PACK *operation on the n individual tasks resulting into $|\mathcal{S}^0| \leq n$ servers. Figure 53 shows this*

situtaion, where τ_{1A} is a sporadic task and is initially inactive. Assuming that $d_{\tau_{1B}}(0) \leq D_{\tau_{1A}}$, the deadline at time 0 of server τ_1 in which τ_{1A} is packed is $d^0_{\tau_1}(0) = d_{\tau_{1B}}$, corresponding to the deadline of task τ_{1B}, and server τ_1 is assigned budget proportional to the acive tasks packed in it, i.e. $BDGT(\tau_1, 0) = \sum_{\tau_j \in \tau_1 \cap \mathcal{A}(0)} U(\tau_j) \times (d_{1B} - 0)$. Supposing now that τ_{1A} becomes active (and thus releases a job) at time t_1, with $0 < t_1 < d_{1B}$, the budget of server τ_1 should be raised to satisfy the execution demand of τ_{1A}. The amount such increment is given by $\Delta BDGT(\tau_1, t_1) = \sum_{\tau_j \in \tau_1 \cap Rel(t_1)} U(\tau_j) \times (d^0_{\tau_1}(t_1) - t_1) = U(\tau_{1A}) \times (d_{1B} - t_1)$. Later at time d_{1B} the budget is reset to $BDGT(\tau_1, d_{1B}) = (U(\tau_{1B}) + U(\tau_{1A})) \times (d_{1A} - d_{1B})$ since both τ_{1B} and τ_{1A} are active at d_{1B}. Thus overall the budget assigned to τ_1 for the execution of τ_{1A} is given by the sum of the budgets assigned in the 2 slots, i.e. $BDGT(\tau_{1A}, d_{\tau_{1A}}) = BDGT(\tau_{1A}, [t_1, d_{1B}]) + BDGT(\tau_{1A}, [d_{1B}, d_{1A}]) = BDGT(\tau_{1A}, [t_1, d_{1A}]) = U(\tau_{1A})(d_{1A} - t_1) = C_{1A}$.

We now prove formally that scheduling the packed servers at level 0 is equivalent to scheduling the task set $\mathcal{T}$.

Lemma 2. *Let S^0_k be a server at level $l = 0$ of the reduction tree and assume that S^0_k complies with Rules 3 and 4 for computing its budget. If S^0_k always exhausts its budget by its deadlines then all jobs released by the tasks in S^0_k respect their deadlines.*

Proof. In the following, we provide a proof sketch. According to Definition 9, all the deadlines of the jobs released by the tasks in S^0_k correspond to the deadlines of the jobs of S^0_k. Therefore, from Rules 3 and 4, the budget allocated to S^0_k between any instant corresponding to the release of a job by a task $\tau_i \in S^0_k$ and the deadline of any job released by the same or another task $\tau_j \in S^0_k$, is proportional to the utilisation of the tasks in S^0_k that are active between those two instants. That is, the budget allocated to the server (i.e., the supply) is larger than or equal to the sum of the worst-case execution times of the jobs of the tasks in S^0_k with both an arrival and a deadline in the interval (i.e., the demand). And because EDF is an optimal scheduling algorithm, all those jobs respect their deadlines. □

Properly assigning deadlines and budgets to servers is not sufficient to guarantee that the algorithm works. As mentioned at the beginning of this section, due to the fact that all tasks are not always active at any given time t, the prioritisation rules of the servers must also be adapted in SPRINT in order to avoid to waste computing time while there is still pending work in the system. Indeed, as shown in Example 4,

blindly using EDF to schedule the servers in the presence of sporadic tasks may lead to deadline misses. Because a sufficient condition for guaranteing the schedulability of the tasks in $\mathcal{T}$ is that all jobs of the servers at level $l = 0$ respect their deadlines (as proven by Lemma 2), then it is straightforward to conclude that there is no need to execute any server S_k^0 of level $l = 0$ for more than its assigned budget. To enforce this situation we rely on the idea of *dual schedule*, ensuring that S_k^0 does not execute when $S_k^{0^*}$ is running. Therefore, we just need to enforce the execution of $S_k^{0^*}$ as soon as a server S_k^0 exhausts its budget (even if $S_k^{0^*}$ already exhausted its own budget, i.e. $bdgt(S_k^{0^*}, t) = 0$): this can be achieved by assigning the highest priority to $S_k^{0^*}$. As a consequence, by virtue of Rule 2, $S_k^{0^*}$ will be favourably chosen to execute at level $l = 0^*$ (unless antoher server S_p^0 also completed its execution), thereby implying that S_k^0 will not execute (Rule 1). These observations are formalised by the following rule:

Rule 5 (Server priorities at level $l = 0^*$)**.** *If the budget of a server S_k^0 is exhausted at time t, i.e. $bdgt(S_k^0, t) = 0$, then the dual server $S_k^{0^*}$ is given the highest priority. Otherwise, if $bdgt(S_k^0, t) > 0$, the priority of $S_k^{0^*}$ is given by its deadline $d_k^0(t)$ as defined in Definition 9.*

REDUCTION AT LEVEL 1. We can extend the reasoning above to determine how the execution budgets should be replenished and how the scheduling decisions should be taken at levels $l = 1$ and $l = 1^*$ of the reduction tree. We first start with the observations in the following lemmas.

Lemma 3. *If S_i^{1*} executes at time t then all severs $S_k^0 \in S_i^1$ execute at time t.*

Proof. This lemma is a consequence of the dual operation applied in Rule 2. If S_i^{1*} executes at time t then, by Rule 2, S_i^1 does not execute at time t. Consequently, no component server $S_k^{0*} \in S_i^1$ executes either, which implies (applying again Rule 2) that all tasks $S_k^0 \in S_i^1$ execute at time t. □

Lemma 4. *If S_i^{1*} does not execute at time t then all severs $S_k^0 \in S_i^1$ but one execute at time t.*

Proof. If S_i^{1*} does not execute at time t then, by Rule 2, S_i^1 executes at time t. Consequently, by Rule 1, one component server $S_p^{0*} \in S_i^1$ executes at time t. Therefore, applying Rule 2 again, we get that all tasks $S_k^0 \in \{S_i^1\} \setminus S_p^0$ execute at time t. □

A direct consequence of Lemmas 3 and 4 is that there is no need for executing S_i^{1*} when at least one of the servers $S_k^0 \in S_i^1$ exhausted its budget. Therefore, S_i^{1*} is assigned the lowest priority to prevent its execution as long as a server $S_k^0 \in S_i^1$ has budget $bdgt(S_k^0, t) = 0$. Hence, the following rule applies at level $l = 1^*$:

Rule 6 (Server priorities at level $l = 1^*$)**.** *If the budget of a server S_k^0 is exhausted at time t, i.e. $bdgt(S_k^0, t) = 0$, then the server $S_k^{1^*}$ such that $S_k^0 \in S_k^{1^*}$ is given the lowest priority. Otherwise, if $bdgt(S_k^0, t) > 0$ for all $S_k^0 \in S_k^{1^*}$, the priority of $S_k^{1^*}$ is given by its deadline $d_k^1(t)$ as defined in Definition 9.*

At levels 1 and 1^*, the budget replenishment policy applied at any instant $r_n(S_k^1) \in R(S_k^1)$ is not different from RUN. Hence, the algorithm still respects the following rule:

Rule 7 (Budget replenishment at level 1)**.** *At any instant $r_n(S_k^1) \in R(S_k^1)$ servers S_k^1 and $S_k^{1^*}$ are assigned execution budgets*

$$\begin{cases} bdgt(S_k^1, r_n(S_k^1)) = U(S_k^1) \times \left(r_{n+1}(S_k^1) - r_n(S_k^1)\right) \\ \\ bdgt(S_k^{1^*}, r_n(S_k^1)) = (d_k^1(t) - r_n(S_k^1)) - bdgt(S_k^1, r_n(S_k^1)) \end{cases}$$

Additionally, one more rule is needed to define the behaviour of the algorithm when a task releases a job at time t such that $r_n(S_k^1) < t < r_{n+1}(S_k^1)$. This rule is given below and uses the operator $[x]_y^z$ defined as $\min\{z, \max\{y, x\}\}$.

Rule 8 (Budget update at level 1)**.** *At any instant t such that $r_n(S_k^1) < t < r_{n+1}(S_k^1)$, corresponding to the update of one or more jobs from one or more server $S_p^0 \in S_k^1$, if $bdgt(S_p^0, t^-) = 0$ and calling $t_0 \geq r_n(S_k^1)$ the instant at which $bdgt(S_p^0, t)$ became equal to 0, then the execution budgets of servers S_k^1 and $S_k^{1^*}$ are updated as follows:*

$$\begin{cases} bdgt(S_k^1, t) = \left[U(S_k^1) \times (d_k^0(t) - t)\right]_{bdgt(S_k^1, t_0) - (t - t_0)}^{bdgt(S_k^1, t_0)} \\ bdgt(S_k^{0^*}, t) = (d_k^1(t) - t) - bdgt(S_k^1, t) \end{cases}$$

where $bdgt(S_k^0, t^-)$ and $bdgt(S_k^1, t^-)$ are the remaining execution budgets of S_k^0 and S_k^1, respectively, right before the budget updates occurrs at time t.

REDUCTION AT LEVEL 2. By assumption, there is a maximum of two reduction levels in SPRINT. Consequently, level 2 can only be the root of the reduction tree. Since by definition the root has always an utilisation equal to 100%, it is always executing and no budget nor priority have to be computed. Therefore, servers on level 1^* will always be scheduled for execution according to their relative priorities, computed using Rule 6.

4.3.3 *Evaluation*

We now compare SPRINT to state-of-the-art multicore scheduling algorithms: in particular, we are interested in counting the number of preemptions and migrations incurred by the task sets to be scheduled. These two metrics are a trustworthy indicator of the interference caused by OS back-end activities to the executing applications, and eventually give a hint on how composability is the eased or impaired by the choice of a specific algorithm.

It has been demonstrated in [199] that RUN can be actually implemented with reasonable performance when compared to other existing partitioned and global algorithms. We therefore assume that this result can be easily extended to SPRINT, which is in the end based on RUN, to only focus on simulations for now; we are perfectly aware however that the implementation of a scheduling algorithm on a real target entails considering run time overheads and other implementation issues, whose solutions may question some of the results obtained by simulations.

All our experiments compare SPRINT with Partitioned-EDF (P-EDF), Global-EDF (G-EDF) and U-EDF by scheduling randomly generated sporadic task sets. Individual tasks are characterised by their period, randomly chosen in the a range of $[5, 100]$ time units; sporadic release is simulated by randomly picking a task's arrival delay in a range of values depending on the specific scenario. Every point in the graphs we present in this section is the result of the scheduling of 1000 task sets with each algorithm. During the offline reduction process of SPRINT, no task set required more than 2 levels in its reduction tree.

In the first batch of experiments we wanted to study SPRINT performance as a function of the varying system utilisation. We simulated a system with 8 processors and we randomly generated task utilizations between 0.01 and 0.99 until the targeted system utilisation was reached, increasing it progressively from 55% to 100%. Sporadic inter-arrival times of tasks were generated by adding a value in the range $[0, 100]$, randomly picked from a uniform integer distribution, to their period. Figures 55(a) and 55(b) show the good results obtained for SPRINT in terms of preemptions and migrations per job, respectively; in particular, we notice that the number of migrations incurred by SPRINT is always smaller than the number experienced under both G-EDF and U-EDF. The number of preemptions approaches the well-known results for P-EDF, at least up to a utilisation of 85%, i.e. $U = 7$. After that point however, the number of scheduled task sets for P-EDF and G-EDF drops substantially, as evident from

Figure 55(c)[5], until the extreme of 100% utilisation where not even a valid partitioning is found for P-EDF. The schedulability ratio does not change instead for U-EDF and SPRINT as a consequence of their optimality results.

We tried then to repeat a similar experiment, this time keeping system utilisation fixed to 90%, i.e. $U = 7.2$, and rather making the number of tasks vary. In our expectations this would challenge even more the relative performance of the algorithms, since growing the number of concurrent tasks in the system increases the potential operations to be performed by the scheduling algorithm. Figures 56(a) and 56(b) show that the number of preemptions for SPRINT is similar to that of P-EDF and G-EDF, while the number of migrations is even smaller (in fact null) than the migrations registered by G-EDF and U-EDF. However, with a small number of tasks, whose individual utilisation must be therefore large, P-EDF and G-EDF fail to schedule some task sets as a consequence of the impossibility of finding a good partitioning and of taking advantage of task migration, respectively (Figure 56(c)). U-EDF is instead comparable to SPRINT in terms of achieved schedulability, still paying some penalty due to a higher number of preemptions and migrations.

As a final experiment we observed the behaviour of SPRINT and U-EDF when the number of processors in the system increases, while keeping the system fully utilised. As expected, both the number of preemptions (Figure 57(a)) and migrations (Figure 57(b)) increase for U-EDF with the size of the system, whereas for SPRINT it remains constant on average, and always below the value of 3. This is in line with the results obtained for RUN and by virtue of the observation that no task set in our experiments requires more than 2 reduction levels.

At the same time we were interested in understanding how the behaviour of both algorithms is affected by changes in the minimum inter-arrival times of sporadic releases. To this purpose we defined three representative equivalence classes for sporadic release times. represened again as random delays to be added to task periods: (i) the first is $[0, 0]$, which corresponds to having only periodic tasks and is therefore suitable to roughly compare SPRINT and U-EDF on a strictly periodic system; (ii) the second is the range $[0, max\_period]$, so that there is at least one job release every two task periods; finally, (iii) the third range $[0, 10 \times max\_period]$ allows larger delays than task periods. What we notice is that case (ii) is the most expensive both in terms of preemptions and migrations, for both algorithms. This is explained by the fact that in that scenario jobs are released often enough to cause a significant amount of

5 In this case we only count the number of preemptions and migrations incurred by the schedulable task sets.

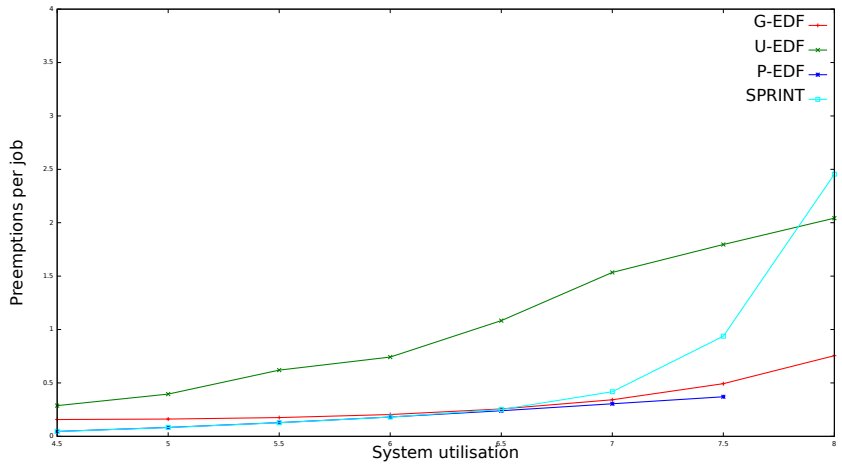
scheduling; additionally such releases are likely to happen out-of- phase with respect to each other, therefore generating even more scheduling points. On the contrary in setting (i) jobs are more likely to be released in phase, whereas in setting (iii) job releases are far less frequent, thus diluting the number of dispatched scheduling events.

4.4 SUMMARY

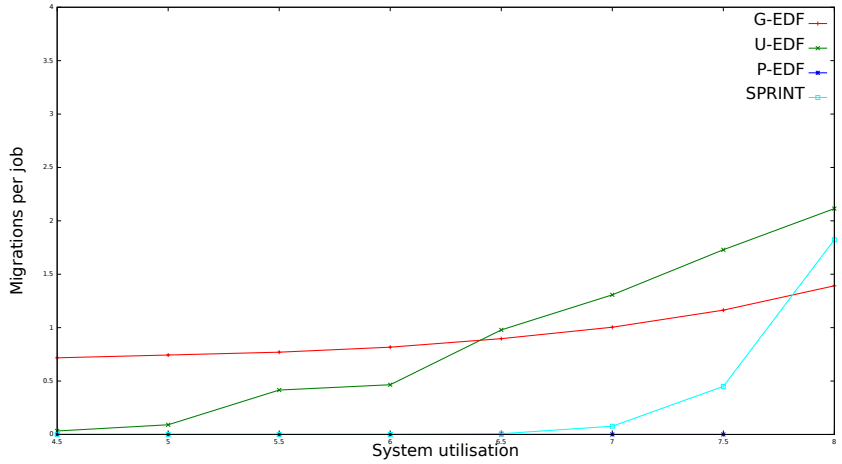
In this section we attacked the problem of providing time composability in a multiprocessor environment. We first identified the sources of interference within the HW layer, whose behaviour is likely to be even more disrupting to time composability than in a single core scenario as a consequence of the increased number of shared HW resources and interactions thereof. We then showed how different models of computation have different implications on the applicability of multicore processing in a real-world industrial setting, defining either a partitioned architecture or a more ambitious setup where, in the quest for work conservation, tasks are allowed to migrate across processors. As an example of the former setting we showed how the PROARTIS avionics case study has been deployed to a partitioned multicore setup, where the degree of time composability achieved in TiCOS enables probabilistic timing analysis, similarly to the single-core setting. . Initial results finally proved the limited increase in the runtime overhead that SPRINT incurs over RUN. We then proceeded by studying how the design decisions taken in the areas of intervention presented in Chapter 3 are affected by the deployment in a real multiprocessor environment, and identified the problem of optimal multicore scheduling as the most critical to time composability.

Looking at the existing algorithms for multiprocessor scheduling, we observed that both partitioned and global approaches present some drawbacks which limit their usefulness to real-world applications: the former have to cope with the bin-packing problem, which does not account for inter-task dependencies, and introduce over provisioning in the system; the latter leave freedom of migration to tasks, whose overhead may be unacceptable in practice. In this scenario, the approach taken by RUN mediates between both partitioned and global scheduling, grouping tasks into servers and partitioning among them while allowing migration at run time; RUN has proven to be effective in practice by limiting the number of incurred preemptions and migrations to a minimum. Building on this analysis, our interest has focused on providing a solution to counter one limitation of RUN, i.e. the support to sporadic task sets, giving

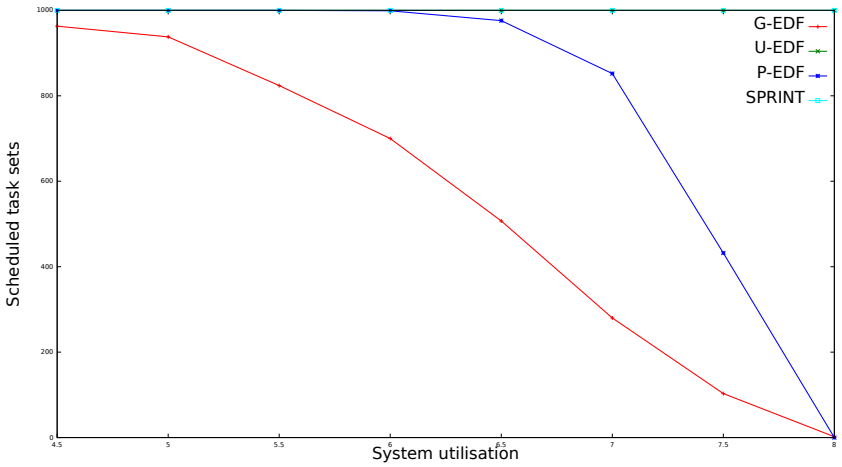
rise to a new algorithm which we called SPRINT. Although not general enough yet to support task sets requiring more than 2 reduction levels in RUN reduction tree, SPRINT has proven to outperform other state-of-the-art algorithm for multiprocessor scheduling in terms of both tasks incurred preemptions and migrations.



(a)

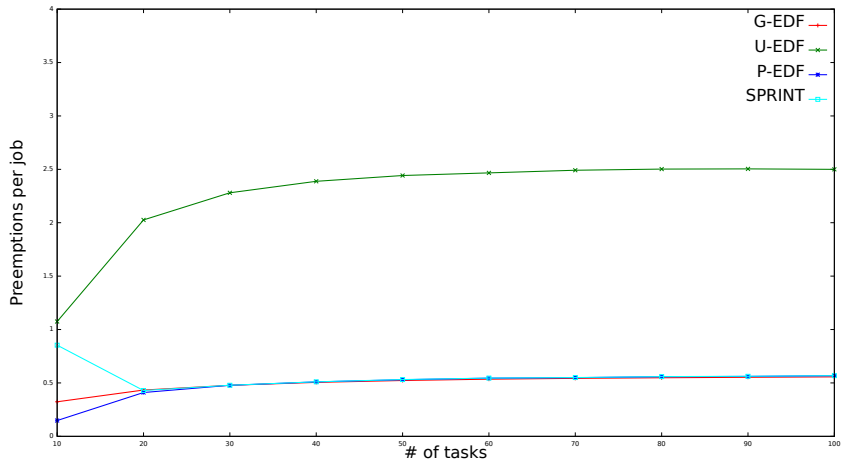


(b)

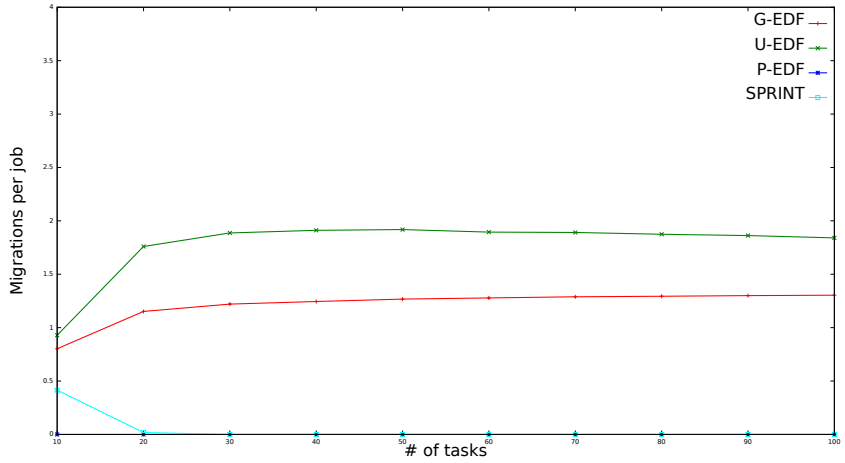


(c)

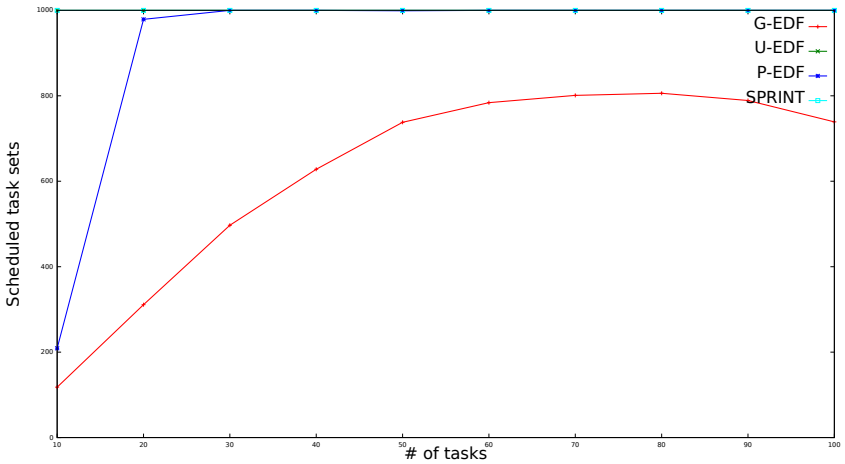
Figure 55.: Comparative results for SPRINT with respect to G-EDF, P-EDF and U-EDF in terms of preemptions (a) and migrations (b) per job, and number of schedulable task sets (c) with increasing system utilisation.



(a)

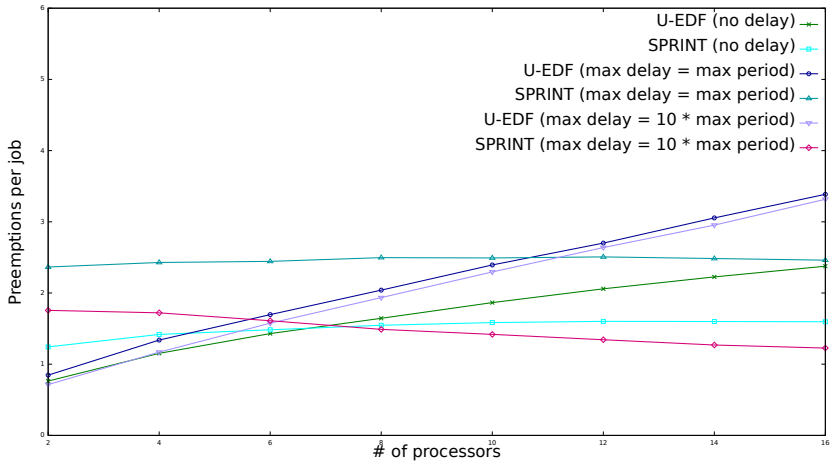


(b)

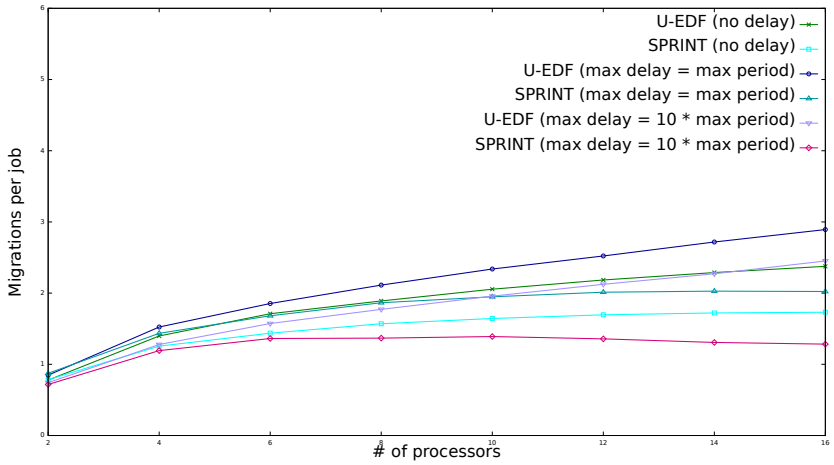


(c)

Figure 56.: Comparative results for SPRINT with respect to G-EDF, P-EDF and U-EDF in terms of preemptions (a) and migrations (b) per job, with increasing number of tasks and limited to the number of scheduled tasks (c).



(a)



(b)

Figure 57.: Comparative results for SPRINT with respect to U-EDF in terms of preemptions (a) and migrations (b) per job, with different inter-arrival times and increasing number of processors.

5

CONCLUSIONS AND FUTURE WORK

5.1 TIME COMPOSABILITY AND THE ROLE OF THE OS

The increasing performance expectations placed on real-time systems are pushing even the more conservative industrial stakeholders to consider the use of advanced hardware features into architecture designs. Moreover, the need of cutting production costs causes COTS components to become an attractive alternative to ad-hoc expensive solutions. This is not an immediate option for the high-integrity domain, where large investments are dedicated to qualification activities, which cover significant amounts of project budgets.

In the industrial scenario of interest to this work, incrementality and compositionality are the cornerstone principles guiding all the phases of the system delivery process. When those two properties are injected into a system blueprint, the resulting product will be more robust to both unstable requirements or execution conditions and to the technology shift pushing towards the adoption of more performing solutions.

Unfortunately, the advocacy of compositionality and its supporting methodologies are scarcely supported by existing technology. When it comes to assessing the timing behaviour of a system in fact, existing timing analysis techniques hit the composability wall, which prevents the analysis of each individual component in isolation without considering the contingent execution conditions and dependencies on the specific deployment environment. Providing composability is therefore crucial to the industrial process at the basis of HIRTS production, which eventually results into improved performance and rationalised costs.

In this dissertation we started by observing that existing research on time composability has attacked the problem either at the hardware level, to study unpredictable and disruptive hardware effects or at the application level, by looking at those computation paradigms that make an application more easily analysable. Unfortunately, at either level the problem of enumerating all possible HW/SW interactions during execution is known to be hard and it is far from being solved, even in a uniprocessor environment, unless resorting to a simplified or overprovisioned architecture is accepted. We contend that the role played by the Real-Time Operating System in this

problem deserves deeper investigation: as a consequence of its privileged mediating position in the architecture, the OS may potentially mitigate the reciprocal perturbation effects between the HW platform and the applications deployed on top of it. We have given a characterisation of time composability in terms of *zero disturbance* and *steady behaviour*, which we have sought as distinctive properties emerging from an educated OS design. If the OS is time composable with applications, its contribution to their cumulative execution time is a linear function of the invoked services. Additionally, its internals may be completely replaced requiring minimal re-qualification effort of client applications. In parallel to this benefits we demonstrated how a time composable operating system is also one mandatory enabler of the emerging probabilistic timing analysis techniques, in which the hypotheses of independence and identical distribution of described events are satisfied only if interference effects are removed from the intra- and inter-layer interactions among architectural components.

5.2 SUMMARY OF CONTRIBUTIONS

The main goal of our work shifts the attention to the role played by the OS in the challenge of providing time composability in the delivery process of complex high-integrity real-time systems. Along our quest for time composability, however, we wanted to sustain our claims with real-world evidence on the good results that can be achieved when OS design is performed with composability in mind, thus favouring analysability over raw average performance. This approach has led to the following contributions:

1. **Time composability** has been characterised in Section 3.1 in terms of disturbance and steady behaviour, in a way that can be adapted to a broad variety of existing real-time systems, with no assumption on the specific system architecture. Later in Section 3.2 we spotted those areas of an OS which are most critical to achieve composable timing behaviour of the entire system and we instantiated those principles in the context of an RTOS design.

2. With a solid notion of time composability in our hands, we focused our investigation in Section 3.3 on the IMA architecture for the **avionics domain**: we showed how the design of an ARINC 653 RTOS kernel can be modified to inject time composability into it (Section 3.4). The results of our refactored kernel, TiCOS, have been validated both in isolation and in the context of a real-world Airbus case study application (Section 3.4.3), by comparing against a non-time-composable

design of the same OS. We then moved outside the boundaries of the simplified task model defined by the ARINC specification by studying time composability in the **space domain**. We presented in Section 3.6 the implementation of an activation-triggered limited-preemptive scheduler on top of a Ravenscar Ada compliant RTOS kernel, ORK+, which is to-date the only existing implementation of this kind in a real-world OS. Its beneficial effects on time composability have been demonstrated in Section 3.6.4. Additionally, we proposed a solution to the problem of dealing with resource sharing in a limite-preemptive scheduling framework.

3. As a final contribution, the problem of providing time composability in **multicores** has been attacked. We showed how the criticalities identified in the design of a single core OS are possibly amplified in a multiprocessor setup (Section 4.2.2). We proved how the time composable behaviour of TiCOS enables probabilistic analysis of applications running on a partitioned multiprocessor platform (Section 4.2.1). In order to support the industrial adoption of more ambitious multicore computing styles, we looked at multicore scheduling in Section 4.3.1 as a crucial problem to facilitate system design while ensuring high system performance. We motivated our interest in the RUN algorithm by illustrating its virtues and we proposed a way to extended it to handle sporadic tasks which defines a new algorithm called SPRINT (Section 4.3.2). In Section 4.3.3 we show the good performance of SPRINT when compared to state-of-the-art multicore scheduling techniques.

5.3 FUTURE WORK

The contributions above try to cover a number of aspects in the multifaceted problem of injecting time composability in both unicore and multicore systems. An extensive evaluation of the problem would probably require an effort which goes well beyond the scope of a PhD thesis. Therefore we focused on the study of the OS layer where we observed the most promising margin for improvement over existing design and analysis techniques. In our study however we spotted a number of open problems which are certainly at least as stimulating as those we already faced, and which we are currently investigating or we want to cope with in the near future.

In the single core setting it would be nice to provide a low-level OS API to manipulate those HW shared resources whose behaviour is immutably defined at design time.

For example, the possibility of selectively enabling and disabling the use of hardware acceleration features (like caches) would permit to consciously take benefit from their use in moments of execution when performance matters; on the contrary, when timeliness is a major concern one may want to disable them to guarantee predictable timing behaviour of an application. This clearly emerged while refactoring the ARINC 653 input/output API interface.

In the implementation of our limited-preemptive scheduler in ORK+, we had the chance to observe how an activation-triggered scheduling policy, which lazily defers any scheduling operation until the laxity of a job drops to zero, is particularly sensitive to kernel primitives overhead. This means that even modest overruns of the expected execution time within the kernel may eventually result in job missing their deadlines. We are therefore interested in precisely characterising the timing behaviour of time composable ORK+ primitives, so that a precise response-time analysis accounting for kernel overheads can be devised.

For the continuation of the multiprocessor investigation we certainly need to study whether and how SPRINT can be made more general to be applied to reduction trees with more than two levels. If this were possible, we could start reasoning about the possible optimality of SPRINT, deriving from the results obtained for RUN. Additionally, we need to produce further data to confirm (or deny) the promising simulation results obtained in the comparison with other existing multicore scheduling algorithms. In this particular regard, we plan to implement both U-EDF and SPRINT on top of LITMUS^{RT} to compare their relative performance on a real platform.

A

ARINC APEX

We provided an implementation of a subset of the ARINC APEX services, as required by the Flight Control Data Concentrator (FCDC) industrial case study we were involved in (see Section 3.4.4). In our effort of redesigning part of the ARINC API layer according to the principle of time composability we obviously preserved the original semantics and the signature of the standard services. The intent is that legacy application could be ported on our RTOS with no change. Not all ARINC services needed attention: those that belong to the system initialization phase (e.g., process creation during partition WARM_START mode) have no effect on time composability. Similarly, a number of other ARINC services just operate as simple wrappers to RTOS kernel primitives that we redesigned to be time composable. What is most interesting to this work is the implementation of the I/O communication services and other synchronisation primitives. Table 14 summarises the set of ARINC services involved in the FCDC case study and evaluates their potential effects on timing composability: the next sections discuss each of them in more detail.

Partition Management

The ARINC 653 model is centred on the concept of partitions as a means to guarantee time and memory isolation between separate functions. Partitioning allows separate software functions (possibly characterised by different criticality levels) to execute in the same computational node without affecting each other. The effects of timing or memory faults should remain within a partition and should not propagate to the system.

GET_PARTITION_STATUS returns the status of the partition that is currently executing. This service naturally exhibits a constant timing behaviour, as the current partition in TiCOS is represented by a PARTITION_ID that allows an $O(1)$ access to a global array of data structures storing the information required by the implementation of this service. Although data dependent hardware features will be polluted by the execution of this service, the incurred interference is quite limited and should be easily bounded (e.g., number of unique memory accesses for caches). As this service is not loop-intensive and does not exhibit any temporal locality (only spatial), we expect it not to take great

Table 14.: ARINC services required in the FCDC case study.

Id	Service name	Composability issues	
		Disturbance	*Unsteady behaviour*
PARTITION MANAGEMENT			
S 1	GET_PARTITION_STATUS	Limited, boundable	Almost constant-time
S 2	GET_PARTITION_START_CONDITION	Limited, boundable	Almost constant-time
S 3	SET_PARTITION_MODE	*Not invoked in nominal behaviour*	
PROCESS MANAGEMENT			
S 4	PERIODIC_WAIT	Limited	Implementation-dependent
S 5	GET_TIME	Boundable	Slight (input-dependent)
S 6	CREATE_PROCESS	*Not invoked in nominal behaviour*	
S 7	STOP	Reduced interference	Implementation-dependent
S 8	START	Potentially large	Context-dependent
LOGBOOK MANAGEMENT			
...	*Discarded*	(not used in FCDC)	
SAMPLING PORTS MANAGEMENT			
S 9	CREATE_SAMPLING_PORT	*Not invoked in nominal behaviour*	
S10	WRITE_SAMPLING_MESSAGE	Variable	Input-dependent
S11	READ_SAMPLING_MESSAGE	Variable	Input-dependent
S12	GET_SAMPLING_PORT_ID	*Not invoked in nominal behaviour*	
QUEUING PORTS MANAGEMENT			
S13	CREATE_QUEUING_PORT	*Not invoked in nominal behaviour*	
S14	WRITE_QUEUING_MESSAGE	Variable+blocking	Input-dependent
S15	READ_QUEUING_MESSAGE	Variable+blocking	Input-dependent
S16	GET_QUEUING_PORT_ID	*Not invoked in nominal behaviour*	

benefit of history-dependent acceleration features. In this case, freezing the hardware state could be a reasonable option as it would not incur overly penalising effects on performance.

GET_PARTITION_START_CONDITION returns the reason why the partition is started. In our case study this service has been stubbed to return always NORMAL_START as start condition. The same considerations as above can be applied to this service.

SET_PARTITION_MODE switches the partition execution mode. Partition execution modes and the respective transitions play a relevant role during system initialisation, when a partition enters the NORMAL_MODE, and error handling. When entering the NORMAL_MODE, the partition processes can be scheduled for execution. The timing behaviour of this service is extremely variable on both the input parameter (i.e., the new partition MODE) and the number of processes in the partition – $O(\#processes)$. The latter dependence is due to the fact this service may need to iterate over all the

partition processes either to change their state to READY or to set the respective release points. In fact, this service can be disregarded when considering composability issues as its invocation is confined to the system initialisation or error handling phases. Both cases are typically not accounted for when analysing the timing behaviour.

Process Management

Each partition supports the concurrent execution of several processes, characterised by either a periodic or sporadic (aperiodic in ARINC terminology) behaviour. Processes exist only within their partition, which is responsible for their scheduling. Besides a proper scheduling and dispatching mechanism, the underlying OS must therefore provide a set of scheduling services to control the process execution.

PERIODIC_WAIT enforces the cyclic execution of periodic processes. The service is invoked by the process itself, which therefore self-suspends until its next periodic activation. From the functional standpoint, this service consists in accessing and manipulating the scheduling data structures to set the next process activation[1]. The behaviour of this service may suffer from large variability when timing-unaware OS data structures are adopted to manage the system processes. In an ideal case, when a constant-time OS data structures – with $O(1)$ process queue management[2] – is provided, the timing behaviour is pretty stable (i.e., exhibiting a single execution path). We already discussed on constant-time scheduling data-structures and functions in section 3.4.2. With respect to the amount of disturbance possibly generated by this OS primitive, we should note that this is the last call of a process before self-suspension. Thanks to the run-to-completion policy, the execution of this OS call cannot have any disturbing effect on other processes as they have either terminated or not started their execution yet.

GET_TIME returns the value of the system-wide clock, starting from the system start up, which is globally valid for all partitions. Transforming the values retrieved from the PPC TIME_BASE into μseconds requires a 64-bit division which is a relatively fast but not constant-time operation as it depends on the division operands. It should be possible, however, to provide a safe upperbound to its worst-case execution time.

CREATE_PROCESS creates an ARINC process within a partition. A process is fully characterised by a set of qualifying attributes given as parameters on process creation. An ARINC-compliant system is required to support no more than 128 processes per

1 Next activation of a periodic process is just the time of the previous activation plus the process period (with no need to retrieve the current time).

2 This includes constant-time insertion, selection and process state update.

partition. Since process creation is allowed only during the initialisation phase, before entering the partition NORMAL mode, we do not consider this service as part of the nominal system behaviour.

STOP stops the execution and inhibits the schedulability of any other process, except itself[3]. The stopped process cannot be scheduled until another process invokes the dual START service. In order to stop a process the OS scheduling data structures need to be accessed and manipulated (e.g., update of process queues). As observed for the PERIODIC_WAIT service, the timing variability incurred by the stop service depends on the actual implementation of the scheduling data-structures and functions. When it comes to disturbance, the STOP service has positive effects on preserving the HW state as it actually implies the removal of a potential source of interference.

START triggers the execution of another process within the same partition of the calling process. This service causes the started process to be initialised to its default attributes value. According to [40] the effects of the invocation of this service vary on the nature of the started process. Staring an aperiodic process, in particular, is considered as a dispatching point and may cause the preemption of the calling process. Such behaviour is at the same time extremely variable (depending on the call context) and disturbing, as it can clearly introduce additional inter-task interferences on the HW state. By enforcing run-to-completion semantics we do not allow the START service to trigger a dispatching point, but we defer it until after the process job has returned.

Inter-Partition Communication

With respect to the subset of ARINC services provided by current implementation, the main timing-composability issues are to be ascribed to the I/O communication between partitions. The basic message-oriented communication mechanisms provided by the ARINC specification are *channels* [40], which are defined as logical links between one source and one or more destinations. Partitions can then gain access to communication channels via points called PORTS[4]. Channels, ports and their associations are statically predetermined and defined via configuration tables, and cannot be modified at run-time. Two kinds of communication modes are defined for inter-partition message exchange: sampling mode and queuing mode. Each system port must be either a sampling or a queuing port. While in sampling mode, successive messages typically carry identical but updated data, in the queuing mode each new instance of a message is unique and cannot be overwritten.

3 The STOP_SELF service is used in this case.
4 A channel thus connects a SOURCE port to a DESTINATION port.

On the one hand, the execution time of I/O-related activities, as those involved in reading from and writing into a communication channel, are inherently dependent on the amount of data in hand. On the other hand, the communication mechanisms themselves break the assumption of isolation between partitions: read and write become potentially blocking operations, depending on the semantics of the communication port.

CREATE_SAMPLING_PORT and CREATE_QUEUING_PORT create a sampling or queuing port, respectively. Ports are not actually created but only mapped to an already reserved memory area defined at configuration time. The main effect of the port creation service is the association of a unique identifier to a port name. Port creation, of course, also comes with a set of parameters that correctly initialise the identified port. From our standpoint, however, these services are quite irrelevant as they are invoked at initialisation time and are not included in the canonical concept of nominal execution.

GET_SAMPLING_PORT_ID and GET_QUEUING_PORT_ID are typically invoked from within the respective port creation services to associate a unique identifier to the newly created port. This PORT_ID can be used to identify a port without using the port name. The use of a proper port id allows to organise the system ports into constant-time access data structures. In our implementation we assume that these services are exclusively invoked at port initialisation and the returned port id is visible within the owning partition.

READ_SAMPLING_MESSAGE and WRITE_SAMPLING_MESSAGE provide the APEX interface for I/O operations on sampling ports. As observed before, in sampling mode each read request to a destination port simply reads the last value that can be found in the temporary storage area of the port (which in turn is overwritten upon every new message reception); each write request to a source port, instead, overwrites the previous one. A READ request then simply copies the value in the port to a specified address. A WRITE request, instead, simply writes a message of a specific length form a defined address to the port. The invocation of any of these operations cannot block the execution of a job awaiting data, which are instead simply read/written from/to a memory location with no guarantee on its actual content [5].

READ_QUEUING_MESSAGE and WRITE_QUEUING_MESSAGE have exactly the same role as the previous ones except for the fact that they apply to queuing mode communication. In queuing mode each write message request is temporarily stored in an ordered message queue associated to the port: in contrast with sampling ports however, if the

5 A read request when no message is still available would just result in an invalid message.

queue cannot accept the message (i.e., it is full or has insufficient space) the calling process gets blocked, waiting for the required space (eventually setting a timeout); each read request will eventually pick up the first message from the receive queue. Although the level of implementation required by the FCDC experiments excludes the occurrence of blocking and the use of timeout timers, a potentially blocking service is extremely challenging from the time composability point of view.

To summarise, the time composability issues raised by I/O services, either through sampling or queuing ports are mainly due to the variability induced by the amount of data to be read or written. Whereas ports are characterised by a maximum size, forcing the exchange of the maximum amount of data would obtain a constant-time behaviour at the cost of an unacceptable performance loss. Moreover, the potential blocking incurred by queuing port could further complicate the disturbing effects of inter-partition communication. Also the natural countermeasure of isolating the effects of the service execution on the hardware state cannot be seamlessly applied in this context. Inhibiting the caches for example is likely to kill performance since the read and write operations are inherently loop intensive and greatly benefit from both temporal and spatial locality.

To counter this unstable and disturbing behaviour we separate the variable part of the read/write services (i.e., the loop intensive data transfer operations). Unfortunately, we cannot simply use the service as a trigger for a separate sporadic process which would do the dirty job, as memory partitioning does not allow it. Although we cannot simply remove the I/O-related variability, we can always decide to accommodate such variability so that it is likely to incur less disturbing effects on the execution of the application code. Based on the fact that communication channels are entirely defined at configuration time, we exploit the available information on the inter-partition communication patterns to perform some sort of preventive I/O in between partition switch, as depicted in Figure 58. Assuming that the addresses – local to a partition – involved in the I/O are statically known we postpone all actual port writes to the slack time at the end of a partition scheduling slot. Similarly, we preload the required data into partition destination ports (and the respective local variables) in a specular slack time at the beginning of a scheduling slot. The information flow is guaranteed to be preserved as we are dealing with inter-partition communication: (i) the state of all destination (input) ports is already determined at the beginning of a partition slot; (ii) the state of all source (output) ports is not relevant until the partition slot terminates and another partitions gets scheduled for execution.

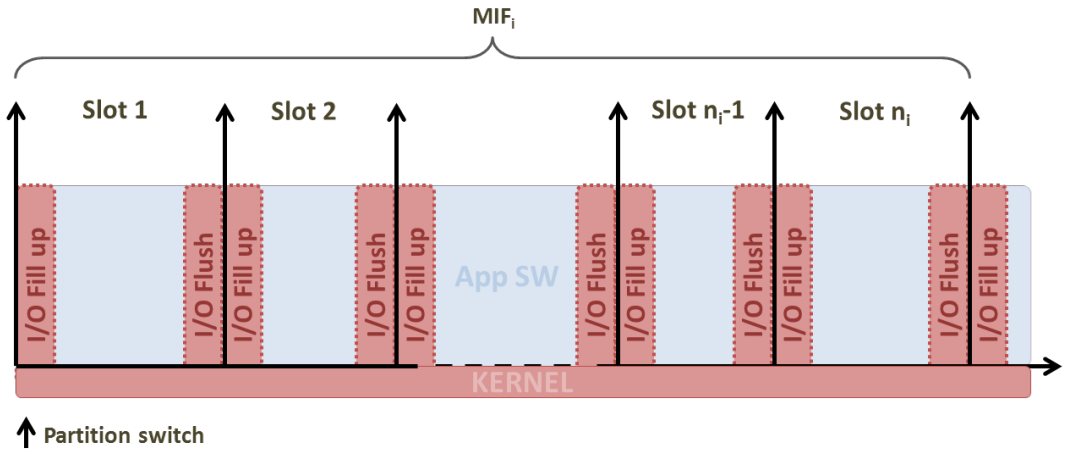


Figure 58.: Inter-partition I/O management.

This way, we should not worry about the disturbing effects on the hardware state as no optimistic assumption should ever be made on partition switching; moreover, the input-dependent variability can be analysed within some sort of end-to-end analysis.

We further present some additional ARINC services whose behaviour poses serious challenges to the achievement of time composability, although they are not directly involved in the Airbus FCDC case study.

Intra-Partition Communication

All communications involving processes within a partition are realised through BLACKBOARD and BUFFER services. Similarly to the inter-partition case, variable-size data messages are exchanged in a sampling or queueing fashion respectively. The main concern with intra-partition I/O services, comes from potentially blocking calls as the timing variability stemming from different workloads can easily be accountable for at application level. Potentially blocking calls are removed and treated with the *process split pattern* as explained below.

A process that makes potentially blocking calls has more than one source of activation events. We know that this feature complicates timing analysis, as the functional behaviour of that process can no longer be studied as a single sequential program with a single entry point. So we want to be break it into as many parts as its sources of activation events. Conversely, from the ARINC task model perspective discussed in Section 3.3, the segment of the original (periodic) process that is released after

the blocking event must be regarded as an indistinguishable part of the original periodic process. Hence, the CPU time needed for executing the whole process must be accounted for in the scheduling slot regardless of whether the blocking condition is satisfied or not.

In our solution we want all processes to have a *single* source of activation event, whether time- or event-triggered. To this end we systematically break all processes that make potentially blocking calls into concatenation of processes. For the sake of discussion, let us consider a single blocking call while the argument easily extends to multiple such calls. Any such process can be divided in two parts: the first part (predecessor) performs all the activities of the original process up to the potentially blocking call; the other part (successor) includes all the activities from past the blocking call until the end. The release event for the predecessor is the same as the original process. The release event for the successor occurs when the predecessor has completed *and* the blocking condition has been asserted to open.

The adoption of the process split pattern in the ARINC architecture introduces a new run-time entity, which would correspond to a true sporadic process if we added to its release event the condition that a specified minimum inter-arrival time had elapsed since its last run. Unfortunately, the introduction of sporadic processes resulting from the process split pattern in our RTOS complicates scheduling. The policy we implemented to guarantee good responsiveness places sporadic processes in a separate FIFO queue (similar to the one we used for asynchronous activations). As shown in Figure 14 (d) sporadic processes may be dispatched as early as possible so long as the following condition holds: they should be able to complete within their absolute deadline without causing a deadline overrun to ready periodic processes. If periodic processes were conceptually ordered by increasing slack time, a sporadic process would be run in the earliest eligible scheduling slot where its WCET was no greater than the slack of the periodic process at the head of the periodic ready queue.

Low-Level Synchronisation Services

The adoption of a run-to-completion semantics and the avoidance of of any form of process blocking downsizes somehow the role of ARINC process synchronisation services, such as EVENTS and SEMAPHORES. EVENTS still have a *raison d'être* as a trigger for application-level synchronisation: for the sake of time composability they are implemented according to the process split pattern, with constant time signal and broadcast operations. Conversely SEMAPHORES become pretty useless in a system were processes are guaranteed not to be preempted since there is no need to worry

about mutual exclusion. SEMAPHORES are implemented in our OS only as a means to support legacy applications.

Table 15 summarises the solutions we adopted for the sake of a time-composable OS layer.

Table 15.: TiCOS implementation status.

Id	Primitive/Service name	Implemented solution
KERNEL		
K 1	Time management	Switch from tick-based (decrementer) time management to a timer-based one.
		Enforcement of the *run-to-completion* semantics.
K 2	Scheduling primitives	Constant-time dispatching.
		Constant-time process state update.
		Limited deferral of asynchronous events.
ARINC-653 PARTITION MANAGEMENT		
S 1	GET_PARTITION_STATUS	*Standard implementation.*
S 2	GET_PARTITION_START_CONDITION	*Standard implementation.*
S 3	SET_PARTITION_MODE	[*Start-up phase*]*Standard implementation.*
ARINC-653 PROCESS MANAGEMENT		
S 4	PERIODIC_WAIT	*Standard implementation.*
S 5	GET_TIME	*Standard implementation.*
S 6	CREATE_PROCESS	[*Start-up phase*]*Standard implementation.*
S 7	STOP	*Standard implementation.*
S 8	START	*Standard implementation.*
ARINC-653 LOGBOOK MANAGEMENT		
...	*Discarded*	—
ARINC-653 SAMPLING PORTS MANAGEMENT		
S 9	CREATE_SAMPLING_PORT	[*Start-up phase*]*Standard implementation.*
S10	WRITE_SAMPLING_MESSAGE	Zero-disturbance posted writes.
S11	READ_SAMPLING_MESSAGE	Zero-disturbance read prefetch.
S12	GET_SAMPLING_PORT_ID	[*Start-up phase*]*Standard implementation.*
ARINC-653 QUEUING PORTS MANAGEMENT – *Not in FCDC*		
S13	CREATE_QUEUING_PORT	[*Start-up phase*]*Standard implementation.*
S14	WRITE_QUEUING_MESSAGE	Zero-disturbance posted writes.
S15	READ_QUEUING_MESSAGE	Zero-disturbance read prefetch.
S16	GET_QUEUING_PORT_ID	[*Start-up phase*]*Standard implementation.*
ARINC-653 BLACKBOARDS MANAGEMENT – *Not in FCDC*		
S17	CREATE_BLACKBOARD	[*Start-up phase*]*Standard implementation.*
S18	DISPLAY_BLACKBOARD	Process split pattern.
S19	READ_BLACKBOARD	Process split pattern.
S20	GET_BLACKBOARD_ID	[*Start-up phase*]*Standard implementation.*
ARINC-653 BUFFERS MANAGEMENT – *Not in FCDC*		
S21	CREATE_BUFFER	[*Start-up phase*]*Standard implementation.*
S22	SEND_BUFFER	Process split pattern.
S23	RECEIVE_BUFFER	Process split pattern.
S24	GET_BUFFER_ID	[*Start-up phase*]*Standard implementation.*

B

SYSTEM MODEL AND NOTATION FOR LIMITED PREEMPTION

We address the problem of scheduling a set $\mathcal{T}$ of n periodic or sporadic tasks on a uniprocessor system. Each task $\tau_i \in \mathcal{T}$ is characterized by a worst-case execution time C_i, a period (respectively minimum inter-arrival time) T_i and a constrained deadline $D_i \leq T_i$. A task is also assigned a fixed priority π_i^b that subsumes a total ordering between tasks such that $i \leq j$ if $\pi_i^b > \pi_j^b$: hence τ_1 is the highest priority task. Accordingly we define $\mathcal{T}_i^+ \doteq \{\tau_j | j < i\}$ resp. $\mathcal{T}_i^- \doteq \{\tau_j | j > i\}$ as the sets of higher resp. lower priority tasks of τ_i.

In deferred preemption approaches, a task τ_j can defer a preemption request and continue to execute for a limited amount of time. As observed in [202, 41], the effect of this temporary inhibition of preemption is that all tasks in $\mathcal{T}_j^+$ may be blocked for the duration of q_j^{max}, the longest NPR in τ_j. For preemptively feasible task sets, an upper bound B_i to the blocking suffered by task τ_i is given by:

$$B_i^{\text{NPR}} = \max_{i<j\leq n} \left\{ q_j^{max} \right\} \tag{5}$$

The maximum length of a NPR in τ_j that preserves feasibility with respect to a fully preemptive case is termed Q_j and its computation is based on the concept of *blocking tolerance* β_i of all tasks $\tau_i \in \mathcal{T}_j^+$. β_i in fact is an upper bound to the maximum blocking suffered from τ_i that preserves the task set feasibility. The computation of β_i, in turn, depends on the cumulative execution request of all tasks in $\mathcal{T}_i^+$ on the longest level-i busy period in the general case. However, as long as we consider preemptively feasible task sets, and knowing that limited preemptive approaches dominate fully preemptive scheduling [41], we can restrict the analysis to a smaller set of points in time, as proven by Bini and Buttazzo in [203]. In case of preemptively feasible task sets in fact, a necessary and sufficient schedulability condition under deferred preemption is given in [202, 41] as:

$$B_i^{\text{NPR}} \leq \beta_i \doteq \max_{a \in A, a \leq D_i} \left\{ a - \sum_{j \leq i} \left\lceil \frac{a}{T_j} \right\rceil C_j \right\} \tag{6}$$

with

$$A = \{kT_j, k \in \mathbb{N}, 1 \leq j < n\} \tag{7}$$

Aside from limiting preemption, task blocking also stems from serialized accesses to shared resources. In a typical system a set of shared resources are accessed from within critical sections: a task τ_i may thus access a shared resource R through a critical section $cs^R_{i,k}$, where the latter identifies the k^{th} critical section in τ_i accessing resource R. Critical sections are assumed to be properly nested so that they can never overlap and $cs^R_{i^*}$ is the longest outermost critical section in τ_i accessing R. The maximum blocking suffered from a task depends on the resource access protocol in use[167].

In the Immediate Ceiling Priority Protocol (ICPP), a derivative of Baker's Stack Resource Policy[204], each shared resource R is assigned a ceiling priority, $ceil(R)$, which is set to at least the priority value of the highest-priority task that uses that resource. Upon entering a critical section a task immediately gets its *active priority* raised to the ceiling priority. Amongst other properties (i.e., deadlocks are prevented and blocking can occur only once, just before task activation) ICPP minimizes the blocking duration for a task τ_i, B_i^{CS}, to the longest outermost critical section executed by a lower-priority task τ_j using a resource with a ceiling priority greater than or equal to the priority of τ_i, whereby:

$$B_i^{\text{CS}} \leq \max_{\substack{i<j\leq n, \\ ceil(R)\geq\pi_i^b}} cs^R_{j^*} \tag{8}$$

where $cs^R_{j^*} = 0$ if τ_j does not access shared resource R. For the sake of notation we denote $\mathcal{T}_i^B \subseteq \mathcal{T}_i^-$ as the set of lower priority tasks that may block τ_i.

Table 16 summarizes the notation that we adopted in Section 3.6.3.

Symbol	Meaning
C_i	Worst-case execution time
T_i	Period (or minimum inter-arrival)
D_i	relative deadline
π_i^b	Base priority
π_i^{active}	Active priority (ceiling)
q_i^{max}	Longest non preemptive region
β_i	blocking tolerance
$cs^R_{i,k}$	k^{th} critical section in τ_i accessing R
$cs^R_{i^*}$	Longest outermost critical section in τ_i accessing R
$\mathcal{T}_i^+$	Set of higher priority tasks
$\mathcal{T}_i^-$	Set of lower priority tasks
$\mathcal{T}_i^B$	Subset of lower priority tasks that may block τ_i

Table 16.: Characterization of a task.

BIBLIOGRAPHY

[1] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, Apr. 2008.

[2] R. T. C. for Aeronautics (RTCA), "Do-178b: Software considerations in airborne systems and equipment certification," 1982.

[3] G. Edelin, "Embedded systems at thales: the artemis challenges for an industrial group," in *Presentation at the ARTIST Summer School in Europe 2009*, 2009.

[4] T. Scharnhorst, "Autosar - an industry-wide initiative to manage the complexity of emerging e/e architectures." Presentation at the 75th Geneva International Motor Show, 2005.

[5] E. W. Dijkstra, "On the role of scientific thought," 1974.

[6] P. Puschner, R. Kirner, and R. G. Pettit, "Towards composable timing for real-time programs," *Software Technologies for Future Dependable Distributed Systems*, pp. 1–5, Mar. 2009.

[7] M. Panunzio and T. Vardanega, "On component-based development and high-integrity real-time systems," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 79–84, IEEE, Aug. 2009.

[8] D. Schmidt, "Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[9] M. Panunzio and T. Vardanega, "Pitfalls and misconceptions in component-oriented approaches for real-time embedded systems: lessons learned and solutions," in *Proceedings of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2010.

[10] M. Panunzio, *Definition, Realization and Evaluation of a Software Reference Architecture for Use in Space Applications*. PhD thesis, University of Bologna, 2011.

[11] I. Crnkovic and M. R. V. Chaudron, *Software Engineering; Principles and Practice*, ch. Component-based Software Engineering. Wiley, 2008.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, January 1973.

[13] M. Joseph and P. K. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[14] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, pp. 159–176, Sept. 1989.

[15] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proceedings of the 23rd Real-Time Systems Symposium (RTSS)*, pp. 279–288, 2002.

[16] F. Cazorla, E. Quinones, T. Vardanega, L. Cucu-Grosjean, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analysable real-time systems," *ACM Transactions on Embedded Computing Systems. Special issue on Probabilistic Computing*, To appear, 2012.

[17] `http://www.absint.com/ait`, Feb. 2012. AbsInt aiT Tool Homepage.

[18] Y.-T. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS)*, pp. 298–307, IEEE, 1995.

[19] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pp. 12–21, IEEE, 1999.

[20] R. Kirner, A. Kadlec, and P. Puschner, "Worst-case execution time analysis for processors showing timing anomalies," Research Report 01/2009, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009.

[21] A. Coombes, "How to measure and optimize reliable embedded software," in *ACM SIGAda Annual International Conference*, 2011.

[22] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*, vol. 2, UBooks Verlag, Dec. 2005.

[23] S. Edgar and A. Burns, "Statistical analysis of wcet for scheduling," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 215–224, IEEE, 2001.

[24] G. Bernat, A. Colin, and S. Petters, "pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems," tech. rep., 2003.

[25] A. Prantl, M. Schordan, and J. Knoop, "Tubound - a conceptually new tool for worst-case execution time analysis," in *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2008.

[26] Rapita Systems Ltd., "Rapitime," 2012. `http://www.rapitasystems.com/rapitime`.

[27] F.J. Cazorla et al., "Proartis: Probabilistically analysable real-time systems," Tech. Rep. 7869 (http://hal.inria.fr/hal-00663329), INRIA, 2012.

[28] J. Hansen, S. Hissam, and G. Moreno, "Statistical-based WCET estimation and validation," in *9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[29] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *ECRTS*, 2012.

[30] `http://www.proartis-project.eu/`, Feb. 2013. PROARTIS Project Homepage.

[31] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis," in *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 333–339, IEEE, May 2008.

[32] E. Mezzetti and T. Varadanega, "On the industrial fitness of wcet analysis," in *11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2011.

[33] E. Mezzetti, N. Holsti, A. Colin, G. Bernat, and T. Vardanega, "Attacking the sources of unpredictability in the instruction cache behavior," in *Proc. of the 16th Int. Conference on Real-Time and Network Systems (RTNS)*, 2008.

[34] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *44th Asilomar Conference on Signals, Systems, and Computers*, pp. 2111–2115, November 2010.

[35] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: Tightening the crpd bound for set-associative caches," in *Proc. of the Conference on Languages, compilers, and tools for embedded systems*, LCTES '10, 2010.

[36] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, 2009.

[37] J. Schneider, "Why you can't analyze RTOSs without considering applications and vice versa," in *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2002.

[38] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, no. 2, pp. 115–128, 2000.

[39] J. Gustafsson, "Usability aspects of wcet analysis," in *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 346–352, IEEE, May 2008.

[40] I. Aeronautical Radio, *ARINC Specification 653-1: Avionics Applicaiton Software Standard Interface*, 2003.

[41] G. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 3 –15, 2013.

[42] G. Yao, G. Buttazzo, and M. Bertogna, "Comparative evaluation of limited preemptive methods," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pp. 1–8, 2010.

[43] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 104–115, 2011.

[44] A. Burns and D. Griffin, "Predictability as an emergent behaviour," in *Proceedings of the 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, pp. 27–29, 2011.

[45] M. Delvai, W. Huber, P. Puschner, and A. Steininger, "Processor support for temporal predictability - the spear design example," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pp. 169–176, IEEE, 2003.

[46] S. Basumallick and K. Nilsen, "Cache issues in real-time systems," in *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[47] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pp. 204–212, 1996.

[48] I. Puaut, "Cache modelling vs static cache locking for schedulability analysis in multitasking real-time systems," in *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2002.

[49] A. M. Campoy, A. P. Ivars, and J. V. B. Mataix, "Dynamic use of locking caches in multitask, preemptive real-time systems," in *15th FAC Triennal World Congress*, 2002.

[50] D. Kirk, "Smart (strategic memory allocation for real-time) cache design," in *Real Time Systems Symposium, 1989., Proceedings.*, pp. 229–237, 1989.

[51] E. Quinones, E. Berger, G. Bernat, and F. Cazorla, "Using randomized caches in probabilistic real-time systems," in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pp. 129–138, 2009.

[52] L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, "Multi-level unified caches for probabilistically time analysable real-time systems," in *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*, IEEE, Nov. 2013.

[53] R. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pp. 168–179, 2013.

[54] K. Patil, K. Seth, and F. Mueller, "Compositional static instruction cache simulation," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)*, (New York, NY, USA), pp. 136–145, ACM, 2004.

[55] E. Mezzetti and T. Vardanega, "Towards a cache-aware development of high integrity real-time systems," in *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 329–338, IEEE, Aug. 2010.

[56] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Systems*, vol. 18, no. 2, pp. 249–274, 2000.

[57] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for wcet analysis," *Real-Time Systems*, vol. 34, pp. 195–227, June 2006.

[58] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "Wcet coverage for pipelines," tech. rep., 2006.

[59] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 169–178, IEEE, July 2007.

[60] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, "AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures," in *Convergence International Congress & Exposition On Transportation Electronics*, pp. 325–332, 2004.

[61] H. Fennel and S. e. a. Bunzel, "Achievements and exploitation of the autosar development partnership," technical report, AUTOSAR Partnership, 2006.

[62] A. Colin and I. Puaut, "Worst-case execution time analysis of the rtems real-time operating system," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 191–198, IEEE, 2001.

[63] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of wcet analysis of real-time operating systems," in *Proceedings of the International Conference on Embedded Software and Systems (ICESS)*, pp. 65–72, IEEE, 2009.

[64] G. Khyo, P. Puschner, and M. Delvai, "An operating system for a time-predictable computing node," in *Proceedings of the 6th IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pp. 150–161, Springer-Verlag, 2008.

[65] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 339–348, IEEE, Nov. 2011.

[66] M. Lv, N. Guan, Y. Zhang, R. Chen, Q. Deng, G. Yu, and W. Yi, "Wcet analysis of the mc/os-ii real-time kernel," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 2, pp. 270–276, Aug 2009.

[67] J. Schneider, *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2002.

[68] L. K. Chong, C. Ballabriga, V.-T. Pham, S. Chattopadhyay, and A. Roychoudhury, "Integrated timing analysis of application and operating systems code," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pp. 128–139, Dec 2013.

[69] J. Yang, Y. Chen, H. Wang, and B. Wang, "A linux kernel with fixed interrupt latency for embedded real-time system," in *Proceedings of the 2nd International Conference on Embedded Software and Systems (ICESS)*, 2005.

[70] I. Molnar, "Goals, Design and Implementation of the new ultra-scalable O(1) scheduler," Jan. 2002. Available on-line at `http://casper.berkeley.edu/svn/trunk/roach/sw/linux/Documentation/scheduler/sched-design.txt`, visited on April 2012.

[71] J. Aas, "Understanding the linux 2.6.8.1 cpu scheduler." Available online at `http://joshaas.net/linux/linux_cpu_scheduler.pdf`, Feb. 2005.

[72] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static timing analysis of real-time operating system code," in *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA)*, October 2004.

[73] L. George, N. Rivierre, and M. Spuri, "Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling," Research Report RR-2966, INRIA, 1996. Projet REFLECS.

[74] A. Burns, "Preemptive priority-based scheduling: An appropriate engineering approach," in *Advances in Real-Time Systems*, pp. 225–248, 1994.

[75] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS '07, pp. 269–279, 2007.

[76] R. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, pp. 63–119, 2009.

[77] G. Yao, G. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with fixed preemption points," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pp. 71–80, 2010.

[78] G. Yao, G. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Real-Time Systems*, vol. 47, pp. 198–223, 2011.

[79] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption points placement for sporadic task sets," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pp. 251–260, 2010.

[80] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pp. 217–227, July.

[81] J. Marinho and S. Petters, "Job phasing aware preemption deferral," in *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pp. 128–135, 2011.

[82] J. Marinho, S. Petters, and M. Bertogna, "Extending fixed task-priority schedulability by interference limitation," in *RTNS*, pp. 191–200, 2012.

[83] M. Bertogna and S. Baruah, "Limited preemption edf scheduling of sporadic task systems," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 4, pp. 579–591, 2010.

[84] J. Marinho, V. Nelis, S. Petters, and I. Puaut, "Preemption delay analysis for floating non-preemptive region scheduling," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 497–502, 2012.

[85] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pp. 328–335, 1999.

[86] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pp. 25–34, 2000.

[87] J. Gustafsson, "Worst case execution time analysis of object-oriented programs," in *Proceedings of the 7th International Workshop onObject-Oriented Real-Time Dependable Systems (WORDS)*, pp. 71–76, IEEE, 2002.

[88] E.-S. Hu, G. Bernat, and A. Wellings, "Addressing dynamic dispatching issues in wcet analysis for object-oriented hard real-time systems," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pp. 109–116, IEEE, 2002.

[89] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad, "A modular worst-case execution time analysis tool for java processors," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 47–57, IEEE, Apr. 2008.

[90] E. Kligerman and A. D. Stoyenko, "Real-Time Euclid: a language for reliable real-time systems," *Transactions on Software Engineering*, vol. 12, pp. 941–949, Sept. 1986.

[91] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[92] P. Puschner and A. Schedl, "Computing maximum task execution times - A graph-based appraoch," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.

[93] R. Kirner, "The programming language wcetC," Research Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[94] B. Carré and J. Garnsworthy, "Spark - an annotated ada subset for safety-critical programming," in *Proceedings of TRI-ADA*, pp. 392–402, ACM, 1990.

[95] M. I. S. R. A. (MISRA), "Guidelines for the use of the c language in critical systems," 2004.

[96] E. Mezzetti, *Cache-aware Development of High Integrity Real-time Systems*. PhD thesis, University of Bologna, 2012.

[97] A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat, "Evaluating tight execution time bounds of programs by annotations," *Real-Time System Newsletter*, vol. 5, pp. 81–86, May 1989.

[98] C. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems*, vol. 5, pp. 31–62, Mar. 1993.

[99] P. Puschner and A. Burns, "Writing temporally predictable code," in *Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pp. 85 –91, 2002.

[100] P. Puschner, "The single-path approach towards wcet-analysable software," in *Proceedings of the International Conference on Industrial Technology (ICIT)*, vol. 2, pp. 699–704, IEEE, 2003.

[101] P. Puschner, "Algorithms for dependable hard real-time systems," in *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pp. 26–31, IEEE, 2003.

[102] A. Colin and G. Bernat, "Scope-tree: a program representation for symbolic worst-case execution time analysis," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 50–59, IEEE, 2002.

[103] L. David and I. Puaut, "Static determination of probabilistic execution times," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 223–230, IEEE, 2004.

[104] F. Wolf, R. Ernst, and W. Ye, "Path clustering in software timing analysis," *Transactions on Very Large Scale Integration Systems*, vol. 9, no. 6, pp. 773–782, 2001.

[105] Y.-t. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, 1997.

[106] S. Bygde, A. Ermedahl, and B. Lisper, "An efficient algorithm for parametric wcet calculation," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 13–21, IEEE, Aug. 2009.

[107] J. Deverge and I. Puaut, "Safe measurement-based wcet estimation," in *Proceedings of the 5th International Workshop on Worst Case Execution Time Analysis (WCET)*, pp. 13–16, 2005.

[108] J. Fredriksson, T. Nolte, A. Ermedahl, and M. Nolin, "Clustering worst-case execution times for software components," in *Proceedings of the 7th International Workshop on Worst Case Execution Time Analysis (WCET)*, pp. 19–25, July 2007.

[109] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, pp. 275–298, Nov. 1998.

[110] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pp. 134 –143, Dec. 1998.

[111] P. Atanassov, S. Haberl, and P. Puschner, "Heuristic worst-case execution time analysis," in *Proceedings of the 10th European Workshop on Dependable Computing (EWDC)*, pp. 109–114, Austrian Computer Society (OCG), May 1999.

[112] G. Bernat and N. Holsti, "Compiler support for wcet analysis: a wish list," in *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET)*, pp. 65–69, 2003.

[113] R. Kirner and P. P. Puschner, "Classification of code annotations and discussion of compiler-support for worst-case execution time analysis," in *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2005.

[114] R. Kirner and P. Puschner, "Transformation of path information for wcet analysis during compilation," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 29–36, IEEE, 2001.

[115] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo, "A tool for automatic flow analysis of c-programs for wcet calculation," in *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pp. 106–112, IEEE, 2003.

[116] C. Curtsinger and E. D. Berger, "Stabilizer: statistically sound performance evaluation," in *ASPLOS*, pp. 219–228, 2013.

[117] P. Puschner and G. Bernat, "Wcet analysis of reusable portable code," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 45–52, IEEE, 2001.

[118] L. Thiele, E. Wandeler, and N. Stoimenov, "Real-time interfaces for composing real-time systems," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT)*, pp. 34–43, ACM, 2006.

[119] J. Fredriksson, T. Nolte, M. Nolin, and H. Schmidt, "Contract-based reusableworst-case execution time estimate," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 39–46, IEEE, 2007.

[120] M. Santos and B. Lisper, "Evaluation of an additive wcet model for software components," in *Proceedings of the 10th Brazilian Workshop on Real-time and Embedded Systems (WTR)*, 2008.

[121] J. Yi, D. Lilja, and D. Hawkins, "Improving computer architecture simulation methodology by adding statistical rigor," *Computers, IEEE Transactions on*, vol. 54, no. 11, pp. 1360–1373, 2005.

[122] M. Santos, B. Lisper, G. Lima, and V. Lima, "Sequential composition of execution time distributions by convolution," in *Proceedings of the 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, pp. 30–37, Nov. 2011.

[123] A. Marref, "Compositional timing analysis," in *Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, pp. 144–151, 2010.

[124] T. Leveque, E. Borde, A. Marref, and J. Carlson, "Hierarchical composition of parametric wcet in a component based approach," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 261–268, IEEE, Mar. 2011.

[125] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proceedings of the 25th Real-Time Systems Symposium (RTSS)*, IEEE, 2004.

[126] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky, "Compositional schedulability analysis of hierarchical real-time systems," in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 274–281, IEEE, May 2007.

[127] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional scheduling framework for digital avionics systems," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 371–380, IEEE, Aug. 2009.

[128] L. De Alfaro, T. Henzinger, and M. Stoelinga, "Timed interfaces," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 108–122, Springer, 2002.

[129] R. Ben Salah, M. Bozga, and O. Maler, "Compositional timing analysis," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 39–48, IEEE, 2009.

[130] L. Santinelli and L. Cucu-Grosjean, "Toward probabilistic real-time calculus," in *Proceedings of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2010.

[131] H. Kopetz and R. Obermaisser, "Temporal composability," *Computing and Control Engineering*, vol. 13, pp. 156–162, Aug. 2002.

[132] T. Henzinger, C. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in *ACM SIGPLAN Notices*, vol. 40, pp. 21–30, ACM, 2005.

[133] H. Kopetz, "Why time-triggered architectures will succeed in large hard real-time systems," in *Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of*, pp. 2–9, 1995.

[134] H. Kopetz, "The time-triggered model of computation," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pp. 168–177, 1998.

[135] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "A definition and classification of timing anomalies," 2006.

[136] IBM, *PowerPC 740, PowerPC 750 - RISC Microprocessor User's Manual, GK21-0263-00*, 1999. "http://www.chips.ibm.com/.

[137] S. Edwards and E. Lee, "The case for the precision timed (pret) machine," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 264–265, 2007.

[138] http://www.t-crest.org/, Feb. 2013. T-CREST Project Homepage.

[139] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," in *Computer Safety, Reliability, and Security* (F. Ortmeier and P. Daniel, eds.), vol. 7613 of *Lecture Notes in Computer Science*, pp. 382–391, Springer Berlin Heidelberg, 2012.

[140] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, NOCS '12, (Washington, DC, USA), pp. 152–160, IEEE Computer Society, 2012.

[141] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst, "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems* (P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, eds.), vol. 18 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 11–21, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

[142] PROARTIS Consortium, "D1.2 - platform design guidelines for single core - version 1.0," tech. rep., 2011.

[143] J. Bradley, *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[144] F. Mueller, "Compiler support for software-based cache partitioning," in *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[145] G. Yao, G. C. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Real-Time Systems*, vol. 47, no. 3, pp. 198–223, 2011.

[146] R. T. C. for Aeronautics, *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. Nov. 2005.

[147] J. Delange and L. Lec, "POK, an ARINC653-compliant operating system released under the BSD license," *13th Real-Time Linux Workshop*, 2011.

[148] C. E. Leiserson, H. Prokop, and K. H. Randall, "Using de Bruijn Sequences to Index a 1 in a Computer Word," 1998.

[149] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *the International Workshop on Worst-case Execution-time Analysis* (B. Lisper, ed.), (Brussels, Belgium), pp. 137–147, OCG, July 2010.

[150] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder, "Principles of timing anomalies in superscalar processors," *Proceedings of the Fifth International Conference on Quality Software*, pp. 295–306, 2005.

[151] L. Kosmidis, J. Abella, E. Quinones, and F. J. Cazorla, "A cache design for probabilistically analysable real-time systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 513–518, 2013.

[152] Esterel, "SCADE." `www.esterel-technologies.com/products/scade-suite/`.

[153] W. Feller, *An introduction to Probability Theory and Its Applications*. John Willer and Sons, 1996.

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

[155] M. Garrido and J. Diebolt, "The ET test, a goodness-of-fit test for the distribution tail," in *Methodology, Practice and Inference, second international conference on mathematical methods in reliability*, pp. 427–430, 2000.

[156] S. of Automotive Engineers (SAE), "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment," *ARP4761*, 2001.

[157] Universidad Politécnica de Madrid, "GNAT/ORK+ for LEON cross-compilation system." `http://www.dit.upm.es/~ork`.

[158] ISO SC22/WG9, "Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1," 2005.

[159] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems," *TR YCS-2003-348, University of York*, 2003.

[160] T. Vardanega, J. Zamorano, and J. A. de la Puente, "On the dynamic semantics and the timing behavior of ravenscar kernels," *Real-Time Systems*, vol. 29, no. 1, pp. 59–89, 2005.

[161] Freescale, "PowerPC 750 Microprocessor," 2012. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_750_Microprocessor.

[162] J. Zamorano, J. F. Ruiz, and J. A. de la Puente, "Implementing Ada.Real_Time.Clock and Absolute Delays in Real-Time Kernels," in *Proceedings of the 6th International Conference on Reliable Software Technologies*, Ada Europe, pp. 317–327, 2001.

[163] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 824–834, 1997.

[164] I. Molnar, "Goals, design and implementation of the new ultra-scalable O(1) scheduler," 2002. Linux Kernel, Source tree documentation.

[165] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pp. 137–144, 2005.

[166] R. I. Davis and M. Bertogna, "Optimal fixed priority scheduling with deferred pre-emption," in *Proceedings 33rd IEEE Real-Time Systems Symposium (RTSS'12)*, 2012.

[167] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[168] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, pp. 99–122, November 2007.

[169] PROARTIS Consortium, "D1.3 - platform design guidelines for multicore - version 1.0," tech. rep., 2013.

[170] PROARTIS Consortium, "Multicore phase requirements - version 1.0," tech. rep., 2012.

[171] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis – definition and challenges," in *CRTS*, December 2013.

[172] PROARTIS Consortium, "D4.4 - multicore case study results - version 1.0," tech. rep., 2013.

[173] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, pp. 35:1–35:44, Oct. 2011.

[174] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[175] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. Comput.*, vol. 44, pp. 1429–1442, Dec. 1995.

[176] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. pp. 127–140, 1978.

[177] C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation (extended abstract)," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, (New York, NY, USA), pp. 140–149, ACM, 1997.

[178] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*, RTSS '01, (Washington, DC, USA), pp. 183–, IEEE Computer Society, 2001.

[179] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, (New York, NY, USA), pp. 345–354, ACM, 1993.

[180] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[181] S. K. Baruah, J. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, (Washington, DC, USA), pp. 280–288, IEEE Computer Society, 1995.

[182] J. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pp. 76–85, 2001.

[183] D. Zhu, D. Mosse, and R. Melhem, "Multiple-resource periodic scheduling problem: how much fairness is necessary?," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pp. 142–151, 2003.

[184] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pp. 3–13, 2010.

[185] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Syst.*, vol. 47, pp. 389–429, Sept. 2011.

[186] H. Cho, B. Ravindran, and E. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pp. 101–110, 2006.

[187] S. K. Lee, "On-line multiprocessor scheduling algorithms for real-time tasks," in *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pp. 607–611 vol.2, 1994.

[188] G. Nelissen, *Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems*. PhD thesis, UniversitÃ© Libre de Bruxelles, 2013.

[189] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, vol. 1, pp. 15–24, 2011.

[190] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 13–23, 2012.

[191] P. Regnier, *Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor*. PhD thesis, Universidade Federal da Bahia, 2012.

[192] P. Regnier, G. Lima, E. Massa, G. Levin, and S. A. Brandt, "Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach," *Real-Time Systems*, vol. 49, no. 4, pp. 436–474, 2013.

[193] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pp. 322–334, 2006.

[194] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling ofÂ sporadicÂ tasks with high utilisation bound," *Real-Time Systems*, vol. 47, no. 4, pp. 319–355, 2011.

[195] G. Levin, C. Sadowski, I. Pye, and S. Brandt, "Sns: A simple model for understanding optimal hard real-time multiprocessor scheduling," Tech. Rep. ucsc-soe-11-09, UCSC, 2009.

[196] D. Zhu, X. Qi, D. Mossé, and R. Melhem, "An optimal boundary fair scheduling algorithm for multiprocessor real-time systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1411–1425, 2011.

[197] M. Moir and S. Ramamurthy, "Pfair scheduling of fixed and migrating periodic tasks on multiple resources," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, (Phoenix, AZ, USA), pp. 294–303, IEEE Computer Society, December 1999.

[198] A. Burns and A. Wellings, "A schedulability compatible multiprocessor resource sharing protocol – mrsp," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pp. 282–291, 2013.

[199] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting run into practice: implementation and evaluation," in *submission to the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[200] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pp. 111–126, 2006.

[201] `http://www.litmus-rt.org/`, Feb. 2013. LITMUS^{RT}, The Linux Testbed for Multiprocessor Scheduling in Real Time Systems.

[202] G. Yao, G. Buttazzo, and M. Bertogna, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pp. 351–360, 2009.

[203] E. Bini and G. C. Buttazzo, "Schedulability analysis of periodic fixed priority systems," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1462–1473, 2004.

[204] T. P. Baker, "Stack-based Scheduling for Realtime Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.