Worst-Case and Best-Case Timing Analysis for Real-Time Embedded Systems with Limited Parallelism

(Submitted for the degree of Doctor of Philosophy)

Konstantinos Bletsas

(*Κωνσταντίνος Μπλέτσας*)

Department of Computer Science,

University of York

May 2007

Abstract

Manufacturing advances throughout the last decade have led to a steady increase in the capacity of Field Programmable Gate Array (FPGAs). This development has in turn triggered the proliferation of mixed hardware/software implementations in the domain of embedded real-time systems. In such systems, part of the functionality of a process is implemented as software (running on a general-purpose processor core) and part of it is implemented by specialised co-processors, formed out of reconfigurable hardware logic. Such mixed systems are often the product of hardware/software codesign.

Functions implemented in hardware nowadays are often complex and take up many clock cycles to execute. In that case, idling the processor while awaiting for the results of hardware computation would be inefficient. Instead, the processor is made available to other processes competing for it. Multiple processes may thus be executing simultaneously on a given instant – at most one on the processor, the rest in hardware. We term this behavior limited parallelism.

For real-time systems, it is imperative that process deadlines be met even in the worst-case. Static timing analysis establishes upper bounds for worst-case process response times; a comparison of those bounds with the respective deadlines is a sufficient (but not necessary) test for schedulability. However, established timing analysis techniques (when applicable at all) are far from accurate when applied to limited parallel systems.

Within this thesis, we formulate static analysis targeted at this class of systems which accurately characterises their timing behavior. Although this analysis stands out on its own merit, we note that is also suitable for use within a hardware/software codesign flow. This matters because mixed hardware/software systems are often developed via codesign.

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank my supervisor Dr Neil Audsley for his guidance, responsiveness, patience and support. His help with obstacles, whether research-related or not, has been decisive.

However, I could not possibly overstate how grateful I am to my parents who encouraged me and provided me with material support.

I am thankful to the people from the Real-Time Systems Group who made it such a great environment for research. In particular, I would like to thank Dr Iain Bate, then Dr Rob Davis for their useful comments and suggestions, on various occasions; also Adam Betts for his help with certain concepts from graph theory and Areej Zuhily for our discussion on multiframe processes. A last-minute thanks goes to fellow computer scientist and friend Margaritis Voliotis who helped with the visualisation of some results.

I am also thankful to Associate Professor Euripides Petrakis (from the Technical University of Crete) for having supported me in pursuing a research degree in York.

Finally, when working towards a PhD, a successful completion is far from certain until the very end. It is a long way and many things can go wrong. In my case, that this specific PhD now reaches its completion, I can only attribute to the intercession of the Theotokos.

Declaration

Certain parts of this thesis have appeared in the previously published papers listed below (the respective principal author being marked by an asterisk):

N.C. Audsley and K. Bletsas* - Realistic Analysis of Limited Parallel Software / Hardware Implementations (10th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 2004), pages 388-395, Toronto, Canada, 25-28/05/2004)

N.C. Audsley and K. Bletsas* - Fixed Priority Timing Analysis of Real-Time Systems With Limited Parallelism (16^{th} Euromicro Conference on Real-Time Systems (ECRTS 2004), pages 231-238, Catania, Italy, 30/06-02/07/2004)

K. Bletsas* and N.C. Audsley - Extended Analysis With Reduced Pessimism For Systems With Limited Parallelism (11^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), pages 525-531, Hong Kong, 17-19/08/2005)

K. Bletsas* and N.C. Audsley - Optimal priority assignment in the presence of blocking (Information Processing Letters, Volume 99, Issue 3, Pages 83-86 (16 August 2006), Elsevier)

1 Introduction

Real-time embedded systems are an important class of information processing systems whose engineering presents designers with significant challenges. This thesis aims to provide a contribution to the understanding of the timing behavior of a particular category of real-time embedded systems.

The term "embedded systems" covers information processing systems which form part of some greater system, whose primary function need not be directly computation-related [22, 28, 49]. For example, an automobile (a composite system whose primary purpose is to provide transportation) may contain multiple embedded computing nodes – some examples:

- An engine controller may fine-tune the firing delay of spark plugs, in response to variables such as engine temperature, fuel octane rating and engine speed (in rotations per minute) [65].
- A cruise controller may provide estimates of fuel consumption under current operational parameters (which may include vehicle speed, engine speed, external air temperature and load) [65].
- An Anti-lock Braking System (ABS) controller may, upon sensors detecting significant difference in the rotational speed of some wheel(s) relative to the others, regulate the diversion of braking pressure away from such a wheel so as to maintain steering ability and reduce braking distance.

In all of the above examples, the respective embedded computing node serves an auxiliary function which helps accomplish the primary function of the system (in this case, transportation). The importance of this secondary function may vary: provision of incorrect fuel consumption estimates to the driver will not prevent the vehicle from being driven. On the other hand, inappropriate spark plug timing

(in automobiles where this is only managed electronically) may cause anywhere from engine knock to permanent mechanical damage. ABS failure may lead to an accident [65].

An embedded system may thus be characterised as *mission critical* (when its failure may compromise the successful operation of the overall system [37]) or as *safety critical* (when its malfunction may cause loss of life, severe damage to the environment or other catastrophic results [16]) or both.

The qualifier "real-time", in reference to a computing system, refers to the fact that computation carried out by the system (usually in response to some external stimulus) is to be completed in a *timely* manner. Untimely completion of computation is, in the context of real-time systems, failure (anywhere from minor to catastrophic, depending on the nature of the application). "Real-time" does not (necessarily) mean "fast" (contrary to the popular usage of the term).

The notion of timeliness varies from context to context but is typically expressed as a *deadline* [22]. A deadline is an upper bound for the time (relative to the initiation of the computation) within which a certain computation must be completed. Deadlines are generally determined by the nature of the respective application (i.e. by real-world considerations). Real-time systems may be classified into two broad categories, according to the degree to which they are able to tolerate missed deadlines:

- In a *hard* real-time system no missed deadlines may be tolerated at all (for at least a subset of information processing activities) [22].
- A *soft* real-time system, by contrast, can handle the occasional missed deadline [22].

Soft-real time systems may be further subcategorised according to different considerations, for example, according to whether belated computation results are of any value or not [22]. Another classification involves the actual criterion which dictates

whether a certain missed deadline matters or not: that may be formulated either in probabilistic terms (i.e. "the probability of tardy completion must not exceed $x\%$") or in terms of a sliding time window (i.e. "within any y consecutive instances of the same computation, no more than $z < y$ may be tardy") [22, 38]. However, within this thesis, we will only be concerned with hard real-time systems.

Real-time systems typically lack a user-interface and are not user-interactive. Input is received via sensors. The various computing tasks are generally repetitive. We can identify two basic behaviors: *event-triggered* and *time-triggered* tasks. An event-triggered task involves computation initiated as a response to an event (such an interrupt triggered by a specific sensor or an exception) [65]. A time-triggered task, by contrast, is one that is released at predetermined instants [65]; associated input is then typically received via polling (of sensors or memory locations).

In the early days of computing, real-time systems were usually implemented as hardwired electronic circuits [58]. Today, they are instead usually implemented as sets of software processes executing on a microprocessor-based platform [58]. Thus, computation in real-time systems is accordingly usually implemented as *sporadic* and *periodic* processes [22]. A periodic process is characterised by its *period*, the regular interval under which it is activated (or, in established terminology, *released*) [22]. A sporadic process is characterised by its *worst-case interarrival time*, which is a lower bound for the interval separating the initiation of any two successive releases of it [22]. For a periodic process, any variability from strict periodicity is referred to as *release jitter*. Release jitter is usually expressed as a respective upper bound for this variability [22].

1.1 Challenges in the engineering of embedded real-time systems

Cost [47, 84, 83, 65] and time-to-market [57, 2, 84, 48, 83, 36, 8, 65] are important factors in the engineering of embedded systems. Thus, embedded system design generally strives either to implement the desired functionality in the least costly (in the fiscal sense) way or, if time-to-market is the main priority, to come up with a "good enough" design within the shortest possible timeframe. Ultimately though, a longer time-to-market results in decreased revenue (which in turn increases per-unit production cost given that nonrecurring engineering costs are fixed) [83]. Some markets are especially sensitive to such effects [83] and the average time-to-market in 2002 for embedded systems was claimed to have shrunk to just 8 months [83].

Goals with respect to cost and time-to-market have to be met subject to other constraints, which vary with the nature of the application and further complicate system design. For example, low energy consumption may be required for some battery-operated system (such as a mobile phone) whereas low weight may be of importance in avionics.

Meanwhile, real-time requirements dictate that considerable attention be given to system performance in the form of both engineering resources and processing power. The system must not only be "fast-enough" most of the time; it must *consistently* meet its timing requirements (as per the definition of a real-time system) in all cases. This is a problem which may not be solvable simply by "throwing" more (or faster) processing hardware at it. Even if possible at all, such a brute-force approach would be too inefficient to be viable (given the constraints imposed on the design by the embedded nature of the system, which must also be addressed).

Careful consideration must thus be given to the interaction of functional and architectural components. The design must either be carried out under an approach which will prevent the emergence of pathogenic behaviors (such as livelock, race

conditions or priority inversion) or, at least, it must be possible for the worst-case effect of such problems to be quantified so as for it to be possible to determine whether the end design will be sound (i.e meeting its real-time constraints) or not.

The choice of an architecture (i.e. the selection of processing elements and other hardware components and the interconnection between them) and the allocation of functionality to the various processing elements are aspects of the design that impact the satisfaction of both real-time constraints and constraints imposed by the embedded nature of the application. However, the designer may further drive any such system configuration towards meeting its timing constraints via appropriate *scheduling* of computation. Real-time systems once relied on rigid schedules (*cyclic executives*) but more flexible approaches have since been adopted (such as preemptive *priority based* scheduling). Given a scheduling policy, *timing analysis* is then employed to derive estimates of system performance. Simulation may be used as well; however, the engineering of real-time systems cannot rely on simulation to verify schedulability as there is no guarantee that the worst-case will have been covered, however extensive the simulation [90]. Instead, static (i.e offline) *worst-case* timing analysis is used. Such approaches derive upper bounds (valid even in the worst-case) for the *response time* of each process (defined as the time from release to completion). Worst-case response timing analysis is usually pessimistic; the elimination of pessimism from analysis techniques continues to be the subject of extensive research.

1.2 The context of this thesis

The complexity of considerations associated with the design of embedded real-time systems has motivated the development of computer-aided design approaches, which take into account the various tradeoffs. The established term for this design paradigm is *hardware/software codesign* and covers various quite distinct approaches. A CAD toolset implementing such an approach is termed a codesign flow.

The objective of codesign is to simultaneously optimise the various design aspects of a system for one another; the hardware is developed as a match for the software (and vice versa). However, the qualifier "hardware/software" also refers to the fact that portions of functionality may be implemented either as software (running on some instruction set processor) or as specialised hardware. The advent of Field Programmable Gate Arrays (FPGAs) [88, 3], which are, essentially, reconfigurable hardware modules, has popularised such mixed systems.

One of the most widely deployed mixed hardware/software architectures involves a single general-purpose processor and multiple application-specific co-processors. The latter implement select portions of process code. All processing elements are capable of execution in parallel; however any given process, on any given instant, may be advancing in computation either by executing on the processor (as software) or by computation on some specialised co-processor – but not both. Due to this characteristic, we have termed this model of computation *limited parallel* [9, 10, 18]. Such systems are the focus of our work.

Existing static timing analysis techniques do not address the semantics of the limited parallel systems in a satisfactory manner; they tend to be too pessimistic (i.e. offline estimations of the worst-case derived by them considerably exceed the actual worst case). Through our work, we have sought to deliver timing analysis which can accurately characterise such systems. Given that such mixed hardware/software implementations are typically the product of codesign, it has been our aim that such analysis be amenable for use within a codesign flow and fit in well with current codesign practice.

1.3 Codesign as motivation

It has been customary, since early on [35], for software designers to take into account the properties of the target hardware or for microprocessor designers to optimise for

specific kinds of software. However, such design practice does not qualify as codesign, as at least one of the two components (be it the software or the hardware) is a given.

The term *hardware/software codesign* describes the **tool-based** [27] design of a system from an initial specification which contains the complete system functionality but where at least certain elements of **both** the software and the hardware component are yet unspecified. A hardware/software codesign *flow* treats the finalisation of those aspects of the design as a joint optimisation problem [35, 28].

Which parameters of the hardware and software component of the system are (or have to be) fixed at the start of the codesign process and which are unspecified or flexible (thus to be finalised through the codesign process) depends both on the philosophy (and the limitations) of the codesign toolset and on the characteristics of the actual application. The same is true of the metrics that the flow aims to optimise.

Some common design metrics for optimisation are: speed of execution, silicon usage, memory footprint, power consumption, communication bandwidth – but also design time. Often, the design must be optimised for multiple metrics. However, the construction of an implementation which simultaneously optimises multiple metrics is generally a difficult challenge [83]. The satisfaction of one metric typically competes with the satisfaction of the other [65]; improvement with respect to one metric often impacts another metric negatively [83].

In that case, the designer often either seeks to meet a certain minimum threshold, with respect to each of the respective individual metrics or tries to optimise for a composite metric (often called a fitness function). These two approaches may even be used in conjunction (i.e. when optimising for a composite metric while seeing to it that some elementary metric is satisfied).

The set of possible alternative designs which may be generated from the initial specification, by respecting the fixed aspects of the design and trying out all possible

configurations of the unspecified or flexible parameters is termed the *design space*. Each such alternative design is then termed a *point in design space*. Although we just defined the design space as a set and the points in design space as elements of that set, the terminology stems from (an equally valid) view of the design space as a M-dimensional space, where M is the number of design parameters (i.e. the individual variables). If the metrics relevant to the design are N, then, for them to be plotted as dependent variables, an (M+N)-dimensional space would be required, and the design space would be represented as a surface within that space.

The immensity of the design space typical in embedded design (except perhaps in the case of small systems) makes it very difficult for human designers to properly address all optimisation targets [28, 34]. Computer-aided techniques (in the form of codesign) substantially facilitate the derivation of better-optimised implementations [28, 34]. However, even codesign flows typically resort to heuristics so as to efficiently traverse the design space [34, 39]. The design space is then sampled, according to some heuristic, in such a way so as to identify the neighborhoods where good solutions lie; the exploration then focuses in those neighborhoods.

1.3.1 Software/hardware tradeoffs and codesign

Embedded real-time systems are often based on hybrid architectures (including both instruction set processors and custom hardware) so as to balance the strong points of both kinds of processing elements. Software implementations are generally associated with rapid development, ease of programming, portability, availability of standard and mature compiler tools, standard (possibly commercial-off-the-shelf) cores, flexibility in terms of updates but also unpredictability of execution times [83]. Hardware implementations, on the other hand, are typically fast [15], in comparison, while at the same time predictable in their timing properties, permit parallelism and consume low power [15, 28] but are costly (in terms of initial development cost and prototyping, especially in the case of fully custom hardware and sometimes also in

terms of silicon) and less amenable to revisions [83].

The tradeoffs regarding the architectural generation and the allocation of functionality to software and hardware are often too complex to be resolved by an ad hoc design approach if close to optimal solutions are to be sought [34, 28]. Additional, non-functional and non-timing related constraints such as battery life, weight, dimensions, acceptable operational temperature ranges add further complexity to the problem [65]. As embedded systems are especially cost-sensitive and usually require fast time-to-market, the industry is increasingly relying on CAD tools for architectural selection and design space exploration, as already noted.

1.3.2 Alternative implementation options

We proceed to offer a summary of some common implementation options generally available to designers:

- **Standard general-purpose cores:** These cores implement a certain, standard instruction set [83] (which allows them to be targeted by readily available compilers) [36]. These cores are typically commercially available as distinct modules. Increasingly often, however (ever since it has become common for entire systems to come on a single chip), they are available as licensable intellectual property (IP). For example, the ARM processor family [4] is both available as IP and as modules (possibly by third-party licensees).

- **Special-purpose processors:** These are auxiliary processors which serve a specialised purpose, as in the case of audio/video encoders/decoders, DMA controllers, communication controllers. These processors are available as standard components and come preprogrammed at the microcode level. An Ethernet controller would fit the description nicely, since it is a standard commercially available component used by third parties in their own products such as motherboards, networked appliances, industrial control systems etc.

- **Custom silicon:** These modules are the product of synthesis on an application-specific basis. They are hard-wired implementations of functionality in silicon. They offer the potential of extensive optimisation (for example, towards very fast implementations) [83]. However, this level of specialisation typically leaves little opportunity for such modules to be reused in systems other than the one they were designed for. This issue, combined with the fact that one-off costs for custom hardware chips are extremely high [83] makes this option particularly expensive [15] (unless these costs are to be amortised over a high volume of chips) [83]. The long turnaround time (for the design and in the foundry, until tape-out) may also an issue, where a fast time-to-market is important [83]. Examples of custom chips are: controller chips in industrial applications or engine controllers in automobiles or chips for mobile phones and calculators.

- **Reconfigurable custom logic:** Field-Programmable Grid Arrays (FPGAs) tend to replace custom logic in areas where the flexibility that reconfigurability provides is important. Currently, it is possible to implement entire systems (including soft cores – see below) on a single FPGA module. In cases where the targeted systems are not to be produced in the volume that would amortise the cost for the development of respective VLSI masks, FPGAs are a low-cost alternative due to their availability as standard off-the-shelf components [83]. Developer tools are provided by the FPGA manufacturers and there is a wide range of modules for designers to choose from. The technology is mature. Custom components in FPGA are often slower than fully custom logic but may still offer speedup [28, 83], compared with software implementations.

- **Soft cores:** An application of FPGAs, soft cores are implementations of instruction-set processors on reconfigurable logic [46, 30]. They combine the benefits of flexibility offered by FPGAs with the use of standard programming languages and compilers. An application that might need just a tiny fraction of the processing power of an off-the-shelf processor could be run on a minimal

implementation of the instruction set (saving on logic at the cost of performance or using microcode traps for certain instructions) or even a trimmed down version thereof [30] (for example, omitting floating-point units in the case of integer-only code). Alternatively, a soft-core may be optimised for low-power operation. In either case, the software developer has access to the full range of mature development tools targeting the original standard architecture. Moreover, soft cores are reusable IP (which brings the cost down) [30]. This technology is increasingly popular.

- **Custom (or reconfigurable) instruction set processors:** For custom instruction set processors, the instruction set itself is a target for optimisation [15, 28]. However, whether the development of dedicated compiler tools is then required or not, has to be considered. Compiler development is typically costly [28], time-consuming and complex. Moreover, in terms of meeting design goals, the use of a processor for which a mature compiler exists may be preferable to an optimised, custom instruction set coupled with a mediocre compiler [28]. For these reasons, typically the core of the instruction set is standard and only a subset thereof is custom. In that case, the development of compiler hooks (for the custom instructions) for an existing compiler might suffice [15]. The logic for implementing custom instructions is often reconfigurable [15]. This allows for reconfigurable instruction set processors to either come as standalone modules (commercial-off-the-shelf, with reconfigurable hardware on-die for the custom instructions) or entirely on FPGA.

Typical codesign

The simplest mixed hardware/software architecture consists of a single instruction set processor and one or more application-specific hardware co-processors connected with each other and with a shared memory by a bus. This architecture is depicted in Figure 1(b) and is derived from the classic von Neumann architecture [86] (depicted

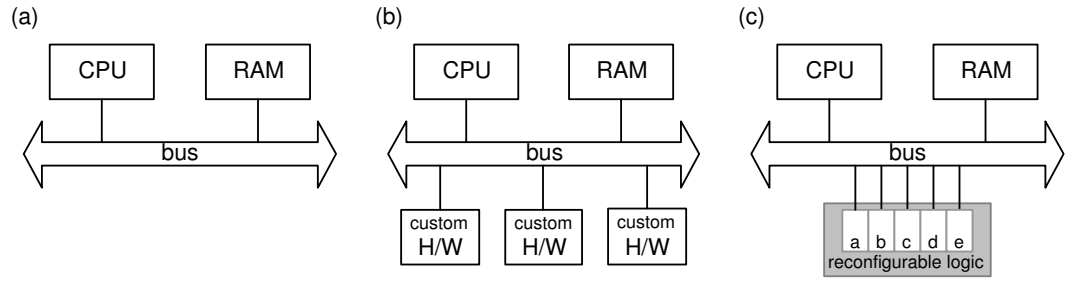

Figure 1: Typical architectures for embedded systems (redrawn from [9, 10, 18])

in Figure 1(a) and consisting of a processor and a memory module, connected by a bus) via the addition of custom hardware. This archetypal architecture, identified since the early days of codesign [27] has been the most studied target platform for codesigned systems and remains popular and relevant to our days. Although there exist examples of mixed hardware/software systems characterised by complex, heterogeneous architectures and interconnection topologies [65], a large share of embedded systems are based on this simple architecture or variants thereof.

Early approaches would often have the processor and the hardware execute in mutual exclusion [27]. However, in the general case, a hardware co-processor may execute in parallel with the processor (and with the other hardware co-processors). Note that the various co-processors need not necessarily come in distinct packages (as in Figure 1(b)). With the increasing deployment of FPGAs, the various logical co-processors may be implemented as partitions of the same FPGA, capable of independent execution in parallel with each other. The typically high pin count of FPGAs would enable each one of them to have a physically separate connection to the bus (Figure 1(c)). Often the entire system, including the bus, the instruction set processor (in the form of a soft core) and possibly even the memory, is implemented on a single FPGA. Alternatively, a hybrid processor containing both a processing core and reconfigurable logic on the same device may be used; an example of an early such hybrid processor is the Triscend E5.

In any case, the partitioning problem (i.e. which functions to implement in software and which in hardware) is largely orthogonal to these implementation options.

While the various codesign flows are quite diverse in their aspects and internals, they share the fact that they are iterative and structured as closed loops. Given a specification (including some constraints), a (non-optimal) point in design space (typically at a high level of abstraction) is chosen as a starting point and is iteratively improved upon until an acceptable solution is found. The initial specification to the codesign toolset includes the functional description of the system, its timing constraints and possibly the architecture and other constraints (for example, some allocation constraints) [35].

The forward loop of the inner loop samples the design space by generating mutated variants of the system configuration initially given to it as input. This process is mostly automated and guided by some heuristic [34, 39] (such as simulated annealing [51], tabu search [42], genetic algorithms or other). Each derivative configuration is then evaluated within the feedback loop and the result influences the next iteration.

After a large (possibly bounded) number of iterations or at whichever point there is sufficient confidence that the design is close to meeting its goals, a candidate design is derived. If found satisfactory, the flow exits successfully. If not, though, the designer will have then typically have to inspect the failed candidate design and, based on the findings, repeat the procedure with another starting point or with different parameters, until a satisfactory solution is derived. This forms an outer loop which, as mentioned, is typically not fully automatic but requires some human intervention [35], unlike the inner loop which is largely automatic and which (due to the high number of iterations) has to be fast.

Codesign flows may be classified according to the following characteristics:

- The language/model in which they require the initial functional specification

to be represented.

- Whether they allow real-time constraints to be explicitly specified and whether these guide the process.
- The architecture targeted and whether this is a given or architectural exploration is performed as well.
- The design metrics for which design optimisation is possible.
- The heuristics employed for design space exploration
- The timing analysis used.
- The computational model which they are based on and their internal representation of a candidate system configuration.
- The degree in which they are automated.

The above list is not exhaustive. We will proceed to discuss some notable approaches to codesign using the above classifications for comparison.

1.3.3 Early approaches to codesign

Among the earliest approaches to hardware/software codesign is **Vulcan**, presented in [45]. Early implementations of real-time systems (i.e. before the advent of powerful microprocessors) were very often hardware-based and "hard-wired" in terms of functionality [58]. Having been developed not much later those days, the approach in [45] is rather hardware-oriented designs. For example, the starting point for the codesign flow is a fully hardware-based implementation of system functionality. What was novel for the time however is that their flow iteratively extracted functionality from hardware and reimplemented it in software, which ran on an instruction set processor. The target architecture then consisted of the hardware, the processor and the system memory, all connected by a shared bus.

The starting point for the codesign flow was an implementation which did meet the timing constraints of the application. The objective of seeking a mixed implementation was to optimise other design parameters: a somewhat slower – but still meeting its timing constraints – mixed implementation with reduced silicon footprint, under obvious constraints such as the bandwidth of the bus and the processing capacity of the processor. Codesign was thus treated as an optimisation problem and candidate designs were evaluated by use of what could be described as a cost function.

In this approach we identify many characteristics which later came to be typical of codesign approaches, namely the iterative nature of the design flow, the use of a suboptimal implementation as a starting point, the use of specialised metrics for the quality of the design. However, we see that it involved very simple designs, and that it very fine-grained (which would not be very practical with larger systems). In particular, the scheduling approach is simplistic in that the hardware and the instruction set processor execute in mutual exclusion, never simultaneously. Moreover, hardware operations are to be scheduled according to a fixed schedule. Software scheduling may be dynamic (via list scheduling) but not preemptive. The system functionality is structured as a set of single-rate graphs of processes (each graph modelling the partial precedence constraints between processes). End-to-end latencies are then derived by graph analysis, with some degree of uncertainty because latency bounds to some operations (processes) are assigned arbitrarily.

Another early approach, in many respects archetypal of codesign, is **COSYMA** by Ernst et al. [36]. A notable difference from the previously discussed approach is that the starting point is an implementation of system functionality entirely in software; portions thereof are then iteratively moved to hardware. Here, the starting point in design space is one for which timing constraints are not met and the objective of the codesign flow is to move to a design where timing constraints are met but hardware budgets are minimised. The initial functional specification is input in C-like code. Here, we observe that the use of general-purpose programming languages has since

then become the norm. The use of hardware definition languages (HDLs) is generally not dominant any more (unlike, what was the case for Gupta et al. [45]), except for certain niches. Both HDLs and general purpose programming languages are capable of expressing behavioral specifications but the latter approach is considered more flexible and higher-level.

This initial approach of Ernst et al. is very fine-grained; the granularity is that of the **basic software block** (BSB), i.e. a series of statements which does not contain branches. Since then, the need for codesign of much larger systems with exponentially larger design space, has favored more coarse-grained approaches. Nevertheless, in place of BSBs, it is often larger code constructs with a single point of entry and a single point of exit which are used as the unit of granularity.

Ernst et al. also employ heuristics (namely *simulated annealing* so as to effectively traverse the large design space. The use of such heuristics (such as annealing, tabu search, genetic algorithms) has since become commonplace in codesign, so as to cope with ever-increasing design space. Another characteristic which subsequently found wider adoption is the use of co-simulation (that is joint simulation of software and hardware).

Whereas simulation may indeed be useful in uncovering design weaknesses, it may never be proven that all cases have been tested - and especially so regarding timing constraints being met. Instead we believe that, regarding the codesign of hard real-time systems, such systems must be **valid by construction** - possibly with the use of static analysis.

The scheduling model used within the codesign flow proposed in [36] is rather simple (either list scheduling or "as soon as possible" scheduling) and so is the timing analysis (graph flow analysis). As with Gupta et al. [45], computation in hardware and software may not be carried out in parallel. We consider this practice wasteful of system resources. The target architecture is the same as for Gupta et al. [45]. The system is single-rate, which is restrictive in that it cannot accommodate com-

petitively scheduled multirate processes.

1.3.4 More contemporary approaches

The codesign environment **SpecSyn** [40] was novel in that it opted for a two-stage approach: Most of the design space exploration takes place within the first stage; points in design space are evaluated within a fast loop; this loop uses performance and cost estimates derived earlier by a preprocessor for the various functional objects involved. These estimates are in turn dependent on design libraries (required) and are "accurate enough" [40] to be used as input to an evaluation loop engineered with the goal of fast design space traversal, rather than accuracy, in mind. Different partitioning heuristics are supported by the system, so that if one heuristic fares badly another one can be tried. The second stage is more accurate in its derivation of timing estimates and involves co-simulation of hardware and software. By co-simulation, it is meant that a hardware simulator and a software execution simulator execute in lockstep, with ones output serving as input to the other.

The flow does not deal with architectural exploration; the selection of an architecture is left to the designer. Instead, it deals with functionality allocation and interface and communication optimisation. An important restriction is that there is no feedback from the second stage to the first. Other restrictions, though not fundamental weaknesses of the approach, include the limited choice for candidate architectures and the reliance on component libraries.

A strong point of SpecSyn which was identified, was the emphasis on having a good understanding of the design space before actually specifying candidate designs in detail. This way, the exploration is performed faster and more areas in design space are traversed within a given timeframe. Yet, with regard to targeted at hard real-time systems we note that no explicit guarantees of schedulability in the worst-case are either sought or derived and that only a limited choice of (simple) scheduling

approaches are available.

SpecSyn expects its input to be specified in VHDL or SpecCharts [85], an extension of VHDL inspired by the modelling language Statecharts. SpecCharts is built around the concept of Finite State Machines and may be automatically translated to VHDL.

POLIS [14] practices (and advocates) codesign based, essentially, on a variant of the *synchronous* (or *reactive*) computational model. It does not accept as input a functional specification based on a general-purpose programming language. Rather, it expects its input in the Esterel language (typically) but also Statecharts or (subsets of) VHDL and Verilog. This input is then converted to the native formal specification model of POLIS which is based on the concept of the CFSM (i.e. *Codesign Finite State Machine*). The CFSM model is introduced in [55, 56]. Given that CFSM notation is primarily intended to be machine-readable, it is terse [32] for humans, which is why an Esterel frontend is used.

The computational model of POLIS rectifies some of the identified [6, 32] shortcomings of the reactive programming model. We proceed to briefly discuss the synchronous or reactive computational model, Esterel and how they relate to POLIS.

The "classic formulation" of the synchronous model represents the system as a Finite State Machine (FSM). The system responds to input *events* and emits events in response which, if not consumed (by the system) are then emitted as output events (determining the behavior of the system). Because the processing of events (i.e state transitions and corresponding event emissions) is instantaneous [32], the emitted event occurs *at the same instant* as the event which triggered it. This is not realistic (as in the real world any processing takes non-zero time) but is handled in implementations by dividing continuous time into timeslots, with inputs read at the start of each one and outputs being generated at the end [6, 32]. This, in turn, requires a global clock, which is sometimes difficult to implement in large distributed systems [89] and might result in inefficient design [6]. Moreover, the

underlying mechanism relied upon to obtain the above described semantics (i.e. the synchronicity of input and output, given also that contention for shared resources may occur, which is not accounted for by the model) is that of a cyclic executive, an approach which comes with its own problems [16] (which are then immediately inherited by the reactive approach [6]). The synchronous or reactive model also often suffers from poor code density [6], which might be a concern for embedded systems.

Languages targeting the reactive computational model include Esterel [17], Statecharts (which is graphical), LUSTRE, SIGNAL. These languages are completely unlike general-purpose programming languages (typified by C) but exhibit some conceptual similarities with VHDL. Thus, the code primitives of Esterel map well to hardware constructs (signals become wires, pauses become registers, instructions become combinatorial logic and so on) [32]. However, the compilation to software is, essentially, a C function, executing once per cycle and simulating the corresponding hardware [32]. Note though that it is the responsibility of the Esterel programmer to ensure the program is fast enough to execute within a single cycle [32].

In POLIS, the authors remove the assumption of instantaneous state transitions, which eases implementation somewhat, but still rely on a global clock. Additionally, the compilation of the CFSM notation to software is more natural [32]. However, despite these improvements, for large programs, output software suffers from poor code density [32] and there are also difficulties in hardware synthesis from large programs [32].

POLIS was developed dialectically with Ptolemy [33, 21], a powerful simulation engine compatible with a variety of computational models, developed within the same academic department. POLIS employs Ptolemy as its simulation engine of choice (though its architecture does not tie it to any specific simulation engine) and benefits from the scalability of Ptolemy with respect to large systems and also its ability to co-simulate components defined in terms of quite a variety of computational models.

The basis of the POLIS approach is nevertheless its computational model.

POLIS uses an automated approach for the generation of the low-level aspects of implementation (such as communication interfaces, protocols and drivers), via the use of component profiles. However, we note that the timing behavior of candidate designs is for the most part tested by simulation.

A state-machine based specification is also encountered in the codesign tool **MUSIC** [12, 24] but, irrespective of this, this codesign flow has some interesting characteristics. The approach places particular importance on systems communications. It is argued that for the codesign of complex systems-on-a-chip to be efficient and tractable at the same time, system code (i.e. system calls, schedulers, kernels) must be separated from application code (to which it has been traditionally been bound). This is to be accomplished by the use of communication wrappers (which may have a software and a hardware part), based on predefined libraries but generated automatically on an application-specific basis. This allows the system to be treated as a hierarchy of interacting (software and hardware) components and the timing properties of each component are derived by those of its subcomponents (both in analysis and in simulation). It is claimed that this approach has been shown to be scalable and allows for fast design space exploration [12].

MUSIC explores some architectural options but does not go as far as exploring alternative architectural topologies. It is coupled with the simulator GEODE. For simulation, functional models of the application (in VHDL/C for hardware/software respectively) are used which are generated automatically from the initial specification of the system, which is in SDL, a formal object-oriented modelling language).

Once again though, we see that although execution speedup is sought, there is no explicit notion of specific, hard timing constraints which have to be met. Moreover, such constraints are, to the best of our understanding, not part of the initial specification. Thus, whether they are met or not, enters into the picture at a later stage. There are also some issues with the approach to timing analysis. It seems that more

attention is given to simulation results (rather than analysis) for the purposes of evaluating a design. This, however means that there is no way of knowing whether the worst-case is has been covered until reaching that late stage. Moreover, the estimations derived are reasonably accurate but may be optimistic [12], which is problematic for the engineering of hard real time systems. Overall, what we appreciate the most in this approach is the use of abstraction in design space exploration and the decompositional approach (i.e. timing properties of some component being derived by those of its subcomponents).

Magellan [26] is another codesign environment of note. Particularly interesting is the fact that it appears to have been developed with architectures based on reconfigurable hardware in mind [25]. The approaches that we have mentioned so far, either assumed that the custom hardware and the processor core(s) come in distinct packages or made no assumptions whatsoever whereas Magellan as a flow appears more optimised for architectures where the hardware and processor cores are as closely coupled as FPGA-based implementations permit them to be. The architecture targeted by Magellan is multicore; all processing elements (whether processors or hardware co-processors) are connected by a single shared bus. Processing elements may also have separate local memories but there is at least one shared memory module, connected to the shared bus, for the purposes of interprocessor communication. This architectural template is quite flexible but it is the designer, and not some architectural exploration procedure which has to specify the architecture. It must be noted that the codesign flow is targeted at data-processing applications (i.e. with mostly data dependencies between tasks and few control dependencies), such as JPEG and MPEG algorithms. It is thus not appropriate for systems with competitively-scheduled multirate processes, as is often the case with industrial control applications. The application is modelled (the model being derived from the input specification, which is to be expressed in a high-level behavioral language) as a hierarchical control-dataflow task graph. Several pre-determined cat-

egories of tasks exist (some of them composite); the various subtasks forming the overall application are classified by the flow as falling into some category from among those. This classification is then taken into account during the iterative optimisation of the candidate design via the application of some clever techniques built into the codesign flow. Particular attention is given to loop optimisations. Other optimisations include parallelisation, co-processor reuse and pipelining. The codesign flow tests and evaluates various allocations and schedules, its goal being the minimisation of the overall latency of the application graph, subject to processor usage and silicon budgets.

While the codesign flow appears to do well what it is designed to do, we note that its scope is somewhat limited. Data processing, where tasks are scheduled cooperatively and not competitively, appears in a sizeable share of embedded real-time systems. However, far more challenging is the codesign of systems with competitively scheduled processes or where data processing only forms part of the functionality. Interactions between processes in such systems complicate the timing analysis and make the verification of timing correctness difficult. Magellan is not able to cope with such systems because it offers only limited scheduling options (such as non-preemptive and rather inflexible schedules).

We conclude with the discussion of a timing analysis technique formulated with codesign in mind.

The approach detailed in [67] is not a full-fledged codesign flow but, rather, an analytical technique suitable for deployment within the inner loop of a codesign flow (i.e. the loop performing the design space exploration) for the evaluation of points in design space.

The analysis [67] accepts as input a specification of the system in the form of a set of conditional control/dataflow process graphs, with the allocation of tasks to processing elements as a given.

Each conditional graph is single-rate and specifies a partial ordering of operations (dictated by data and precedence constraints. The architecture is in the general case distributed and heterogeneous. Operations are to be scheduled subject to the precedence constraints and are scheduled according to a fixed-priority scheme. The analysis draws from [90] and improves on that work. The upper bounds derived for response times are safe (i.e. cover the worst case) while, at the same time, being tighter than those under more traditional approaches (typified by [54]). This is because, if the precedence constraints are taken into account, it is possible to deduce that certain activities may not interfere with certain other ones. Algorithms are provided which safely identify such patterns, thus reducing the pessimism relative to less sophisticated approaches.

The computation of upper bounds to response times according to this approach may be performed fast enough for the analysis to be integrated within a fast, inner codesign loop, as intended. This is important because it enables static timing analysis to accompany (and even guide, as we argue) the design space exploration. Thus, unschedulable solutions are immediately discarded, as soon as they are identified as such by the timing analysis. The addition of such analysis into the design loop would not significantly affect the high throughput rate of the codesign flow (in terms of points in design space evaluated over a time interval, or equivalently, the turnaround time for one iteration). Instead, time is not wasted on the generation of interfaces, implementations and simulations of candidate designs that would subsequently be found infeasible.

We mentioned earlier that allocation (of functionality to processing elements) is a given when the analysis is conducted. This is not at all at odds with codesign, as the allocation in consideration, as well as the process priorities, may have been derived via codesign. In fact, the earlier work which in part motivated the contribution of [67] has looked into solving allocation, within the context of codesign, via use of heuristics such as *simulated annealing* and *tabu search* [34], given resource

constraints. Therein, cost functions were used to guide the heuristic in between iterations of the codesign loop but the analysis from [67] might be used instead within such a codesign flow of such a kind as it appears fast enough to not slow iterations down. The scheduling of communications, as a codesign problem, over a TDMA bus is addressed in [66], whereas, in terms of software scheduling, we note the progression from list scheduling and scheduling tables, in earlier related literature [34, 31], to fixed-priority scheduling in[67]. The approach is covered in detail by Paul Pop in [65].

1.3.5 Issues within codesign practice

Clearly, each of the approaches to codesign discussed has its relative strengths and shortcomings. After all, the particular classes of mixed hardware/software systems targeted may differ and the same is true of the design metrics targeted. However, we note two main issues:

First, we see that, generally, whether the target system meets its real time constraints or not, is accounted for late in the design stage. A candidate design has to first be finalised to a significant degree (which is costly and slows exploration) before it can be determined whether real time constraints are met or not. If not, then there is no straightforward way to determine what should change in the design for schedulability to be attained. Instead, we maintain that systems must be *valid by design* and insist that a good way to enforce this would be to conduct the timing analysis within the inner loop of the codesign flow. This means that such analysis must then at the same time be tractable (so as not to slow down the loop), safe (i.e. not optimistic) and sufficiently accurate (given that many implementation details will be unknown at that point and pessimistic assumptions will have to be made – such as regarding process offsets, for example). This brings us to our second main observation:

The timing analysis techniques used in codesign were not originally devised with the properties of (or opportunities offered by) mixed/hardware software systems in mind (such as the potential parallelism between software and hardware). Thus they tend to be pessimistic.

In our opinion, the approach to scheduling and analysis (in the context of codesign) suggested by [67] does stand out because of the degree in which it tries to address both of those concerns. Such analysis is suitable for integration into a design space exploration loop, for the purpose of establishing the fitness of points in design space, without slowing it down. Moreover, the potential for parallelism is exploited and this is accounted for, to considerable degree, in the associated worst-case timing analysis. These contributions, however, are focused on the scheduling and interactions of processes (i.e. blocks of functionality) belonging to the same graph, which is single-rate. The understanding of interactions of processes belonging to different graphs (thus with different release rates) does not see any advance.

1.4 Main hypothesis

The main goal of the research presented within this thesis is to provide a characterisation of the temporal behavior of a limited parallel system.

Therefore, the objective of this thesis is to validate the hypothesis that:

Static timing analysis for both the worst and the best case can accurately characterise the timing behavior of limited parallel systems.

1.5 Thesis structure

Chapter 2 provides a survey of literature relevant to the scheduling and schedulability analysis of real-time embedded systems. We conclude with the identification of open research issues related to timing analysis in the context of codesign.

Chapter 3 provides additional motivation for the research issues that we will be coping with and states some of our assumptions.

Chapter 4 introduces our core contribution, the *limited parallel* computational model and explores its semantics. Basic worst-case timing analysis tailored at this computational model is also formulated therein. Some evaluation against established timing analysis is also performed.

Chapter 5 expands on the contributions of Chapter 4. More accurate worst-case timing analysis is provided, addressing the main shortcomings of the basic analysis introduced earlier. Comparisons with both the established timing analysis and basic analysis for the limited parallel model are included.

Chapter 6 complements our contribution to the area of worst-case response time analysis with a formulation of a best-case analysis technique, which relies on the same concepts as our worst-case analysis.

Within Chapter 7 the issue of optimal priority assignment is addressed. The priority assignment algorithm formulated therein employs our timing analysis as the feasibility test and is optimal for limited parallel systems in the presence of blocking (when access to shared resources is managed by the Priority Ceiling Protocol (PCP) [69]). An additional contribution is the proof of optimality of the Deadline Monotonic Priority Ordering scheme (DMPO) for conventional uniprocessor systems with shared resources managed under the PCP.

We conclude (Chapter 8) with a summary of contributions and a discussion of possible future work. Some preliminary work on multiprocessor variants of the limited parallel model (one of the identified directions of future research) is presented in the Appendix.

2 Literature Survey

Having pointed out in Chapter 1 the issues, within current codesign practice, related to the use of timing analysis, we now proceed to a survey of timing analysis per se. Within this chapter we track the evolution of scheduling techniques and corresponding approaches to the static timing analysis of embedded real-time systems. As will become evident, despite continuous advances, the characteristics of systems with limited parallelism fail to be addressed properly, which results in considerable pessimism.

Before the development of *fixed priority scheduling* (FPS) and *earliest-deadline-first* (EDF) scheduling, designers had to rely on rigid *cyclic executives*. A cyclic executive [22] is a rigid allocation of timeslots to each processing activity over the course of a given time window (called the *major cycle*). This schedule repeats itself indefinitely. A major cycle is typically divided into a number of isochronal *minor cycles*. This information may be represented in the form of a table.

While cyclic executives are still in use in the industry, we will not be considering them in our survey, as it is our intention that the contribution of this thesis be based on the theory pertaining to process-based scheduling. We thus proceed with a survey of *worst-case* static timing analysis techniques for systems with FPS and EDF. We also look at shared resource management schemes and examine how the effects of shared resources and blocking are to be quantified, under each of them. We conclude with a discussion of holistic analysis techniques and also best-case timing analysis (an area which has received little attention so far).

2.1 Timing analysis under fixed priority scheduling

Under fixed priority scheduling, processes in a system are assigned static (i.e. not subject to change at run time) priorities. On any given instant, the process with

the highest priority gets to execute on the processor, among those competing for it.

A seminal paper by Liu and Layland [54] is generally considered as the cornerstone [75] for all subsequent research on *fixed priority scheduling* and associated static timing analysis. The model assumed therein contained rather restrictive assumptions, however most of the progress since then has been made by removing or relaxing these assumptions, while otherwise retaining key elements of that contribution.

2.1.1 The process model of Liu and Layland and its associated timing analysis

The process model assumed by Liu and Layland [54] expects strictly periodic processes, executing on a single processor. These are independent (i.e they share no resources and they have no precedence constraints) and fully preemptible, whereas preemption overheads are negligible. There is a (known) upper bound to the computation time of each process and the deadline of each process is equal to its period. Finally, processes may not voluntarily suspend their execution.

Even for such a restrictive model, determining, for all possible combinations of relative process release offsets, whether a system is schedulable (i.e. whether all deadlines are met, even in the worst-case) is NP-hard [41]. Thus, in the general case pessimistic (i.e. sufficient but not necessary) tests for schedulability are employed.

Liu and Layland [54] are credited with being the first to establish a method for deriving upper bounds for the worst-case process response times (under the stated assumptions). Bounds derived under their approach are, in the general case, pessimistic (i.e. the actual WCRTs may be lower. Nevertheless, a comparison of the derived bound with the respective deadline, for each process, constitutes a sufficient (but not necessary) schedulability test.

The validity of those bounds is established according to the *critical instant theorem*,

the most important concept in that analysis. This theorem identifies the scenario (i.e. combination of relative process release offsets) under which interference is maximised. This scenario involves coincident releases of all processes and maximises the response time of each one of them (if, additionally, process activations execute for as long as their respective WCET).

Then, process response times derived by simulation of the system under such a critical instant scenario are valid upper bounds to the actual respective WCRTs.

Within the same paper, Liu and Layland also derive a utilisation-based schedulability test. The processor utilisation of the system is defined as

$$U = \sum_{i}^{n} \frac{C_i}{T_i}$$

where n is the number of processes, C_i is the worst-case (i.e. maximum) execution requirement for the i^{th} process and T_i is the period of that process. (Processor utilisation is often equivalently given as a percentage (i.e. $U = 0.6$ is equivalent to $U = 60\%$) but, within the context of this thesis, we will be using the former representation of the two.)

According to this test, if

$$U = \sum_{i}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

then there exists some assignment of priorities to processes under which the system is feasible (i.e all deadlines, which are assumed equal to the respective process periods, are met) under any possible combination of relative process release offsets. This feasibility test is sufficient but not necessary (as was also the case with the response time based test) and process sets exist with higher processor utilisation (but still below unity), which are feasible.

Regarding the problem of priority assignment, within the same paper, Liu and Layland showed that the *Rate Monotonic* scheme (under which, the smaller the period of a process is, the higher its assigned priority is) is *optimal*. For a priority assignment algorithm, this property of optimality is met when, for any system, if the system is feasible under any priority assignment other than the one derived by said algorithm, then the system will also be feasible under the assignment derived by said algorithm. This property was proven therein [54] for the Rate Monotonic scheme, under the stated assumptions.

2.1.2 Refinements to the above process model and timing analysis

Leung and Whitehead [53] removed a major restriction, that of the process deadline being equal to the respective period and demonstrated that, in systems with deadlines equal to or less than the respective process periods (abbreviated $D \leq T$), the optimal priority assignment scheme for the resulting class of systems is the *Deadline Monotonic Priority Ordering* (DMPO). This more general algorithm encompasses the earlier Rate Monotonic scheme as a special subcase (i.e. $D = T$).

While utilisation-based feasibility tests are no more applicable when the deadline of a process is less than its period, tests based on the calculation of (upper bounds on) process response times remain applicable.

Joseph et al. [50] and Audsley et al. [5] independently developed an arithmetic method for the calculation of response times under the critical instant scenario (instead of a simulation-based derivation). According to this, upper bounds on process response times are given by the fixed-point equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

where $hp(i)$ is the set of higher-priority processes for τ_i. The above equation is solvable via the formation of a recurrence relation:

$$w_i^{(n+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^{(n)}}{T_j} \right\rceil C_j \tag{2}$$

The recurrence is initiated by $w_i^{(0)} = 0$ and converges after a certain number (k) of equations when $w_i^{(k+1)} = w_i^{(k)}$. Then $R_i = w_i^{(k)}$ is the least solution to Equation 1 and a valid upper bound for the WCRT of τ_i [11].

Note that the optimality of DMPO for the above class of systems holds only if the system analysed is *synchronous* (i.e. when a coincident release of all processes occurs within any time window of length equal to the least common multiple of all process periods). If this condition is not met, then it is possible that a process set which would be feasible under some priority assignment other than DMPO, would cease to be so under DMPO. This is demonstrated by example in [53]. Another such example that we came up for illustrative purposes is given in Figure 2.

In an *asynchronous* periodic system, a *critical instant* is not observable. Thus, such systems are usually described in terms of relative release offsets between process releases. The same relative release offsets between any two processes are observable once every least common multiple (LCM) of their process periods. An exact feasibility analysis of such systems requires the construction of a schedule twice as long as the LCM of all process periods [52].

When there is no knowledge of the actual offsets (or, when these are disregarded, so as to ease analysis at the cost of pessimism), a critical instant may be assumed. This assumption leads to a potential overestimation of process response times.

Then, the DMPO may be considered optimal in a limited sense – for which we introduce the term *offset agnostic* optimality. Thus, if the system is schedulable under some priority ordering other than DMPO under the additional assumption of a critical instant, then it is also schedulable under DMPO under any possible combination of relative release offsets. However, even if this is established, it is not useful for deriving an optimal priority ordering for the purpose of establishing the

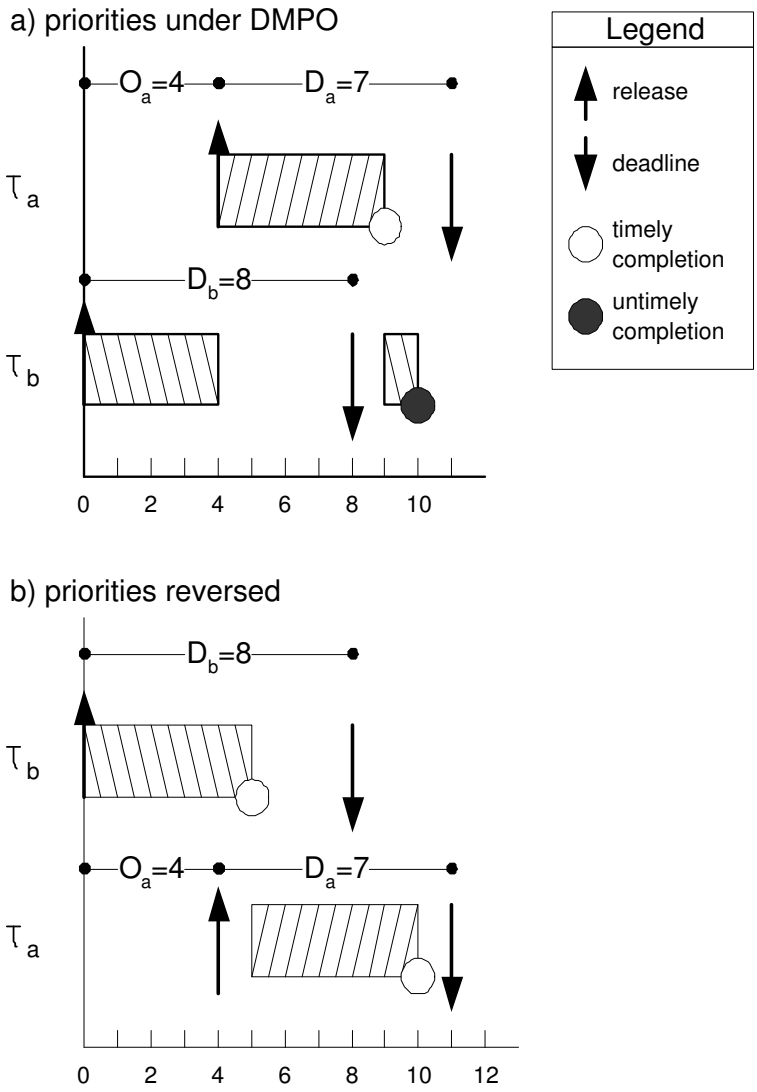

Figure 2: DMPO is not necessarily optimal for asynchronous systems

feasibility of some system under some given combination of relative release offsets.

Leung and Whitehead suggested in [53], as the only way of establishing an optimal priority assignment for asynchronous systems, the enumeration of all possible priority assignments ($n!$ for a system of n processes). Audsley [7], however, introduced an optimal priority ordering for asynchronous systems whose complexity is $O(n^2 + n)$.

Under this algorithm, all processes are in turn tested for feasibility at the lowest priority level until one is found feasible. This process is then assigned that priority level and the procedure repeats itself with remaining processes and remaining priority levels (traversed from lowest to highest). Either, then, an optimal priority assignment is established when all processes have been assigned a priority level, or, if a priority level is reached at which none of the remaining processes is feasible, the system is deemed infeasible.

As an additional contribution, the formulation of this algorithm showed that the NP-hardness of the problem of establishing whether a feasible priority ordering exists is due to the feasibility test, not the priority assignment. Moreover, said algorithm, determines the minimum number of distinct priority levels necessary for a given system to be feasible. Permitting multiple processes to share a single priority removes another one of the original restrictions placed by Liu and Layland and eases implementation.

2.1.3 Shared resources and blocking

So far, it has been assumed that processes are independent. However, in practice, this rarely holds, except perhaps in very simple systems [22]. Typically there exist shared resources (such as devices, memory locations and buffers) which must be accessed in an atomic manner. This reality necessitates some mechanism to enforce this mutual exclusion. Such shared resource management schemes typically implement access synchronisation by causing processes to potentially encounter *blocking*

upon attempting to access a shared resource already in use by some other process. The side effect of such a solution is the behavior termed *priority inversion*: a high-priority process potentially having to await a lower-priority process to release a resource so that it may advance in computation (contrary to the behavior that would have been desirable, given their relative priority ordering). Priority inversion, if not bounded, may compromise schedulability, as highlighted by the following scenario:

Some (high-priority) process τ_A preempts a (low-priority) process τ_C but, shortly after, is blocked upon attempting to access a shared resource already in use by τ_C. However, τ_C may in turn be preempted by other processes (of priority greater than that of τ_C but less than that of τ_A) and this will add to the time that τ_A spends blocked. If that time is sufficiently large, the deadline of τ_A may be violated.

The various shared resource management schemes based on the concept of *priority inheritance* (first introduced in [76]) were devised as an attempt to bound the effects of priority inversion. This is accomplished by boosting the priority of a process (according to some policy) whenever it is blocking (or, alternatively, depending on the scheme, when it might potentially block) a higher-priority process. In the context of uniprocessor systems, this work has culminated in the *Priority Ceiling Protocol* (PCP) [68] in its two variants: the *Original* (OCPP) and the *Immediate* PCP (ICPP) [22].

The Priority Ceiling Protocol associates with each shared resource a static value termed a *ceiling*, defined as the static priority level of the highest-priority process, from among those using the resource. Whenever a process enters a critical section guarding a shared resource, its run-time priority is raised either (under the ICPP) to the ceiling of that resource (immediately upon entering the respective critical section) or (under the OCPP) to the priority of any higher-priority process which it blocks (whenever such a block occurs). Additionally, a process may only access a resource if its run-time priority is higher than the ceilings of all resources currently in use by processes other than itself (otherwise, it blocks).

This protocol is sufficient to prevent deadlock and bound the time that any activation of a process may spend blocked. This bound is limited to the length of the longest critical section from among those guarding a resource which is shared by at least one lower-priority and at least one equal- or higher-priority process (including the process in consideration itself).

By the inclusion of this worst-case blocking term (B_i), Equation 1 is updated to Equation 3

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

which is, again, solved via the formation of a recurrence relation, as before.

The above equation, however, echoes another restrictive (though not related to resource sharing) assumption of the original process model of Liu and Layland [54], that of the strict periodicity of processes in a system. This assumption may be relaxed by modelling the variation in periodicity of some process τ_i as *release jitter*, for which an upper bound J_i exists. This worst-case release jitter was incorporated into the equation for deriving bounds on worst-case response times by Audsley et al. [11] as

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{4}$$

2.1.4 Holistic and best-case response time analysis

An often cited paper by Tindell et al. [80] is generally credited with introducing a method for employing the analysis originally developed for uniprocessor real-time systems so as to derive worst-case process response times in a distributed system.

The process model assumed therein involves a system comprised of multiple (possibly heterogeneous) processors and multiple processes, each statically allocated to some

processing element. Scheduling is conducted on each processing node according to a fixed-priority scheme. However, a process may be either *time-triggered* (i.e. periodically released) or *event-triggered* (i.e. released as a response to some event emitted upon termination of some other process, possibly on some other processor). Thus, notional chain-like *transactions* formed by process activations are formed. The first process in such a chain is time-triggered; subsequent ones are event-triggered.

While the sustained rate for the release of each event-triggered process is determined by the release rate of the first process in the respective transaction of which it forms part, an event-triggered process is not strictly periodic because it inherits the variability of the completion time of its predecessor process, within the transaction, as a release jitter. The establishment of a bound for such jitters would permit the analysis of the overall distributed system by analysing each processing node separately as a uniprocessor system, scheduled under a fixed priority scheme. The approach described in [79] (and, again, in [80]) accomplishes this as follows.

Worst-case response time analysis is carried out on each node with release jitters initially treated as zero. After this round of analysis is completed, release jitter values are updated to match the output jitter (i.e. termination jitter, relative to the release of the transaction) computed for the respective triggering process during the round. The procedure is repeated iteratively until all output jitters (and thus also release jitters) converge to stable values, at which point valid upper bounds on process WCRTs may be derived.

The approach of Tindell et al. also involves a round of communication analysis, "sandwiched" between successive rounds of processor based timing analysis. This accounts for the distributed nature of the system, where events are transmitted as messages. In a distributed system, the transmission time for a message is typically neither negligible nor invariant. This variation adds to the release jitter of the triggered process.

In [80], it is assumed, for the purposes of communication analysis, that message

transmission takes place over a TDMA (i.e. *Time Division Multiple Access*) bus. In [81], however, dedicated communication analysis for *Controller Area Networks* (CAN buses) is formulated. That analysis draws from classic analysis for fixed-priority scheduled systems; the bus mirrors the processor, and messages (characterised by static priorities) mirror processes. A notable difference is that message transmission is not preemptible (unlike process execution). This approach allows for upper bounds on end-to-end message latencies to be derived. That analysis has seen since been augmented with probabilistic guarantees of fault tolerance – see [20].

Refinements to the holistic analysis of Tindell et al. by Palencia et al. were formulated in [64]. There, the notion of a local deadline is introduced (that is, a deadline, for each process forming part of some linear transaction, which is measured relative to the release of that process – as opposed to the release of the transaction) and analysis for meeting such deadlines is formulated. Note that both the original holistic analysis by Tindell et al. and the later analysis by Palencia et al. permit the end-to end deadline for the transaction to exceed the respective period.

We mentioned how the holistic analysis is based on the calculation of output jitters for processes. An upper bound for the output jitter of some process τ_i (if communication delays are negligible), may be computed as

$$output\ jitter = J_i + (R_i - \hat{R}_i)$$

where J_i is a upper bound for the release jitter of τ_i, R_i is an upper bound on its WCRT and $\hat{R}_i$ is a lower bound on its BCRT. Such a derivation in turn necessitates an analytical approach to the derivation of bounds on process BCRTs. This area, however, has so far received limited attention.

Palencia et al [63] note that, prior to their contribution (formulated therein), arbitrarily small (i.e. zero) best-case response times had been assumed. They thus formulate two approaches: the one (aptly) termed "trivial" uses any lower bound to

the actual best-case *execution* time (BCET) of the process as a lower bound for its BCRT. The more advanced approach is based on equations (solved via formation of a recurrence relation) describing a best-case scenario (thus mirroring established WCRT analysis). Redell et al. [73] subsequently came up with analysis based on a different best-case scenario; that analysis outperforms that of Palencia et al. and, under certain conditions, is exact. We will revisit those two approaches in detail prior to the formulation of our own contributions to the area of BCRT analysis within Chapter 6.

2.1.5 Timing analysis of process sets with multiframe processes

The response time analysis techniques covered so far within this literature review do not account for the possibility that different activations of a given process could *a priori* be characterised by a different worst-case execution time. Yet, it is possible (and might make sense from the designer's point of view) for some process to be engineered this way. For example [13], in some industrial applications, a process may periodically perform data collection but only store this data once every N activations. Those activations of the process which only deal with data collection will be characterised by a much shorter worst-case execution time compared to those activations also dealing with the computationally intensive storage operation. This would then constitute a typical example of a *multiframe process*.

A multiframe process, according to the original definition [60], is one for which the worst-case execution time varies according the following general pattern:

Given any sequence of successive activations of said process τ_i (with ${}^aC_i$ denoting the actual execution time of the a^{th} activation within said sequence, a being a positive integer), an array of N integers $\{C_i^{(0)},\ ..., C_i^{(N-1)}\}$ exists such that

$$ {}^aC_i \leq C_i^{((a-1)\ mod\ N)} $$

Note that a multiframe process may be periodic or sporadic with a minimum inter-arrival time [60].

In simple terms, the worst-case execution time for successive activations of such a process varies, albeit with a periodicity of N, in terms of successive activations, as described. And while the absolute worst-case execution time for the process would be $max_{v=0}^{N-1} C^{(v)}$, simply using that scalar for the purposes of timing analysis and disregarding the actual variation pattern (i.e. using the analysis by Liu and Layland [54]) would be pessimistic. As Mok et al. demonstrate [60, 61], this approach may result in process sets which would be schedulable under a rate monotonic policy being erroneously deemed unschedulable. However, for systems wherein multiframe processes meet a property termed *accumulative monotonicity* they then introduce an exact schedulability test. We proceed with a definition of the above property and the formulation of that test.

According to Mok et al. [60, 61], an array of WCETs $\{C_i^{(0)},\ ..., C_i^{(N-1)}\}$ (and, by extension, the process τ_i associated with it) is *accumulatively monotonic* if some integer m $(0 \geq m \geq N-1)$ exists such that

$$\sum_{k=m}^{m+z} C_i^{(k \bmod N)} \geq \sum_{k=y}^{y+z} C_i^{(k \bmod N)},\ 1 \geq y \geq N-1,\ 1 \geq z \geq N-1,$$

It then follows that $C^{(m)}{=}max_{v=0}^{N-1} C^{(v)}$.

Mok et al. show that the worst-case scenario for some process τ_i (i.e. the scenario under which its response time is maximised) involves a maximal execution requirement by τ_i and a release coincident with the release of the respective m^{th} frame of each higher priority process (with execution requirements of interfering activations of higher-priority processes maximal subject to frame constraints).

Thus, if τ_i is schedulable under this scenario, it will always be schedulable.

It is noted in [60, 61] that in the trivial case that all processes are described by

single-element WCET arrays, the worst-case scenario provided by Mok et al. is reduced to the familiar critical instant scenario of Liu and Layland [54].

Regarding the issue of priority assignment, it is shown [60, 61] that the rate monotonic scheme is optimal for the scheduling of systems consisting of accumulatively monotonic multiframe processes (with deadlines equal to their minimum interarrival times).

Nevertheless, the above contributions by Mok et al. were somewhat limited in their scope by the fact that, in the general case, not all multiframe processes are accumulatively monotonic. Therefore, in [61] a transformation is introduced which makes it possible to apply the above analytical contributions irrespective of whether the property of accumulative monotonicity is met or not by multiframe processes in a system. We briefly outline this transformation:

In [61], Mok et al. discuss a process model (they term it the *general real-time* process model) where each process τ_j with a minimum interarrival time of T_j is described by an array $\{\phi_i^1,\ \phi_i^2,\ \phi_i^3,\ ...\}$ with the following semantics:

Any z consecutive activations of τ_i will have a cumulative execution requirement of at most ϕ^z time units.

This model is more general than the multiframe process model in that it accommodates processes with non repeating patterns of WCET variation (unlike the multiframe process model). Thus, it is possible to represent any multiframe process by an appropriate general process.

By definition [61], the *corresponding multiframe process* of a general process, with respect to an integer n is characterised by the same minimum interarrival time and the WCET array $\{\phi^1,\ \phi 2 - \phi 1,\ ...,\ \phi^n - \phi^{n-1}\}$. As Mok et al. then prove, a set of general processes is schedulable by some policy if the set of respective corresponding multiframe processes is schedulable by the same policy.

As shown by example in [61] it is possible for a multiframe process corresponding,

with respect to some integer, to a general process representing an accumulatively monotonic multiframe process to not be accumulatively monotonic - unlike the original process.

Nevertheless, as Mok et al. then proceed to prove, if any multiframe process (whether accumulatively monotonic or not) is represented as a general process, then the corresponding multiframe process of that general process with respect to the frame count of the original process will be accumulatively monotonic. This finding is important because for the purposes of schedulability analysis, the latter accumulatively monotonic process can be substituted for the former [61], thus making the contributions originally formulated in the context of accumulatively monotonic processes applicable to sets of multiframe process sets in general.

We comment that this transformation by Mok et al. reformulates the problem of establishing the schedulability system of a system to that of establishing the schedulability of a notional system derived by transformation of the processes of the original system. We will be revisiting the multiframe process model in Chapter 5 of this thesis (see page 124) to highlight some conceptual similarities (and key differences) with some of our (independently derived) contributions.

2.1.6 Dealing with self-suspending process

We noted earlier how one of the assumptions in the seminal paper of Liu and Layland [54] was that processes may not voluntarily suspend. This assumption is however rarely met in actual systems, as there is often the need for processes to voluntarily suspend. For example the scheduling of device drivers often employs this mechanism [22]. The need then arises for this behavior to be accounted for by worst-case response time analysis. We proceed to discuss the issue in the context of device drivers and how analysis copes with it.

Typically [22], I/O devices are accessed in the following manner:

- Initiate a reading.
- Wait for the value to be read to become available.
- Access the register where this value has been stored and resume program execution.

Three approaches to handling the delay between initiating a reading and finally accessing the data are possible:

- If the delay is small, the process could simply busy-wait on the flag signifying availability of the reading. While busy-waiting though, the process exerts interference on other processes.
- Alternatively, the process may reschedule itself to resume execution at some point in the future (by which time the associated value will be available) and suspend itself until then. We discuss the semantics and implications of this approach below.
- As a third option, for periodic processes, the reading initiated by some activation of a process is taken by the subsequent activation of the same process. Then, the process does not have to either busy-wait or suspend during the reading, thus schedulability is not impacted by the I/O operation. This technique is called *period displacement* [22] and is able to mask the delay, provided that $D \leq T - S$ (where D, T, respectively, signify the deadline and period of the process in consideration and S signifies the delay for the reading to become available). Note however, that not all applications can handle the staleness of the reading (upto $T + R \geq T + D$ in the worst-case, where R is a bound on the WCRT of the process) so this option is not always available.

As noted, the first option (i.e. busy waiting) is not appropriate when the associated delay is is long and the second option (i.e. rescheduling to some future time) is then

preferable, which spares lower-priority processes from the additional interference that busy-waiting could have caused. However this mechanism involves voluntary suspension, which is not permissible under the process model of Liu and Layland [54]. We proceed to describe how this restriction is removed:

For the purposes of deriving an upper bound on its worst-case response time, the self-suspending process is modelled as two "halves", separated by the delay interval. An upper bound on its worst-case response time is then calculated as $R = R_{before} + S + R_{after}$, where R_{before}, R_{after} are upper bounds for the respective worst-case response times of the two "halves" and S is the suspension delay [22]. This is quite pessimistic as it is extremely unlikely that both halves will both meet their worst-case response times within the same activation of the process. This would necessitate that each "half" execute for as long as its worst-case execution time and at the same time each suffer worst-case interference from higher-priority processes. However, in the general case this might not even be possible, hence the pessimism. For example, it might not be possible for the "halves" to be spaced is such a way that both could experience a critical instant. Or process code could make it an impossibility that both halves of a given activation would meet the respective WCETs (see Figure 3 for one such example that we crafted).

For the purposes of bounding the interference exerted by the self-suspending process upon lower-priority processes, each activation of the self-suspending process is taken to exert $C = C_{before} + C_{after}$ time units of interference (where C_{before}, C_{after} are upper bounds on the respective WCETs of the two "halves") [22]. Again, this may be pessimistic in the general case (as exemplified by our specially crafted example in Figure 3). Note that whether the two "halves" are modelled as a single process with a WCET of $C = C_{before} + C_{after}$ or as two separate processes is inconsequential for analysis. This is because, despite the fact that the two halves are offset in time, if modelled as distinct processes they would (pessimistically) be considered as being released at the same time (as the worst-case analysis assumes a critical instant).

```
int main() //entry point for the program
{if (conditionX==true)
  doA();
 else
  doB();
 initiate_reading();
 sleep_for(timeY); //the voluntary suspension
 take_reading();
 if (conditionX==true)
  doB();
 else
  doA();
}
```

Figure 3: In the general case, it might not be possible for both "halves" (before/after the self-suspension) of a process to meet their respective WCETs, as demonstrated by this crafted example. $ConditionX$ is determined upon process release and is not modified throughout the activation of the process and the WCETs of $doA()$ and $doB()$ differ.

One other important implication of introducing voluntary suspension is that worst-case blocking behavior changes. If in the absence of voluntary suspensions (for example, by enforcing a busy-waiting approach during device accesses, all other things remaining equal) the worst-case blocking term suffered by a process τ_i under some shared resource management scheme is B_i, then, in the presence of shared resources, this will rise to $(N+1)B_i$, where N is the number of voluntary suspensions by the process. This has been shown in [71] for the Priority Ceiling Protocol [69] but also holds for other protocols [22].

2.2 Dynamic priority scheduling

Such scheduling approaches are typified by the fact that the priority of a process activation is determined at run-time. The best known such approach is the *Earliest Deadline First* (EDF) scheduling scheme [54, 23].

Under EDF, the processor is granted to the process, among those competing for it on any given instant, whose deadline is nearest.

Dertouzos demonstrated [29] that if a process set is schedulable under some preemptive scheme other than EDF, then it will also be schedulable under EDF. In this sense, it is an *optimal* scheduling algorithm. Indeed, Liu and Layland [54] had earlier shown that any process set with processor utilisation not exceeding unity is schedulable under EDF.

A variation of EDF is the *Least Laxity First* scheme (LLC), introduced by Mok [59]. Laxity is defined therein as the difference between the estimated time until completion, for an activation of some process, and the time remaining until its deadline. Given its definition, laxity has to be recomputed each time that scheduling decisions are being made. For this reason (which translates to greater scheduling overheads in comparison to EDF) LLC has not seen wide deployment, although it is optimal in the same sense that EDF is.

When process deadlines may be less than their respective periods, the utilisation-based test for EDF is not sufficient to demonstrate feasibility (i.e. that all deadlines will be met even in the worst case). For this reason, schedule-based tests are typically employed, which involve plotting an actual schedule for an interval as long as the LCM of all process periods (much in the same manner as for asynchronous systems under fixed priority scheduling). Note however that there also exist equation-based approaches to the derivation of worst-case response times under EDF; these are presented in [78].

While EDF has its merits (namely higher attainable utilisation than what is achievable under fixed priority scheduling) and has never ceased to be a subject of research, we will be focusing mainly on fixed-priority scheduling. The reason is that FPS is more prevalent and better supported by current engineering practice and operating systems in general [22].

2.3 Conclusions

Having tracked the evolution of scheduling theory and associated timing analysis techniques for real-time systems, we note the progress that has been realised. The progress in associated timing analysis has, in each case, allowed for the underlying computational and scheduling model to be more widely adopted and, from the perspective of the designer, better understood (thus also more effectively targeted).

Uniprocessor real-time systems may now be constructed deadlock-free, with deadlines a priori guaranteed and without the inefficiency [16] and inflexibility [16] associated with cyclic executives. Meanwhile, distributed real-time systems, once not amenable to timing analysis at all, may now be analysed by use of the holistic approach (however pessimistic that, occasionally, may be).

However, there do exist computational models which the existing corpus of timing analysis literature does not address adequately. Among those is the limited parallel

model [10, 9], which is "halfway" between the uniprocessor and the fully distributed model. The associated architecture is the one prominently targeted by most of the codesign approaches we reviewed: a single general-purpose processor and multiple, closely coupled, application-specific hardware co-processors. Processes compete for the processor under a fixed-priority scheme but may also issue remote operations on some co-processor. For the duration of such operations, the process in consideration relinquishes the processor to the other processes competing for it. Thus, the execution of some process on the processor may be simultaneous with the execution of multiple other processes, each on some co-processor.

This model of computation is not particularly complex and is widespread in engineering practice. In the general case, a process invoking a co-processor operation will either have to busy-wait on the result or suspend itself [86]. Thus any system with co-processors (and at least two processes) where processes do not busy-wait on hardware exhibits limited parallelism. However, although such systems are analysable under both the uniprocessor and the holistic theory, the outcome is not satisfactory in either case. Not accounting for the properties of the computational model results in pessimism.

Under the uniprocessor theory, as originally formulated in [54] which required that *processes do not voluntarily suspend*, processes would be prevented from relinquishing the processor during a remote operation (which is analogous to *busy waiting*). Such a constraint destroys most of the potential for parallel execution of the processor and the co-processors.

It is possible to circumvent this restriction by employing the technique that we described when discussing the scheduling of device drivers. The process then reschedules itself to some instant in the future by which the co-processor operation while have completed. The analysis copes with this by treating the process as two "halves" (separated by the co-processor operation) for the purposes of calculating its overall worst-case response time. This is pessimistic, as discussed, as it assumes that both

"halves" encounter a critical instant and additionally that the execution requirement of both is maximal. Moreover, for the purposes of bounding the interference exerted by said process on lower-priority processes, a coincident release of both "halves" is assumed which is a third source of pessimism.

Exactly the same issues are encountered if the holistic approach is resorted to and a process is treated as a transaction of distinct subprocesses. Each one of those will be treated as having both a maximal execution requirement and suffering worst-case interference while they will be (pessimistically) treated as being released on the same instant when calculating interference on lower-priority processes.

Having pointed out these shortcomings of existing analysis, we believe that analysis appropriate to the limited parallel model is required. We will present our work in this area within the remainder of this thesis.

3 Overview of Limited Parallelism and Focus of This Thesis

This thesis clearly defines the *limited parallel computational model* and aims to deliver timing analysis which can adequately characterise it.

3.1 The limited parallel architecture and associated process model

We assume an architecture structured around a single general-purpose processor and multiple application-specific co-processors. All processing elements fully share address space and symmetrically access main memory. This architecture is exemplified by Figures 1(b) and 1(c) (see page 30).

The system functionality is structured as multiple processes. A process is generally implemented in software except (possibly) for select portions of its code which are implemented in hardware, on some specialised co-processor. By default, any such application-specific co-processor may only be accessed by a single, specific process. Thus, any process wishing to issue a hardware operation will not face competition for that hardware resource (i.e. the co-processor). However, as all processes require the single general-purpose processor for execution in software, access to it has to be managed by some mechanism. We assume that this mechanism is a fixed-priority scheduler: At any given instant, the processor is granted to the process, among those competing for it, with the highest priority.

This architecture is closely-coupled; since the memory is shared, the transition of process control from software to hardware (or vice versa) may be implemented via interrupt or a short message on the communication link. Thus, associated communication delays are deemed negligible. Additionally, since memory is shared, the notional migration of a process from software to hardware execution (and vice versa)

does not involve any copying of memory regions so as to communicate process state between processing elements (which would place considerable additional traffic on the bus). We also assume that scheduler overheads are negligible as well.

Any given process executing at a given instant may be executing either in hardware or in software. However, multiple processes may be executing at the same instant (at most one on the processor, the rest remotely each on some co-processor). Thus the architecture allows for parallelism, subject to the above limitation.

The class of systems is based on current engineering practice. As we already noted earlier, in the general case, two basic options exist with respect to co-processor operations: The initiating process will either busy-wait for the duration of the operation or it will suspend [86]. Any system with more than two competitively-scheduled processes which implements the second approach thus exhibits limited parallelism.

The use of co-processors *per se* is not novel. Floating point operations have traditionally been handled by specialised co-processors. However, in such circumstances, the main processor would typically pause while awaiting computation results from the co-processor – a form of busy waiting. Given that floating-point operations are typically short, this approach was not unreasonable. However, as noted earlier, the growing capacity of FPGAs [88, 3] has, for some years now, permitted the implementation of complex functions (such as audio/image/signal processing, communications, networking) in reconfigurable hardware. For such complex functions (which take considerably longer than typical floating point arithmetic to complete), pausing the processor becomes wasteful, in terms of processor utilisation. This reality motivates our contributions on the limited parallel model of computation.

3.1.1 Discussion of the assumptions in the above model

Regarding the model we described, objections could be raised towards some of the assumptions. More specifically, it could be argued that the assumption of no shared

co-processors is rather restrictive, or that contention between processing elements for the shared bus and memory is not properly accounted for. We proceed to discuss both of the above issues.

On the assumption of no shared co-processors

It could be argued that, by requiring that co-processors not be shared by multiple processes, our model is too restrictive. We counter that while, in the general case, there could be instances where the sharing of co-processors would be desirable, there are also many classes of systems where it is very unlikely that there would be such a need.

First of all, competitively-scheduled real time processes are likely to be quite distinct in what computation they perform. And while on a lower level two processes carrying out diverse functions might both employ some algorithm which could be implemented in specialised hardware (such a Fast Fourier Transform), functions to be implemented in specialised hardware are likely be higher-level, even more complex [86, 87] and thus specific to one given process. For example, the digital camera example discussed in [83] contains a hardware implementation of the JPEG codec in its entirety (rather than hardware for just the lower-level discrete cosine transform). Another such example would be that of a video accelerator [86].

Additionally, if application-specific functionality to be implemented upon a co-processor is extracted during codesign from software application code (as typified by [36] and unlike *a priori* hardware/software partitioning), it is statistically very improbable that the respective regions of code of any two processes would then match, for a single shared hardware implementation to be considered.

Even in those cases where part of the functionality of some process is mirrored in another process and, additionally, there is incentive for the implementation of this functionality in specialised hardware, an alternative option possibly preferable

(depending on the design in consideration) to using a single shared co-processor would be to replicate it for each process. This would help achieve better performance [86, 87].

Even if two processes in a system, with the same period T, are copies of one another and would need to use the same piece of hardware, one could consider scheduling them under a relative release offset of $T/2$ so as to avoid conflicts over the hardware. They could then be modelled as a single periodic process with a period of $T/2$.

Our point is that, even by disallowing the sharing of co-processors, our model hardly becomes irrelevant. By introducing the assumption that hardware is not shared, we choose to focus on those systems for which this assumption is reasonable – a sizeable share of embedded systems overall. Targeting a greater range of systems, while in principle desirable, is left for future work.

On potential contention between processing elements during bus I/O

In a uniprocessor system, the single CPU is the sole master of the system bus. Thus, while processes may be competing for the processor to execute on, there may be no contention for the bus because only one process may be executing (and thus able to issue reads/writes on the bus) at any given instant.

Under limited parallelism though, the possibility of two or more processes (e.g. one executing in software and one in hardware) contending for the bus arises. As a result, some memory reads/writes might be delayed. The impact on process WCET is difficult to quantify statically and if one assumes worst-case contention upon each and every bus access, considerable pessimism is typically introduced in the analysis [22]. It is thus desirable to design systems so as to prevent such behavior. We suggest a few alternative approaches in that direction:

- One approach is to prevent the co-processors from doing any bus I/O during operation (apart from the short message to the CPU upon completion). This

necessitates that any data that serves as input to hardware-based computation be copied by the CPU over the bus to the address corresponding to the internal buffers of the co-processor before the hardware operation is initiated. During operation, the co-processor stores intermediate computation results internally (or upon some dedicated memory buffer hanging separately off the co-processor; the semantics are the same). Upon completion, it sends a short message over the bus to signal termination, which causes the corresponding process to switch from *waiting* to *ready* state. When the process next gets to execute in software, it fetches back the results via reads from the address corresponding to the associated co-processor buffer.

Under this approach, all bus I/O (apart from messages signalling co-processor termination) is CPU-initiated. However, not all applications lend themselves well to this approach. For example, in the case of streaming applications, where huge amounts of data would have to be copied to/read from the co-processors before/after each hardware operation as input/output, respectively. This approach is thus better left for less bandwidth-intensive applications.

- Another option would be to move away from the assumption of a uniform memory architecture. In the case of bandwidth-intensive applications, the data on which co-processors operate need not be stored in main memory. Instead, a dedicated memory (as in a frame buffer, for graphics applications) could be used. As non-uniform memory architectures (NUMA) often feature multiple buses for performance considerations [87], (notably for streaming applications [43]), the dedicated memory module for co-processor data could be accessed over a separate bus to which the co-processor would also be connected. Traffic in one bus would then not interfere with traffic on the other bus – hence no contention. However, despite the additional bus(es), the process model would not differ from the one we described on the context of a single bus. Figure 4 depicts an instance of this architectural variant:

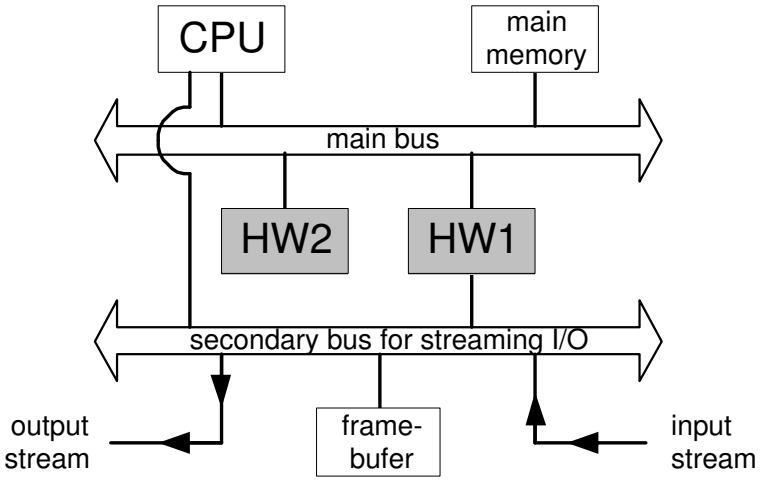

Figure 4: A variant limited parallel architecture, with a dedicated bus for streaming I/O.

> The co-processor $HW1$ does some processing on a stream. All stream-related I/O takes place over the dedicated bus. $HW2$ is another co-processor not related to streaming, hence hangs only from the main bus and has its input written to it over the main bus via CPU-initiated I/O prior to each time that it is invoked. $HW1$ is the sole master on the secondary bus. If a link exists between the secondary bus and the CPU (as in our example), then the control word signalling termination by co-processor $HW1$ may be sent on the secondary bus (so as to not interfere with the main bus).

Nevertheless, we acknowledge that in some cases, there may arise the need for a co-processor to do *occasional* I/O on the main bus, to access main memory. In that case, some contention for that bus will be introduced. Even with just the short messages signalling co-processor termination being sent over the main bus, contention is possible and has to be accounted for by the analysis.

Rajkumar et al. in [70] deal with the issue of contention during memory I/O by considering multiple memory banks and separate of code/data banks. Their approach

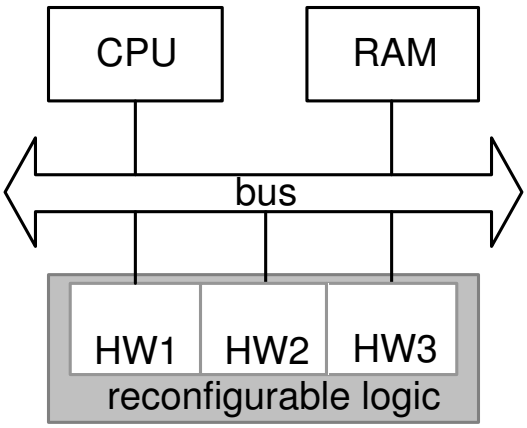

Figure 5: The architecture on which the example system of Figure 6 is based (graphic adapted from [10, 9, 18]).

is much less pessimistic than assuming the worst case for each and every memory access. Although it was formulated in the context of a traditional process model, we see no reason why it could not be made to work with the limited parallel model. Still, this exceeds the scope of this thesis and has to be addressed in future work.

3.1.2 Overview of process management in a limited parallel system

To better highlight the semantics of parallel process execution in the kind of systems we described, we will rely on the following example:

A limited parallel system is formed by three mixed hardware/software processes (τ_1, τ_2, τ_3, by order of increasing priority). The architecture (depicted in Figure 5 consists of the single general-purpose processor (CPU) and three hardware co-processors (HW1, HW2, HW3).

We will proceed to simulate the scheduling on the processor for a sequence of scheduling events. For convenience, we also provide two equivalent visualisations of the above:

Figure 6 describes the scheduling as activity intervals for each processing unit; num-

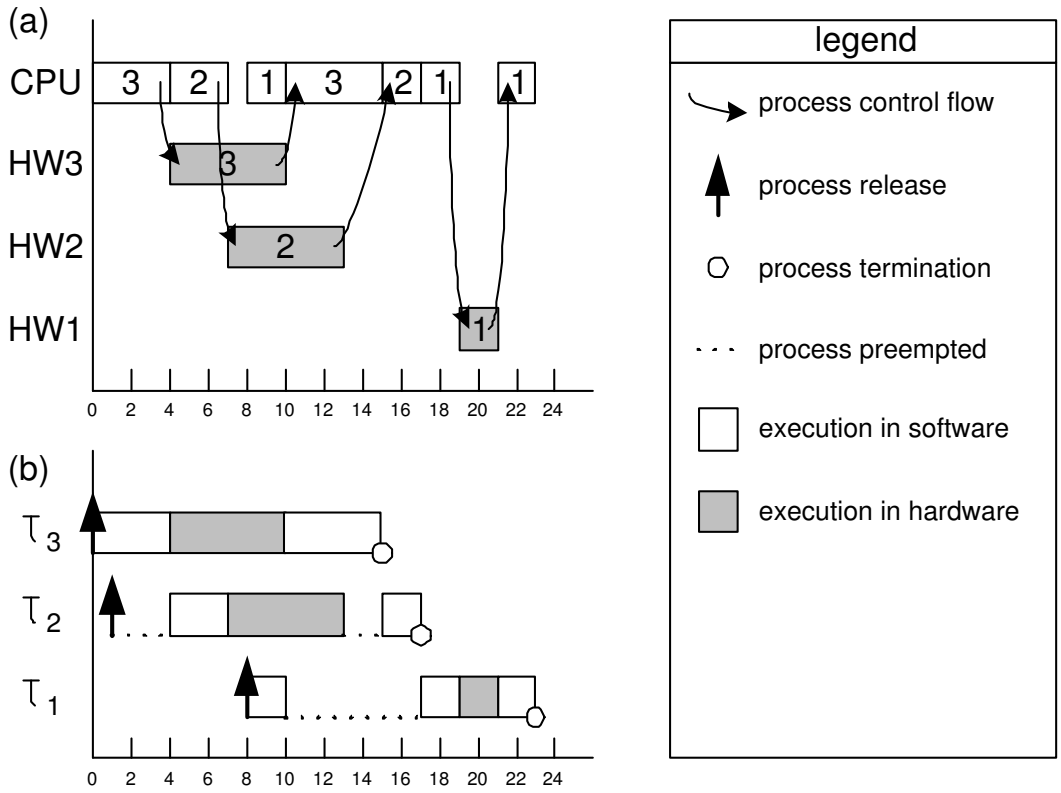

Figure 6: A limited parallel system in action

bers within the rectangles in this Gannt chart denote the process using the processing element during the corresponding interval. Figure 6 equivalently describes the scheduling activity by plotting the status of each process (executing in software; executing in hardware; not executing) as a function of time. Onwards with the simulation:

At $t = 0$, τ_3 is released and immediately requests the processor (so as to execute in software). At $t = 1$, while τ_3 is executing on the CPU, τ_2 is released and requires the processor as well; however, it is preempted because τ has priority. At $t = 4$, τ_3 switches to execution in hardware (on HW3), which frees up the processor for τ_2 to immediately use. At $t = 7$, τ_2 switches to execution in hardware as well (on HW2), which leaves the processor idle, as no process requires it. Note that from $t = 4$ to $t = 7$ we have execution in parallel between hardware and software (the CPU and HW3 respectively) whereas from $t = 7$ to $t = 8$ we have parallel execution in hardware (HW2 and HW3).

At $t = 8$, τ_1, is released and requests the CPU. Despite being the lowest-priority process among those executing at the time, it is immediately granted the processor because the other processes are executing on hardware and do not need it. However, at $t = 10$, upon completion of the operation on $HW3$, τ_3 switches back to software execution and preempts τ_1 (which still requires the CPU). However, as τ_2 is still executing on $HW2$, it is unaffected. τ_2 only gets to be prevented from further advancing in computation at $t = 13$, when it completes the hardware operation and requests the CPU once again (but is preempted due to τ_3 being of higher priority). In terms of parallelism, we observe that between $t = 8$ and $t = 10$, three processing units operate in parallel (CPU, HW3, HW2) and between $t = 10$ and $t = 13$, two processing units (CPU and HW2).

τ_3 terminates at $t = 15$; then τ_2 is granted the processor, in turn terminating at $t = 17$. Subsequently τ_1 executes without any competition.

For illustration purposes only, we also provide (in Figure 7) the visualisation of

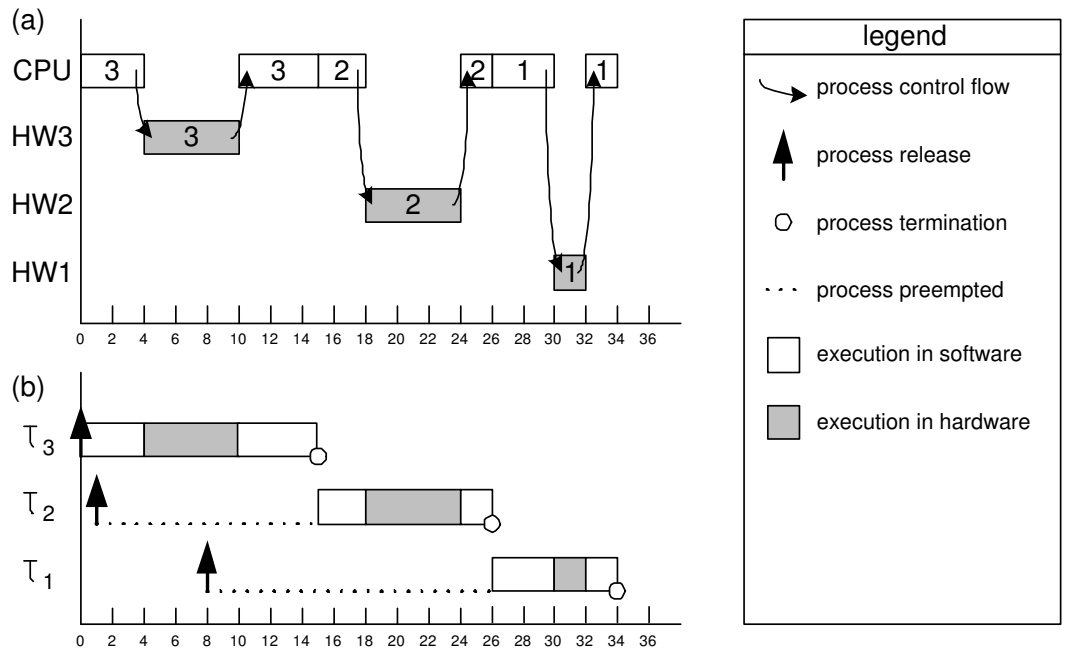

Figure 7: Scheduling decisions for the same system as that of Figure 6 if the processor is idled during hardware operations.

scheduling decisions for a non-parallel version of the same system (i.e. under a policy of idling the processor for the duration of a hardware operation).

Just by inspection, it is obvious how much more underutilised the processor is in this case. Moreover, the response times of processes other than the one with the highest priority increase: (25 time units vs 16 for τ_2 and 26 units vs 15 for τ_1).

While we believe that this example is a good illustration of the merits of limited parallelism in the general case, what is of interest in the context of this thesis is the improvement *in the worst case*. We thus proceed to highlight the pessimism in the timing analysis of limited parallel systems, under established (i.e conventional) timing analysis techniques.

3.2 Issues with established timing analysis when applied to limited parallel systems

Established worst-case timing analysis techniques may indeed be applied to limited parallel systems. In terms of analysis originally formulated for uniprocessor systems, we note the work in [11] and in terms of analysis for distributed systems we note the holistic approach [79, 80]. However, each of those techniques when applied to a limited parallel system is considerably more pessimistic than when applied to its original domain (i.e uniprocessor/distributed systems, respectively).

We need only hint at the problems with uniprocessor analysis:

In order for such analysis to be applicable, any parallelism is disregarded and the system is treated as if the processor were idled during hardware operation. This is safe (i.e. upper bounds on process response times computed are valid) but pessimistic. If some process has a worst-case execution requirement of C time units but at most X units may be as execution in software, then each activation of this process will be treated as exerting up to C time units of interference on any lower-priority process (whereas the actual upper bound is X). However, naively substituting X for C in the worst-case response time equations would not work; in fact it could possibly render the analysis *optimistic* (thus invalid). We demonstrate this by an example:

In a two-process system, let for the higher-priority process τ_2: $X_2 = 5$, $C_2 = 10$. The lower-priority process is entirely software-based and characterised by $X_1 = C_1 = 7$. The period of τ_2 is $T_2 = 15$.

If the uniprocessor analysis as per Liu and Layland (see Equation 5) is applied, an upper bound for the for the response time of τ_1 is calculated as

$$R_1 = C_1 + \left\lceil \frac{R_1}{T_2} \right\rceil C_2$$

Via variable substitution, we obtain $R_1 = 27$.

If however (naively, as we will show) X_2 is naively substituted for C_2 in the above equation, the output is 12. However, this does not take into account the variability, from activation to activation of τ_1 in the time that hardware operations are invoked. This variability is, in the general case, due to variations in control flow. In the worst-case, such operations could be invoked anywhere from as early as possible to as late as possible, within the activation.

Now follow an actually observable case, for this trivial system (plotted as a schedule in Figure 8). τ_2 immediately upon release, switches to hardware execution, then back to software execution at $t = 0$ for as long as possible ($X_2 = 5$ time units). However, at the same instant as this transition, τ_1 is released (thus stays preempted until $t = 5$). At $t = 10$, τ_1 still has 1 time unit of computation left but the next release of τ_2 occurs. This time, τ_2 proceeds with execution in software for as long as possible before any hardware execution – hence τ_1 suffers another 5 time units of preemption before next getting to execute (and complete).

The response time for τ_1 is in this case $R_1 = 17$ – less than 27 (the output of uniprocessor analysis) but more than 12 (the outcome of the "naive" approach previously described).

The reader is challenged to construct some other plausible schedule under which the response time of τ_1 exceeds 17 (spoiler: we prove later within this thesis, that this is impossible). Even more interestingly, under coincident releases of τ_1, τ_2, the response time of τ_1 is at most 12 (same as the "naive" approach). Hence, the worst-case scenario is something other than a critical instant – unlike what holds for uniprocessor systems.

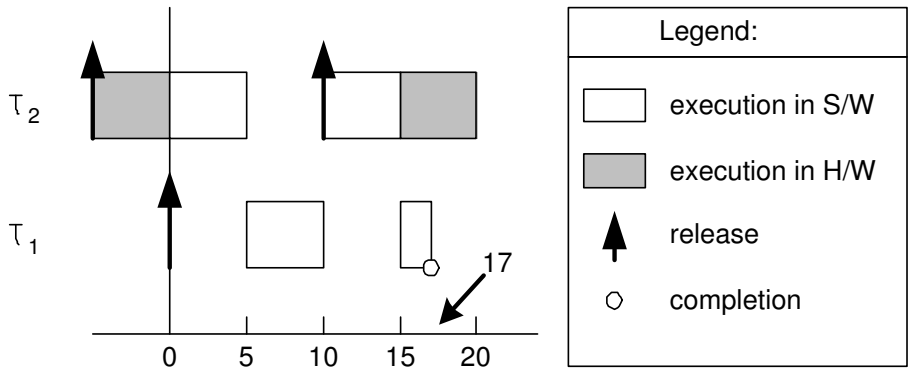

Figure 8: The worst-case for the above limited parallel system is not observed under coincident process releases.

3.3 Summary

We briefly discussed the place of limited parallel systems in current engineering practice and examined them from an architecture/hardware point of view. We next proceeded with a more detailed discussion of the semantics of this computation model and its underlying assumptions (some of which will later be relaxed). Subsequently, some motivation for the work towards the development of appropriate timing analysis was provided.

We may thus proceed with the formulation of our actual contributions.

4 Basic Worst-Case Response Time Analysis For Limited Parallel Systems

Within this chapter the semantics of hardware and software execution in limited parallel systems are examined in detail and appropriate worst-case timing analysis for this computational model is introduced. While this analysis is basic, it forms the core of our contributions; it was originally published in [10, 9]. All of our additional contributions (such as the more exact worst-case response time analysis presented in Chapter 5 and the best-case response time analysis presented in Chapter 6) are based on this work.

4.1 Process model

A process set $\Delta = \{\tau_1,\ \tau_2,\ ...,\ \tau_n\}$ is scheduled by a fixed priority scheme: the highest-priority process, among those competing for the processor on any given instant, is the one executing on it. Priorities are unique, ranging from 1 (lowest) to n (highest). Each process has a period T_i and a deadline $D_i \leq T_i$. For a sporadic process, T_i denotes the minimum interarrival time between successive releases. The worst-case response time (abbreviated as WCRT) of τ_i is R_i.

Whilst executing in software, processes may not voluntarily suspend (other than to access hardware resources - see below). An initial assumption (convenient in terms of structuring the presentation of our contribution, but later removed for software resources) is that processes share no resources (hence do not block).

Up to this point in our description, these are characteristics that may as well be shared with pure uniprocessor architectures. However the *Limited Parallel Model*, which we define here as part of our contributions (abbreviated as LPM), differs from those in that it allows parts of process functionality to execute in dedicated co-processors, while at the same time making the processor available to other processes

for the duration of such operations and modelling the effects. From the perspective of software execution, a process undergoes voluntary suspension when initiating computation on a co-processor. However, how long it will then remain suspended is unknown at the time of suspension and corresponds to the duration of the hardware operation (which is variable within known bounds). This is in unlike the usual technique for scheduling device drivers [22], discussed in our literature review, where the length of the suspension is fixed.

During a remote (i.e. off-processor) operation, the process in consideration advances in computation albeit without requiring use of the processor. As this is not reflected in the traditional state model of a process, we introduce a new state termed *waiting*. The extended state model is shown in Figure 9 with states (labelled by initial: Ready, Executing, Blocked, Idle, Waiting) and allowed transitions. The processor is always assigned to the highest-priority process among those ready, with that process switching to executing state. When control is passed to a hardware co-processor, the respective process transitions from executing to waiting state. It switches back to ready state when the co-processor function terminates. Note that the scheduler may need to reconsider which process to allow to execute whenever a co-processor function terminates, as the process switching to ready state may be of higher priority than the process executing up to that moment.

One could argue that perhaps we need not have introduced a separate state (i.e. *waiting*) given that the semantics are essentially the same as when a process is blocked for I/O. However, we note that whereas I/O latencies are a side-effect of the architecture, hardware operations are part of the actual application so, notionally at least, there is a difference. We also believe that for an efficient scheduler implementation it would make sense for a separate queue of processes executing in hardware (i.e. in the waiting state) to be maintained.

Code blocks mapped onto the processor are termed *local blocks* (abbreviated as *LBs*) The intuition behind the name is that their execution is local to the processor. Con-

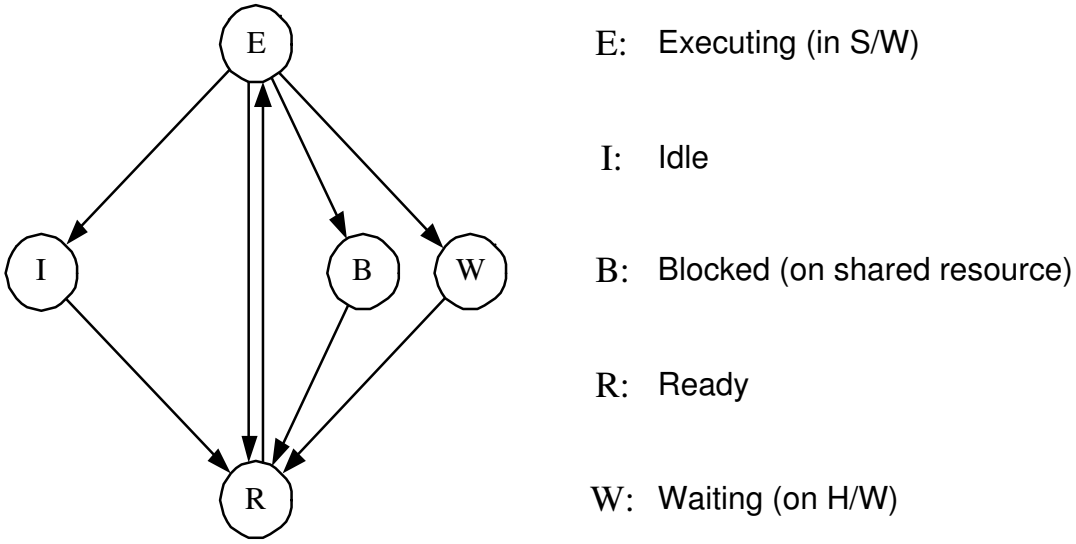

Figure 9: Extended Process model [9, 10]

versely code blocks implemented as functions on a co-processor are called *remote blocks* (abbreviated as *RBs*). The time interval spent in waiting state by a process which has issued a remote operation is termed a *gap*, as it forms an interruption in the conventional processor-based execution of the process. However, when confusion does not arise, we may use the same term to refer to the actual remote block associated with this interval. A process activation may invoke a number of hardware co-processor functions from its release to its termination, hence may contain a number of gaps. The overall time spent by an activation of process advancing in computation (as opposed to being preempted or blocked) equals the sum of its execution time in software and its execution time on the co-processor(s). Note that a remote operation initiated by τ_i at time t will (given that co-processors are not shared between processes) terminate at time $t + g$, where g is the length of the gap and is not dependent on any other process. At any time, upto n processes may be in the waiting state, having issued remote operations.

4.2 Modelling of process structure

At which point, within the lifecycle of a process, gaps occur depends on code structure (which is fixed) and on control flow (which is variable for each activation of the process). Knowing where gaps lie may help exploit potential parallelism, thus reducing pessimism in the analysis. This realisation has motivated the development of the following model for process structure:

A process is modelled as a directed acyclic graph. A single node with only outgoing edges, termed the *source*, corresponds to the point-of-entry for the process. Similarly, a node with incoming edges only (the *sink*) corresponds to the point-of-exit (i.e. the termination). All other nodes model code blocks with single-entry/single-exit semantics. They are annotated by best- and worst-case execution times. An example of our model is given in Figure 10.

This model is a more restrictive version of the one in [67]. The main difference between the two is that in [67] multiple edges outgoing from the same node may "fire" simultaneously. This is because the respective specification allows for the expression of partial precedence constraints regarding operation execution and, by choice, leaves the exact ordering of operations to the implementation. Any edge in that model is activated if its guard condition variable (computed at runtime) evaluates to *true*; if the respective guards of more than one edges outgoing for the same node happen to evaluate to *true*, all of those will fire.

On the other hand, what we instead are interested in modelling is, essentially, control flow. The control flow for each activation of some process can be expressed as a path from *source* to *sink*. This path may vary for subsequent activations of the same process but, in any case, in our model, the conditions (evaluated at run time) for which the outgoing edges of a given node are activated are mutually exclusive. This mutual exclusiveness may also be expressed in the model of Pop et al. but has to be made explicit via mutually exclusive guards (for example, Q and $NOT\ Q$),

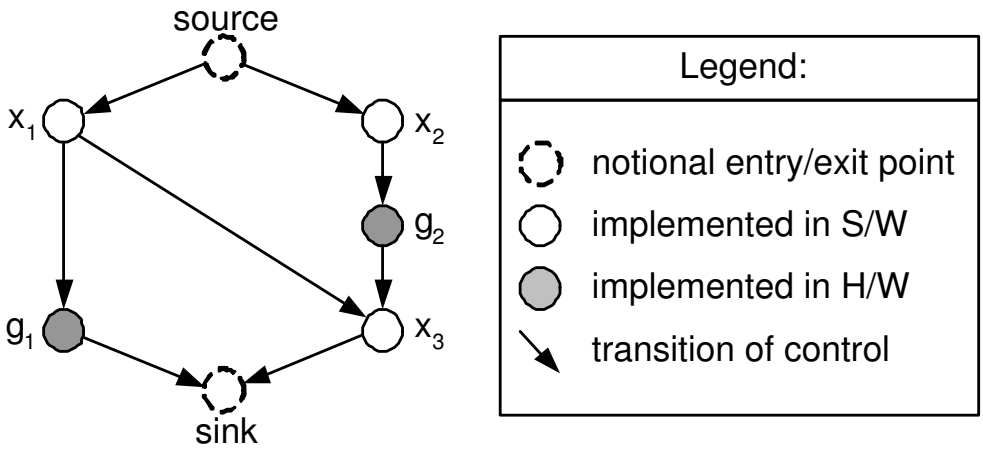

Figure 10: A Model of Process Structure

Thus, an instance of our model may always be ported to the model in [67] without loss of information or the insertion of additional restrictions; the inverse is not always possible. Figure 11(a) is an example of such a lossless translation. In Figure 11(b) however, the graph on the left (an instance of the model of Pop et al. [67]) may only be described in terms of our model if an additional precedence constraint between c and d is enforced – either i) c must execute before d or ii) vice versa. The model of Pop et al. would also permit iii) an implementation where c and d execute in parallel (on different processing elements), however this is not compatible with the limited parallel model (which only permits different processes to execute in parallel, not code blocks belonging to the same process).

The actual sequence of code blocks activated from release to termination and whether they are local or remote defines the length and relative position of gaps and software execution within a specific activation of a process. Figure 10 depicts an example instance of our model. Nodes corresponding to blocks mapped in hardware appear shaded. Node weights express execution times for the respective code blocks. Table 1 displays overall process execution time, execution time in software and execution time in hardware as functions of x_1, x_2, x_3, g_1, g_2 (variables themselves, each

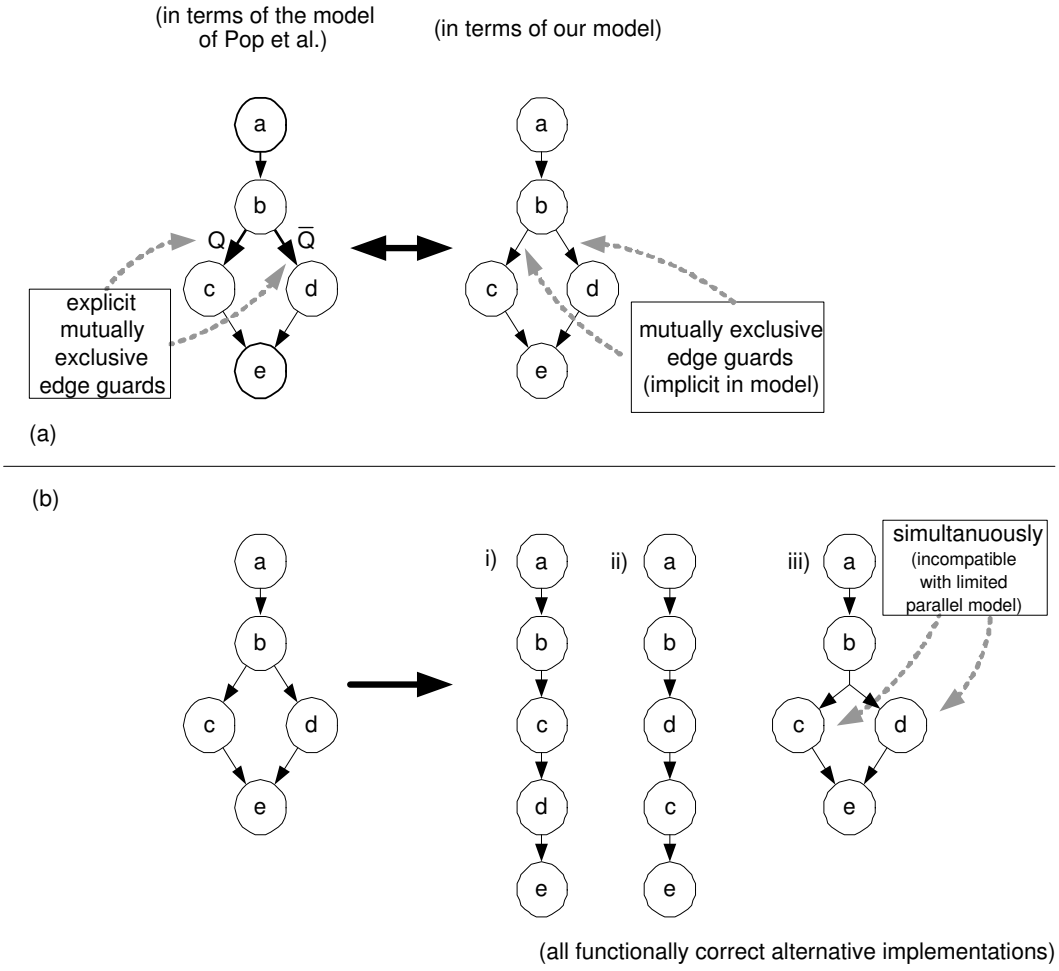

Figure 11: Conversion between our model and that of Pop et al. [67]

	Execution time		
path	overall	in S/W	in H/W
path 1	$x_1 + g_1$	x_1	g_1
path 2	$x_1 + x_3$	$x_1 + x_3$	0
path 3	$x_2 + g_2 + x_3$	$x_2 + x_3$	g_2

Table 1: Execution time (overall, in S/W and in H/W) as a function of code block latencies for all different control flows of the process depicted in Figure 10.

within a respective known range).

By simple substitution of variables with values in the above example, it can be shown that it is possible for the overall WCET, the WCET in software and the WCET in hardware, to be, all three of them, observable for distinct control flows. For example, by assuming the upper bounds for x_1, x_2, x_3, g_1, g_2 to be, respectively, 7, 6, 7, 10, 5, the overall WCET for the process (represented by $\mathbf{C}$ throughout this thesis) is observable for path 3 (18 time units), the worst-case execution time in software (respectively, $\mathbf{X}$) is observable for path 2 (14 time units) and the worst-case execution time in hardware (respectively, $\mathbf{G}$) is observable for path 1 (10 time units).

In any case, for any process, $C \leq X + G$. If the set of control flows for which the worst-case execution time in software (X) may be observed is disjoint from the set of control flows for which the worst-case execution time in hardware (G) is observable, then $C < X + G$; else $C = X + G$.

A similar property can be shown for the respective **best-case** execution times (abbreviated as BCETs), represented by the $\hat{C}$, $\hat{X}$ and $\hat{G}$ in this thesis for overall, software and hardware execution times respectively: $\hat{C} \geq \hat{X} + \hat{G}$.

4.3 Observations

Conventional, entirely processor-based, systems can be thought of as a subset of the class of systems conforming to the above defined process model. Simply, they are systems where processes never issue remote operations, or, in more formal representation, for which the following holds:

$\forall \tau_i \in \Delta : G_i = 0$

Upper bounds on process response times for such systems (assuming no knowledge of relative process release offsets, which, in principle, might reduce the pessimism) are given (in the case of perfectly periodic processes) by Equation 5:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5}$$

where $hp(i)$ is the set of processes with priority higher than that of τ_i. Derived upper bounds on process worst-case response times (abbreviated as WCRTs throughout this thesis) are valid under the previously stated assumptions (i.e. no shared resources, hence no blocking; no voluntary process suspensions).

Now consider the case when some of the processes do issue remote operations. One approach is to pause, not only the initiating process, during a remote operation, but also the processor itself, until the operation completes. This is typically the approach for floating-point operations (i.e. short hardware operations). Bounds derived by Equation 5 then remain valid. However, the higher the ratio of execution time in hardware to overall execution time is, the more and more inefficient this approach becomes (as the processor would be idle for much of the time).

Nevertheless, in the absence of shared resources, there is no downside (given that context switching overheads are negligible as per our stated assumptions) to not making the processor available, for the duration of a gap, to other processes. Bounds derived by Equation 5 would then still be valid (since process response times cannot

increase as a result) but (as already noted in Section 2 of Chapter 3 and also via the example of Figure 8) increasingly pessimistic, the higher the reliance on hardware.

We proceed to identify the worst-case scenario for interference under this new scheduling paradigm (which is **unlike** the *critical instant* defined in [54] for purely uniprocessor systems) and then construct equations which derive associated bounds on process WCRTs. We proceed from simpler (and thus more pessimistic) to more complex (but less pessimistic) analytical approaches and, in the process, add support for shared software resources. We note that our analysis, in its entirety, is **offset agnostic** (i.e. it outputs valid upper bounds for all possible combinations of relative process release offsets). This characteristic permits the accommodation of sporadic processes as well.

4.4 Simple model

This analytical approach detailed within this section permits the derivation of upper bounds on process WCRTs even when knowledge of gap placement is incomplete, provided that for each process, bounds on C and X are known (via WCET analysis, which is outside the scope of this thesis). We then simply (and pessimistically) assume that, within an activation whose execution time is at most C time units, any of the intervals of length equal to 1 time unit into which it may be divided may either be spent executing in software or hardware - but at most X of them in software, overall. The conditions which then maximise interference exerted by a given process τ_j on another (lower-priority) process τ_i are:

- An activation of τ_j released at $t = -(C_j - X_j)$ (assuming that τ_i is released at $t = 0$) and executing in hardware for $t = C_j - X_j$ time units before switching to software execution for (the remaining) X_j time units.

- Subsequent activations of τ_j released at $kT_j - C_j + X_j$ • $k = 1,\ 2, ..$ and

executing for (a full) X_j time units in software before switching to execution in hardware (for upto $C_j - X_j$ units, if at all).

For multiple higher-priority processes, the conditions which maximise cumulative interference on τ_i (hence also its response time) are the above, for each higher priority process τ_j. This has been proven in [10] and in [9] and upper bounds on process WCRTs according to this worst-case scenario are given by Equation 6:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + C_j - X_j}{T_j} \right\rceil X_j \tag{6}$$

(In the above equation, note how a process with an overall WCET of C_j and a WCET in software of X_j exerts the same interference, under our worst-case scenario on a lower-priority process as would an entirely software-based process with the same period, scheduled in its place, with a WCET of X_j and a release jitter of $C_j - X_j$)

We proceed to formulate the above as a sequence of theorems:

Theorem 1 *For a given process τ_i (released at $t = 0$ without loss of generality), fully implemented in software, the interference exerted on it jointly by all higher-priority processes is maximised (assuming that any activation of each $\tau_j \in hp(i)$ has a respective worst-case execution time of C_j overall and X_j in software and also that the initiation time and duration of hardware operations may arbitrary, subject to the previous constraints), if the following hold:*

- *The first interfering activation of each τ_j is released at $t = -(C_j - X_j)$ and immediately executes in hardware for $C_j - X_j$ time units before switching to software execution for an additional X_j time units.*
- *Any subsequent interfering activations of τ_j are released at $kT_j - C_j + X_j \bullet k = 1, 2, ..$ and immediately execute for X_j time units in software before switching to execution in hardware (for upto $C_j - X_j$ units, if at all).*

Proof: There are two possibilities for each $\tau_j \in hp(i)$: either the first interfering activation of τ_j is released prior to the release of τ_i (thus τ_i is released at a point where that activation of τ_j still has outstanding computation) or it is released at some instant $t = t_j^{(0)}$ for which the following holds: $0 \leq t_j^{(0)} < T_j$.

- If the latter is true, then the releases of τ_j past $t = 0$ occur at:

$$t_j^{(k)} = t_j^0 + kT_j,\ k = 0,\ 1,\ 2, ...$$

  If each of those activations places as much demand as possible (i.e. X_j), as early as possible (i.e. immediately upon release), then interference on τ_i by each interfering activation of τ_j individually is maximised, which also maximises the number of interfering activations of τ_j; thus the interference exerted by the sequence of activations of τ_j as a whole, is maximised.

  With that as a given, moving $t_j^{(0)}$ to the left along the time axis may not decrease the interference exerted by the sequence (and may in fact increase it). The minimum value for $t_j^{(0)}$ is 0, which then maximises the interference exerted by the sequence of activations of τ_j upon τ_i.

- If, however, the first interfering activation of τ_j is released at some $\tau_j^{(-1)}$ (for which: $-T_j < \tau_j^{(-1)} < 0$), then subsequent releases of τ_j will occur at

$$t_j^{(k)} = (\tau_j^{(-1)} + T_j) + kT_j,\ k = 0,\ 1,\ 2, ...$$

  We already established that the interference exerted by the sequence of subsequent activations of τ_j is maximised if they request the processor for X_j time units immediately upon release. With that as a given, we must select a value for $\tau_j^{(-1)}$ and a corresponding placement of hardware and software execution within the first interfering activation of τ_j for which the overall interference

exerted on τ_i by τ_j (i.e by both the first and subsequent activations if τ_j) is maximised.

The interference exerted by the first interfering activation of τ_j on τ_i is maximised in the general case if this activation requests the processor for a long as possible (i.e. X_j) just as τ_i is released (i.e. at $t = 0$). For this to occur, τ_j would have to be released at some $t = \tau_j^{(-1)} = -G_j^{(\alpha)}$, execute in hardware for exactly $G_j^{(\alpha)}$ time units (i.e. until $t = 0$), then request the processor for X_j time units and, upon completion of the execution in software, execute in hardware for an optional $G_j^{(\omega)}$ time units, subject to the constraint: $G_j^{(\alpha)} + G_j^{(\omega)} \leq G_j$.

Of all possible combinations of $G_j^{(\alpha)}$ and $G_j^{(\omega)}$, the one under which interference from subsequent activations of τ_j is maximised is: $G_j^{(\alpha)} = C_j - X_j \Rightarrow G_j^{(\omega)} = 0$. The reason is that it moves as far to the left, along the time axis, as possible the releases of subsequent activations of τ_j, subject to the constraint that the interference exerted by the first interfering activation of τ_j be maximal (i.e. X_j). Under this scenario, releases of τ_j would occur at

$$t = -(C_j - X_j),\ T_j - (C_j - X_j),\ 2T_j - (C_j - X_j),\ 3T_j - (C_j - X_j), ...$$

and corresponding requests by τ_j for the processor (all of them for X_j time units) would occur at

$$t = 0,\ T_j - (C_j - X_j),\ 2T_j - (C_j - X_j),\ 3T_j - (C_j - X_j), ...$$

We have thus proven that this is the scenario that maximises overall interference from τ_j on τ_i if the first interfering activation of τ_j is released within the interval $[-(C_j - X_j),\ -1]$. However, upon inspection, the overall interference exerted by τ_j under this scenario, additionally, may be no less than that observable if the first interfering activation of τ_j is released at $t = 0$ or later (the

first request for the processor will occur at $t = 0$ in either case; however each subsequent one occurs $(C_j - X_j)$ time units earlier, which may not decrease interference).

Thus, we have identified the scenario which maximises interference from each τ_j independently, if the first interfering activation of that process is released at $t = -G_j$ or later. Additionally, it is possible for this scenario to hold for all processes. If so, it is the scenario which maximises interference jointly from all $\tau_j \in hp(i)$ upon τ_i, subject to the constraint that the first interfering activation of each $\tau_j \in hp(i)$ must, respectively, be released at $t = -(C_j - X_j)$ or later.

However, what if, for some of the processes in hp(i), the first interfering activation is released even earlier than $t = -(C_j - X_j)$?

Moving the release of the first interfering activation of each τ_j further to the left by some respective Δt_j reduces the overall demand on the processor (i.e. due to all $\tau_j \in hp(i)$ jointly) by at least the same number of time units that each subsequent activation of τ_j is moved to the left. Hence, overall interference on τ_i exerted by hp(i) jointly cannot increase as a result.

□

Since, under our worst-case scenario, requests by each τ_j for the processor are, respectively, for C_j time units and occur at $t = 0,\ T_j-(C_j-X_j),\ 2T_j-(C_j-X_j),\ ...$, the fixed-point equation which gives an upper bound for the worst-case response time of τ_j is constructed as:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + C_j - X_j}{T_j} \right\rceil X_j$$

which is the exact same formulation as Equation 6, provided earlier. We earlier claimed, though, that Equation 6 outputs a valid upper bound on the worst-case

response time of any process (i.e. whether it contains gaps or not). Theorem 1, however, explicitly relied on the assumption that τ_i may not contain gaps. This apparent discrepancy is settled by the following corollary:

Corollary 1 *Equation 6 outputs a valid upper bound for the worst-case response time of τ_i whether it contains gaps or not.*

Proof:

When τ_i is executing in hardware, this execution is not preempted whenever some higher-priority process on the processor (in our terminology, execution in hardware is *immune to interference*). Suppose though that, via some mechanism, the execution of τ_i in hardware would freeze during the execution of some higher-priority process on the processor (and only resume thereafter). The response time of τ_i may only increase as a result. However, the scheduling decisions would then be identical to those that would be observed if, instead of execution in hardware, τ_i was executing on the processor, for the respective intervals.

Suppose then that the execution of higher-priority processes conforms to our (earlier identified) worst-case scenario. If the execution of τ_i in hardware were not immune to interference, an upper bound for the corresponding worst-case response time of τ_i is given by Equation 6. However, that bound then is also an upper bound for the worst-case response time of τ_i under the actual scheduling semantics.
□

An interpretation of this worst-case scenario:

The first interfering instance of each higher-priority process τ_j has software execution occur as late as possible, for as long as possible (i.e. for X_j time units after $C_j - X_j$ time units of hardware execution have been realised) and it is that transition from

hardware (i.e. remote) to software (i.e. local) execution, not the release of τ_j, that which occurs concurrently with the release of τ_i. Subsequent activations of each higher priority process are, however, characterised by the inverse pattern: software execution as early as possible for as long as possible (i.e. X_j).

One similarity to the *critical instant* (i.e. the worst-case scenario for uniprocessor analysis [54], which dictates coincident process releases) is that all higher-priority processes first request the processor on the instant that τ_i is released (i.e $t = 0$). However, subsequent requests for the processor arrive with (what effectively is) a jitter of $T_j - X_j$, that is, at instants $t = T_j - X_j,\ 2T_j - X_j,\ 3T_j - X_j,\ ....$ This reflects our (pessimistic) previous assumption that gaps could "float" anywhere within the activation of a process. Consider this metaphor:

An abacus represents the activation of a process. If the rod of this notional abacus is long enough to fit exactly C_j beads (each of which represents one time unit of execution in software) but the actual number of beads is only $X_j < C_j$, each individual bead can only travel at most a distance of $C_j - X_j$ times its thickness, along the axis (see Figure 12).

Note that when $C_j = X_j$ (i.e. in the case of systems which do not issue any remote operations or, if they do, idle the processor for their duration), this worst-case scenario is reduced to the familiar *critical instant*, which is the worst-case scenario for uniprocessor analysis [54].

This worst-case scenario is visualised in Figure 13.

4.4.1 Evaluation

That our analysis outperforms the established uniprocessor analysis without being computationally more complex is, by itself, significant. However, for proper evaluation, a quantitative assessment of our approach is required.

The most meaningful relevant metric would have been the ratio of overestimation

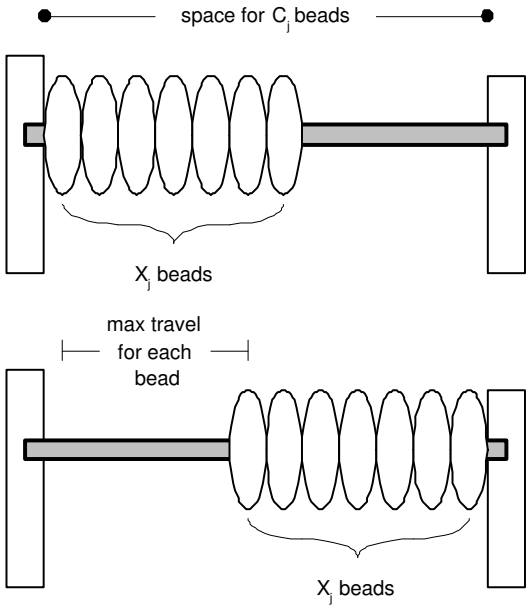

Figure 12: An abacus, as a metaphor for the activation of a process: beads stand for time units of execution in software, whereas stretches of the rod not surrounded by a bead stand for execution in hardware. The above example is drawn to scale for $C_j = 12$, $X_j = 7$.

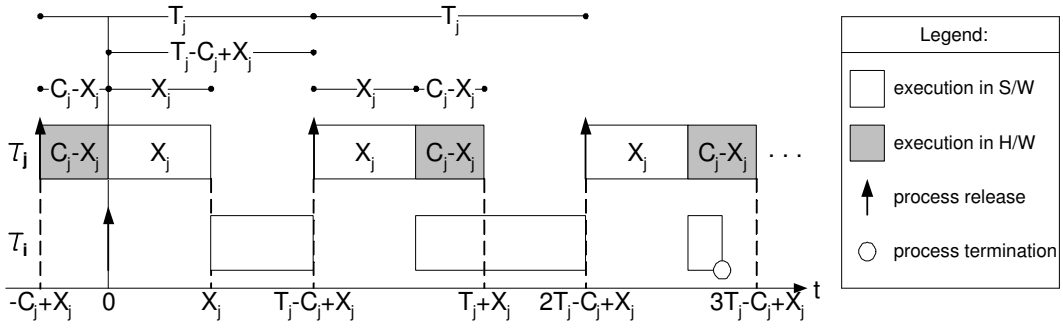

Figure 13: The worst-case scenario for the basic analysis pertaining to the limited parallel model (i.e. assuming unlimited gap mobility)

(on average, over a large set of representative limited parallel systems analysed) of process response times under our approach to the corresponding overestimation under established uniprocessor analysis. However, such an evaluation is not possible because, exact worst-case response times are generally unknown (and intractable to derive via exhaustive simulation of all alternative cases, for all but trivial systems).

Thus the next best way to evaluate our contribution would be a comparison of upper bounds on WCRTs derived under our approach to those derived under conventional uniprocessor analysis. Unfortunately we do not have access to numbers for actual real-world limited parallel systems. Thus we proceed to compile some (made up) example process sets for indicative purposes only.

1^{st} example

The aspects of our first example system are summarised by Table 2. Regarding the choice of process parameters (periods and worst-case execution times in software and overall), we ensured that $\frac{X}{C} \geq 0.75$ for each process. The closer that this ratio of the WCET in software to the overall WCET is to unity, the less a process relies on hardware for computation and the more the system resembles a traditional uniprocessor system (where $\frac{X}{C} = 1$). We assume that, for a typical process with hardware operations, $0.5 \leq \frac{X}{C} \leq 0.8$ so this caters for the typical case without skewing the example towards showing our approach in good light. We also ensured that there is a "comfortable" margin between (even) the upper bounds on WCRTs derived by the uniprocessor approach and the respective process periods (again, so as not to skew the example to favor our approach).

The three rightmost columns in Table 2 show the upper bound on the WCRT of the given process derived by our approach (as ${}^{o}R$), the respective upper bound derived by the uniprocessor analysis (as ${}^{c}R$) and the relative reduction achieved by our approach (as a percentage).

process	priority	T	C	X	${}^{o}R$	${}^{c}R$	$\frac{{}^{o}R-{}^{c}R}{{}^{c}R} * 100\%$
τ_5	5	50	20	15	20	20	-0%
τ_4	4	70	25	20	40	45	-9%
τ_3	3	300	55	45	160	245	-35%
τ_2	2	1000	40	30	240	890	-73%
τ_1	1	4000	40	35	415	2940	-86%

Table 2: An example of a limited parallel system, analysed by both our approach and the uniprocessor analysis.

process	priority	T	C	X	${}^{o}R$	${}^{c}R$	$\frac{{}^{o}R-{}^{c}R}{{}^{c}R} * 100\%$
τ_5	5	50	20	18	20	20	-0%
τ_4	4	70	25	23	43	45	-4%
τ_3	3	300	55	50	196	245	-20%
τ_2	2	1000	40	35	481	890	-46%
τ_1	1	4000	40	38	748	2940	-75%

Table 3: A variant of the system of Table 2, less reliant on hardware.

The results show that the reduction in the derived bounds for WCRT achieved by our approach is more significant the lower the priority of a process is.

2^{nd} example

We modify the above example so that with C staying the same, $(C-X)$ is effectively halved for each process (thus $\frac{X}{C} \geq 0.875$) to explore how our approach fares if reliance on hardware is limited (see Table 3).

The results once again show noticeable decrease in estimated WCRTs for lower-priority processes. Thus, even in the case where there is little reliance on hardware (and the system thus resembles a uniprocessor system) the small reductions in the

estimated interference from each activation of an interfering process add up to a significant improvement in the estimation of a worst-case response time for the lower-priority processes.

The magnitude of improvement, even for conservative inputs, thus leaves us fairly confident that our approach permits considerable accuracy (especially in comparison to uniprocessor analysis).

4.5 Provision for shared resources

Not allowing for shared resources would mean that our model would be of little practical value in the real world. On the other hand, access to shared resources must then be managed by some scheme for the purpose of deadlock avoidance. One additional concern for real-time systems is that the time spent blocked by any process activation must remain bounded in any case (or else the schedulability of the system is compromised). Hence, we will proceed to introduce herein shared resource management mechanisms which meet all of those requirements. As part of our contribution, we lay out equations which derive the aforementioned upper bounds on per-process blocking.

The Priority Ceiling Protocol was formulated by Rajkumar in [69] for uniprocessor systems. It achieves deadlock avoidance and bounded blocking. Two slightly different versions of it exist: The Original Ceiling Priority Protocol (OCPP) and the Immediate Ceiling Priority Protocol (ICPP) [22]. They require proper nesting of critical sections [77] (which, by extension, we require as well). We provide simple descriptions of both:

OCPP:

1. Each process has a static default (and, in our case, unique) priority.

2. Each shared resource has a static ceiling value, defined as the maximum static priority of the processes using it.

3. Each process has, on any given instant, a dynamic priority, derived as the maximum of its own static priority and the static priority of any process blocked by it.

4. A process may only enter a critical section guarding a shared resource if its dynamic priority is higher than the ceiling of **any** shared resource in the system in use on the given instant (excluding any such resource already in use by the process in consideration). Upon failing to enter the critical section guarding a resource, a process blocks.

ICPP:

1. Each process has a static default (and, in our case, unique) priority.

2. Each shared resource has a static ceiling value, defined as the maximum static priority of the processes using it.

3. Each process has, on any given instant, a dynamic priority, derived as the maximum of its own static priority and the ceiling values of any resources locked by it.

4. A process may only enter a critical section guarding a shared a resource if its dynamic priority is higher than the ceiling of **any** shared resource in the system in use on the given instant.

It is obvious that OCPP and ICPP only differ in the 3^{rd} rule. For a uniprocessor system implementing either the OCPP or the ICPP, worst-case per-process blocking is bounded by:

$$B_i = \max_{u=0}^{U} \; usage(u, i)b(u) \tag{7}$$

where critical sections guarding shared resources are numbered 1 to U, $b(u)$ is the worst-case blocking term associated with the respective critical section and the *usage* function is defined as

$$usage(u,i) = \begin{cases} 1 & \text{if the } u^{th} \text{ critical section belongs} \\ & \text{to a process not in hp(i) and} \\ & \text{the resource guarded by the} \\ & u^{th} \text{ critical section is used by at} \\ & \text{least one process with a priority} \\ & \text{less than that of } \tau_i \text{ and at least} \\ & \text{one with priority greater than or} \\ & \text{equal to that of } \tau_i \text{ (incl. } \tau_i) \\ 0 & \text{otherwise} \end{cases}$$

Since ICPP is easier to implement and more widely supported than OCPP (for example, in POSIX, as *Priority Protect Protocol* and in Real-Time Java as *Priority Ceiling Emulation* [22]), we will limit our attention to the accommodation of ICPP with the Limited Parallel Model.

We limit the scope of our discussion to software resources. The hardware co-processors themselves may not be shared. Moreover, when executing in hardware, processes may not be holding access shared resources.

In the past it has been erroneously claimed by us in [10] (and by reference, in [18]) that, when the Priority Ceiling Protocol is applied to limited parallel systems, worst-case per-process blocking is bounded by the terms derived by use of Equation 7. In other words, that the same upper bounds on per-process blocking hold for limited parallel systems as for pure uniprocessor systems. We disprove those past claims of

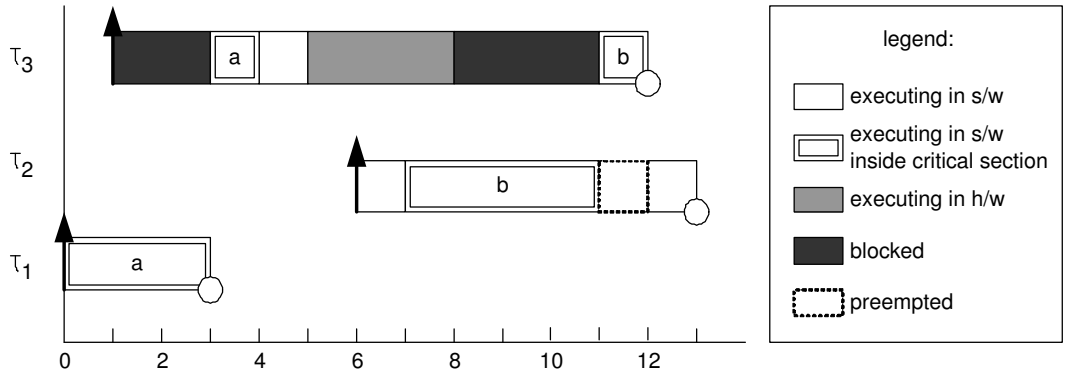

Figure 14: It is possible, in limited parallel system, for a process to block on more than one critical section, despite the PCP being employed.

ours via the example of Figure 14, before proceeding to derive the correct bounds:

In the example of Figure 14, lowest-priority process τ_1 is released at $t = 0$ without contention for the processor and immediately proceeds to access shared resource a, only releasing it at the very end of its execution. At $t = 1$, highest-priority process τ_3 is released and immediately attempts to lock the same resource but blocks (due to it being already in use by τ_1) and stays blocked until τ_1 releases the resource at $t = 3$. Later, at $t = 5$, τ_3 issues a remote operation on a co-processor. During that operation, at $t = 6$, medium-priority process τ_2 is released and is immediately granted the processor (as there is no contention for it, since τ_3 is awaiting the completion of its remote operation). τ_2 attempts to enter the critical section guarding shared resource b at $t = 7$ and succeeds. This is correct behavior, according to the ICPP. However, when the remote operation of τ_3 completes at $t = 8$, τ_3 is blocked upon reentry to the processor, as τ_2 is still using resource b and only releases it at $t = 11$.

This example demonstrates that, in a limited parallel system, despite use of the PCP, a single process activation may block on more than one critical section, unlike its behavior in uniprocessor systems. Thus, Equation 7 does not output per-process

valid bounds for limited parallel systems. However, we are still able to bound per-process blocking under the ICPP for such systems, by reasoning as follows:

We observe that, each time that a process issues a remote operation, it incurs the risk of being blocked upon the completion of that remote operation and reentry to the processor. This is because some lower-priority process, which has had the opportunity to execute on the processor during the remote operation may have entered a critical section guarding some resource whose ceiling is higher than the priority of the original process. This behavior is a result of making available the processor to other processes for the duration of remote operations. Idling the processor instead, would have prevented the multiple instances of blocking but would then have removed altogether any parallelism.

However, whether under ICPP or OCPP, we observe that blocking is still limited to at most one critical section per initiation of execution in software (whether this is upon release of the process or upon completion of a previously issued remote operation). Thus, Equation 7 correctly outputs upper bounds on blocking albeit on a per-LB (i.e. per contiguous single-entry/single-exit piece of software code) basis, not on a per-process basis. Upper bounds on per-process blocking may then be derived by multiplying that blocking term with (the upper bound on the) number of times that the process in consideration initiates software execution within its activation. The corresponding equation then is

$$B_i = n(\tau_i) \max_{u=0}^{U} \; usage(u, i) b(u) \tag{8}$$

where $n(\tau_i)$ is an upper bound on the number of times that τ_i may initiate software execution and is derived by path analysis for the process graph model of τ_i. Intuitively, in terms of the process graph, assume that there are V possible control flows (i.e. paths from *source* to *sink*) for τ_i and that for the for the v^{th} of those, the number of edges pointing from either the source or a non-local node to a local

node is $n(\tau_i, v)$. Then,

$$n(\tau_i) = \max_{v=1}^{V} \; n(\tau_i, v) \tag{9}$$

Thus $n(\tau_i)$ is a static property for the process.

For purely uniprocessor systems, where the entire process activation is one contiguous piece of software code (uninterrupted by remote operations), $n(\tau_i) = 1, \; \forall i$. Thus, purely uniprocessor systems can be once again viewed as a special case (i.e. a subset) of limited parallel systems. Note how Equation 8 reduces to Equation 7 for $n(\tau_i) = 1$.

If OCPP was used in our limited parallel system instead of ICPP, the same behavior would be observed and the same upper bounds on blocking terms per piece of software code and per process would hold as under ICPP. The only difference would be that the original process would not necessarily be blocked directly upon reentry, unless suffering push-through blocking. If not suffering push-through blocking, it would block upon actually trying to access the shared resource already in use by the lower-priority process.

Equation 9 is consistent with the findings of Rajkumar et al. in [71] where they state as a corollary (in the context of uniprocessor systems with resources managed under the PCP) that:

"A job J that suspends itself n times during its execution can be blocked for the duration of at most n+1 critical sections of lower priority jobs."

This issue was earlier discussed, within our literature survey (see page 61) in the context of the scheduling of device drivers. As issuing a hardware operation constitutes a voluntary suspension from the perspective of execution on the processor, the same issue applies to the limited parallel model.

Thus, for limited parallel systems with shared resources managed by the PCP, the

equation according to which upper bounds on process worst-case response times are derived is:

$$R_i = C_i + B_i \sum_{j \in hp(i)} \left\lceil \frac{R_i + C_j - X_j}{T_j} \right\rceil X_j \tag{10}$$

with B_i given by Equation 8.

We do not view as a weak point of the limited parallel model the fact that, following any remote operation, blocking may be experienced each time by the process upon reentry to the processor. It is, rather, a tradeoff left to the system designer. For example, if the aim of the designer is to reduce the WCRT of a process by moving a portion of its code to hardware, this would only make sense if, in the worst case, the speedup is not offset by the additional potential block. Both of these effects are quantifiable. On the other hand, if, despite such a potential (and quantifiable) increase in WCRT, the process in consideration would still meet its deadline, it might make sense to have the operation be in hardware, as this would result in decreased interference on lower-priority processes, thus decreased respective WCRTs.

For select cases, the designer could still get the speedup from the hardware without having the process potentially suffer an additional block by having the process busy-wait (i.e. not relinquish the CPU to any lower-priority processes while awaiting the completion of the hardware operation). However, the remote operation would then no more constitute a gap (i.e. an opportunity for limited parallelism). This remote operation would then instead effectively be exerting interference on lower-priority processes. Thus it would be, for the purposes of our analysis (and excluding the reduced execution time associated with hardware), treated as execution in software.

4.5.1 Evaluation

We will proceed to analyse an example limited parallel system with shared resources and compare the derived bounds on process WCRTs to those derived by the unipro-

resource	shared by	worst-case critical section length
Q_1	$\tau_1,\ \tau_3,\ \tau_5$	3
Q_2	$\tau_3,\ \tau_4$	4

Table 4: Shared resources within the system of Table 5

process	priority	T	C	X	$n(\tau)$	${}^oB$	${}^cB$	${}^oR$	${}^cR$	$\frac{{}^oR-{}^cR}{{}^cR} * 100\%$
τ_5	5	50	20	15	2	6	3	26	23	+13%
τ_4	4	70	25	20	2	8	4	63	49	+29%
τ_3	3	300	55	45	2	6	3	181	278	−35%
τ_2	2	1000	40	30	2	6	3	261	893	−71%
τ_1	1	4000	40	35	2	0	0	415	2940	−86%

Table 5: The system of Table 2, amended to include the shared resources of Table 4.

cessor analysis. For this purpose we shall adapt one of our earlier examples (the system of Table 2) via the addition of some shared resources. For convenience, we will be assuming that the critical sections belonging to different processes but guarding the same resource are of the same worst-case length.

Table 4 lists the shared resources that we introduce in the system, which is in turn described by Table 5. We have assumed that, for each process, $n(\tau_i) = 2$. ${}^cB$ is the worst-case blocking term per process under the uniprocessor analysis, whereas ${}^oB$, respectively, under the analysis for the limited parallel model.

For this example, it is not just two analytical approaches that are juxtaposed. If the processor is not idled during the execution of hardware, the blocking terms given in column ${}^oB$ would have to be used, even if the uniprocessor analysis is employed. Thus, if the uniprocessor analysis is used, enforcing the idling of the processor during hardware operations (paradoxically) would result in lower derived upper bounds for process WCRTs (compared to the case that this idling is not enforced but the

uniprocessor analysis is used anyway), because the worst-case blocking terms would be lower (i.e. those of column ${}^{c}B$). So as then to be fair to that approach, when using the uniprocessor analysis, we will be assuming uniprocessor semantics (i.e processor idling during hardware operations).

A comparison on upper bounds on process WCRTs derived under the two approaches in consideration, for the above example, shows that for higher-priority processes a numerically modest (but potentially significant as a percentage) increase in the response times of higher-priority processes may be observed. This is attributable to the higher worst-case blocking terms per process under limited parallelism.

For lower-priority processes, however, we once again note a considerable decrease in the derived upper bounds for their WCRTs under our approach. This is because the response times of these processes are dominated by interference (the overestimation of which is drastically reduced using our analysis).

Since it is typically at lower priority levels that ensuring schedulability is a challenge, we believe that the considerable reduction in pessimism for the analysis of lower-priority processes should weigh more that the potential modest increase in the response times of higher-priority processes.

4.6 Remaining issues with the simple model

We identify two main remaining sources of pessimism with the simple analysis we provided above.

The first source of pessimism is the overestimation of the freedom of movement of gaps within a process activation. Consider for example the process structure depicted in Figure 15(a). The previous worst-case scenario would require the gap to occur at the start of the execution of the process for the purposes of maximising its interference on other processes (and at a specific offset relative to the release of such lower-priority processes). Yet, from inspection of the structure of the process, it can

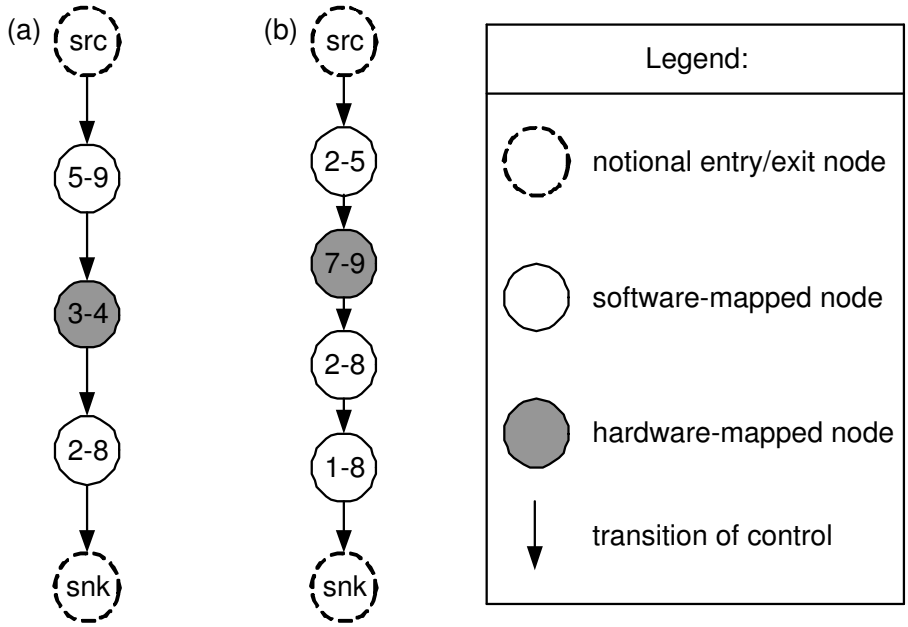

Figure 15: Examples of simple process graphs

be seen that the gap can only occur after the process has completed at least 5 time units of execution. Moreover, that same scenario would have all execution from each interfering process occur in one big, contiguous lump (equal in length to the WCET in software for that process). Yet, again by inspection, we notice that, no more than 9 time units of execution by the process of Figure 15(a) can be contiguous (i.e. not separated by gaps or by intervals of idleness by the process, between successive releases of it). We shall be referring, within this text, to this identified source of pessimism as *Weakness #1*.

The second source of pessimism (which we shall be referring to as *Weakness #2*, respectively, throughout this text) stems from the fact that the way according to which upper bounds on process WCRTs are calculated by Equation 6 is equivalent to assuming that, at any instant when some process is executing on the processor, any activations of lower-priority processes which exist at that instant (i.e. which have been released prior to that instant but not yet terminated) are, by necessity,

suffering interference. Yet, this is pessimistic, because if such a lower-priority process is already executing in hardware by the time that the higher-priority process gets to execute on the processor, it is then spared (some of the) interference. Execution in hardware both does not exert and, at the same time, is immune to interference.

In the following chapters, we will proceed to remove to some extent, where possible, these two shortcomings of the simple analysis presented so far and will do so in an offset-agnostic manner (i.e deriving bounds which are valid for all possible combinations of process release offsets, without actually considering or enumerating any such combinations).

5 Accurate Worst-Case Response Time Analysis: The Synthetic Approach

Within this chapter, we introduce a more exact analytical approach, termed the *Synthetic analysis*. As previously, we will initially formulate our contributions under the assumption that there are no shared resources; subsequently, we will remove that assumption and discuss any repercussions.

5.1 Intuition behind the Synthetic Analysis

For processes which are linear in nature (i.e for which the graph model of their structure is linear, as in the case of the process depicted in Figure 15(a)), it is possible to derive less pessimistic upper bounds for the interference exerted by them on lower-priority processes, as we will proceed to show. For the remainder of this chapter (and for the purposes of formulating our contribution) we assume that all processes fit this criterion. We subsequently remove this restriction and show how the analysis can be applied to systems consisting of both linear and non-linear processes.

What we refer to as the *execution pattern* of a process is an interleaved sequence of blocks of local and remote execution e.g. *xgxgx* or *gxgx* (where x stands for local and g for remote execution). When considering such a pattern, adjacent blocks in a linear process mapped both hardware or software are considered as one (i.e. the process of Figure 15(b) would have an execution pattern of *xgx*, not *xgxx*. Moreover, it is obvious that subsequent activations of linear processes will always conform to the same pattern. This is not necessarily true for non-linear processes.

Consider a process with local-only execution. It is trivial to show that in the worst-case (in terms of interference suffered by it), the release of the process in consideration would have to be coincident with the start of LBs from all higher-priority

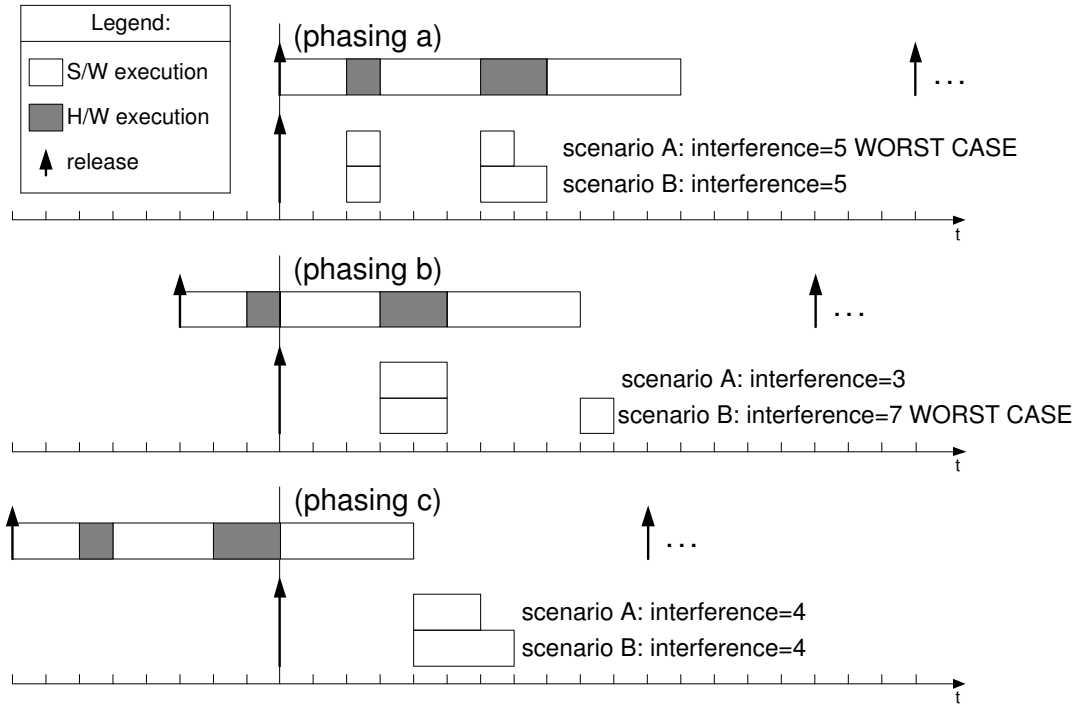

Figure 16: Examining interference under different phasings for each scenario

processes. If the number of LBs in a process is given by $n(\tau)$, the question then is which one of the $n(\tau_j)$ alternative local code blocks (or, equivalently, respective relative release phasings) this is, for each interfering process τ_j. Consider an example: In a two-process system, the high-priority process has an execution distribution pattern of $xgxgx$ with (fixed) respective code block lengths of 2, 1, 3, 2, 4 and a period of 19 (yielding 7 time units between the termination of any given activation of the process and the release of its next activation). To identify the worst-case phasing in terms of interference suffered by the low-priority process we examine all 3 candidate phasings for each of two scenarios: Scenario A involves the lower-priority process having an execution requirement of 2. Under Scenario B, that execution requirement is 3 (see Figure 16).

The example shows that, in the general case, which phasing yields the worst case depends on the execution requirement of the process suffering the interference. More-

over, for multiple higher-priority processes, phasings from each higher-priority process τ_j would need to be considered in combination, which might not be tractable (as the number of possible combinations is $\Pi_{j \in hp(i)} n(\tau_j)$). And this does not even take into account the variability, in the general case (and unlike what holds for the example process set of Figure 16), of code block execution times which would also have to be taken into account, creating a state explosion. This realisation directs us towards an analytical approach which does not involve offset consideration.

5.1.1 Notation

Consider the following notation: The execution pattern of a process τ_j is represented as: $x_{j_1} g_{j_1} x_{j_2} g_{j_2} ... x_{j_{n(\tau_j)}} g_{j_{n(\tau_j)}}$. X_{j_k}, $\hat{X}_{j_k}$ denote the maximum/minimum length of local block x_{j_k}. G_{j_k}, $\hat{G}_{j_k}$ denote the maximum/minimum length of gap g_{j_k}. With specific values assigned to an execution pattern we obtain what we term an *execution distribution* for the respective process activation - a plot of the time intervals of local and remote execution.

The concept of an *execution distribution* (not to be confused with a statistical distribution) is fundamental to the contributions of this thesis. It describes the placement and respective execution times of local and remote blocks within an activation of a process. For example, for a specific activation of a process to follow the distribution $[4,\ (23)\ , 8]$ would mean that the process would be released locally and would switch to remote execution for 23 time units after having accumulated an execution time of 4 locally. Upon completion of the remote operation it would then switch back to local execution for another 8 time units. We will be using parentheses to enclose latencies of remote operations, whenever it is not clear from the context whether the execution is remote or local.

Distinct activations of the same process are characterised by different execution distributions in the general case. The execution time $|\mathbb{K}|$ of a given distribution $\mathbb{K}$

is equal to the sum of the lengths of its constituent blocks. Obviously $\hat{C}_i \leq |\mathbb{K}| \leq C_i$ (where $\hat{C}_i$ denotes the best-case execution time for the given process) for any distribution $\mathbb{K}$ which belongs to the set of execution distributions possible for that process τ_i.

To denote whole families of execution distributions, ranges may be used for execution latencies. This enables the representation of all possible execution distributions for a linear process. An example of a ranged distribution is $[4-7,\ (21-28)\ , 8-9]$ and represents the set of distributions of the form $[x,\ (y),\ z]$ where $4 \leq x \leq 7$, $21 \leq y \leq 28$ and $8 \leq z \leq 9$.

5.2 The algorithm

The synthetic analysis bounds the interference exerted by a given linear process on other (lower-priority) processes by calculating an upper bound for what the interference would have been if instances of the interfering process were characterised by a specific execution distribution (called the *synthetic worst-case execution distribution* for the respective interfering process) and released under a certain jitter. This upper bound (whose calculation is tractable) is in turn an upper bound for the actual interference (whose calculation is intractable). The synthetic distribution is essentially a construct for the purposes of analysis (hence the name) and might not actually be observable in the general case. An important property of the analysis is that derived upper bounds on process WCRTs are valid for all possible relative release offsets between processes, hence the analysis is termed to be *offset-agnostic*. As such, it may accommodate sporadic processes. We proceed to describe the algorithm according to which a synthetic distribution is constructed (and which was first introduced in [18]).

In simple terms, the synthetic distribution is constructed by requiring that local blocks execute for as long possible and that remote blocks execute for as short as

possible and then rearranging their positions inside the activation of the interfering process so that local and remote blocks still appear interleaved, albeit in order of decreasing/increasing execution time respectively. In more formal terms (and considering some border cases which are not covered by the above simplified procedure, which was provided for illustration purposes only), the algorithm (originally formulated in [18]) is as follows:

For some process τ_j, its synthetic execution distribution is obtained as follows:

Algorithm:

- Consider the execution distribution resulting by having LBs execute for as long as their respective WCETs and RBs execute for as long as their respective BCETs.

- Shift-rotate the resulting execution distribution to the left, if necessary, so that it starts with a local block (an operation which we call *distribution normalisation*).

- Attach to the execution distribution a notional remote block of length (i.e. execution duration) $N_j = T_j - R_j$. This is termed the *notional gap*; intuitively it represents the minimum interval between successive activations of τ_j (i.e. from the termination of one activation until the release of the next one).

- Merge any resulting adjacent remote blocks into one. After this, there should be an equal number of local blocks and gaps in the resulting execution distribution, $n(\tau_j)$.

- Let ${}^{\xi}\mathbf{X_j}$ be a set of integers, of cardinality $n(\tau j)$, with the execution latencies of the local blocks from the execution distribution previously generated as its members. Similarly, let ${}^{\xi}\mathbf{G_j}$ be the set consisting of the gap lengths from the previously generated distribution.

- Let ${}^{\xi}x_j(1)$, ${}^{\xi}x_j(2)$, ... , ${}^{\xi}x_j(n(\tau_j))$ be the sequence returning the members of ${}^{\xi}\mathbf{X_j}$ in ascending order.

  Let ${}^{\xi}g_j(1)$, ${}^{\xi}g_j(2)$, ... , ${}^{\xi}g_j(n(\tau_j))$ similarly be the sequence returning the members of ${}^{\xi}\mathbf{G_j}$ in descending order.

- The synthetic execution distribution for τ_j is then constructed as:

$$[{}^{\xi}x_j(1),\ ({}^{\xi}g_j(1)),\ {}^{\xi}x_j(2),\ ({}^{\xi}g_j(2)),\ ...,\ {}^{\xi}x_j(n(\tau_j)),\ ({}^{\xi}g_j(n(\tau_j)))]$$

□

We proceed to reiterate, somewhat more rigorously, a theorem originally published in [18], on which our analysis is based:

Theorem 2 *For a process τ_i, fully implemented in software, suffering interference from a linear higher-priority process τ_j, the interference suffered by τ_i due to activations of τ_j released not earlier than τ_i cannot exceed the interference that would result if these activations of τ_j were characterised by its synthetic distribution (as previously defined).*

Proof: Let τ_i be released at $t = t_1$. Then:

- Consider the execution distributions of all activations of τ_j released at $t \geq t_1$. If each is normalised (as described above, by shift rotation to the left) to start with local execution and the notional gap is appended to it, interference to τ_i from activations of τ_j released at $t \geq t_1$ cannot decrease.

- Shifting, then to the left (i.e. earlier) along the time axis (if necessary) the sequence of releases of τ_i until the release of τ_i coincides with a release of a LB of τ_j does not decrease the interference suffered by τ_i.

- Then, if the execution distributions of all activations of τ_j released at $t \geq t_1$ are modified so that all LBs acquire their respective maximum length, interference cannot decrease.
- Likewise, if all gaps acquire their respective minimum lengths.
- Then, if within all execution distributions of activations of τ_j released at $t \geq t_1$, the first local block (from left to right) swaps its length (if not already the longest one) with the longest local block in the execution distribution, interference cannot decrease. Similarly, then, with the second local block from the left with the longest remaining local block and so on, until we ran out of local blocks. At the end of this transformation, local blocks appear in order of decreasing length from left to right (intuitively, causing longer preemptions to occur earlier in time).
- Similarly then for gaps, as previously with local blocks, but with the first gap from the left exchanging length with the shortest one, the second one from the left with the second shortest one and so on until they appear in order of increasing length, from left to right. Interference cannot decrease as a result.

We thus obtain, after all those transformations, an infinite sequence of activations of τ_j, released at $t = t_1,\ T_j + t_1\ T_j + t_2,\ ...$ and characterised by the distribution

$$[{}^{\xi}x_j(1),\ ({}^{\xi}g_j(1)),\ {}^{\xi}x_j(2),\ ({}^{\xi}g_j(2)),\ ...,\ {}^{\xi}x_j(n(\tau_j)),\ ({}^{\xi}g_j(n(\tau_j)))]$$

(which we know as the synthetic worst-case execution distribution).

Suppose then that, under different distributions for (some of) the activations of τ_j released at $t \geq t_1$ and/or under a different offset relative to the release of τ_i for the infinite sequence of releases of τ_j, greater interference could be exerted upon τ_i.

By subjecting the sequence of activations of τ_j to the transformation just described, the interference cannot decrease. But the product of the transformation would,

once again, be the synthetic worst-case execution distribution - a finding which contradicts the initial supposition.

Thus the theorem is proven.

□

Since the proof for Theorem 2 makes no assumptions regarding other processes potentially interfering with τ_i, in the case of multiple interfering processes it holds for each one of them independently (i.e. $\forall \tau_j \in hp(i)$). In other words, if an activation of some process suffers interference only from activations of higher-priority processes released not earlier than its own release, it is trivial to show that this interference is maximised if we consider the releases to be coincident, with activations of higher-priority processes each characterised by its respective synthetic worst-case distribution.

However, in the general case, a process may also suffer interference from instances of higher-priority processes released earlier than it. Even if all activations of the same process are characterised by the exact same sequence of blocks, the variability in the release times of LBs (only the one coincident with the process release is strictly periodic) allows for scenaria that yield greater interference than a synchronous release of an infinite sequence of activations of each higher-priority process characterised by the respective worst-case synthetic distributions. Observe the example of Figure 17:

Figure 17 plots the execution of a two-process system. Process τ_i, which has a worst-case execution requirement of 13, suffers interference from process τ_j. Activations of τ_j are released every 28 time units and are described by the (ranged) execution distribution: $[4, (4-8), 3, (6), 5]$ (i.e all block lengths are fixed except the first gap from the left, which takes from 4 upto 8 time units - see Figure 17(a)). Thus, $R_j = 26$ and $T_j - R_j = 2$.

In Figure 17(b), τ_j is released synchronously with τ_i (i.e. at $t = 0$). But instead of being characterised by some execution distribution which is actually observable,

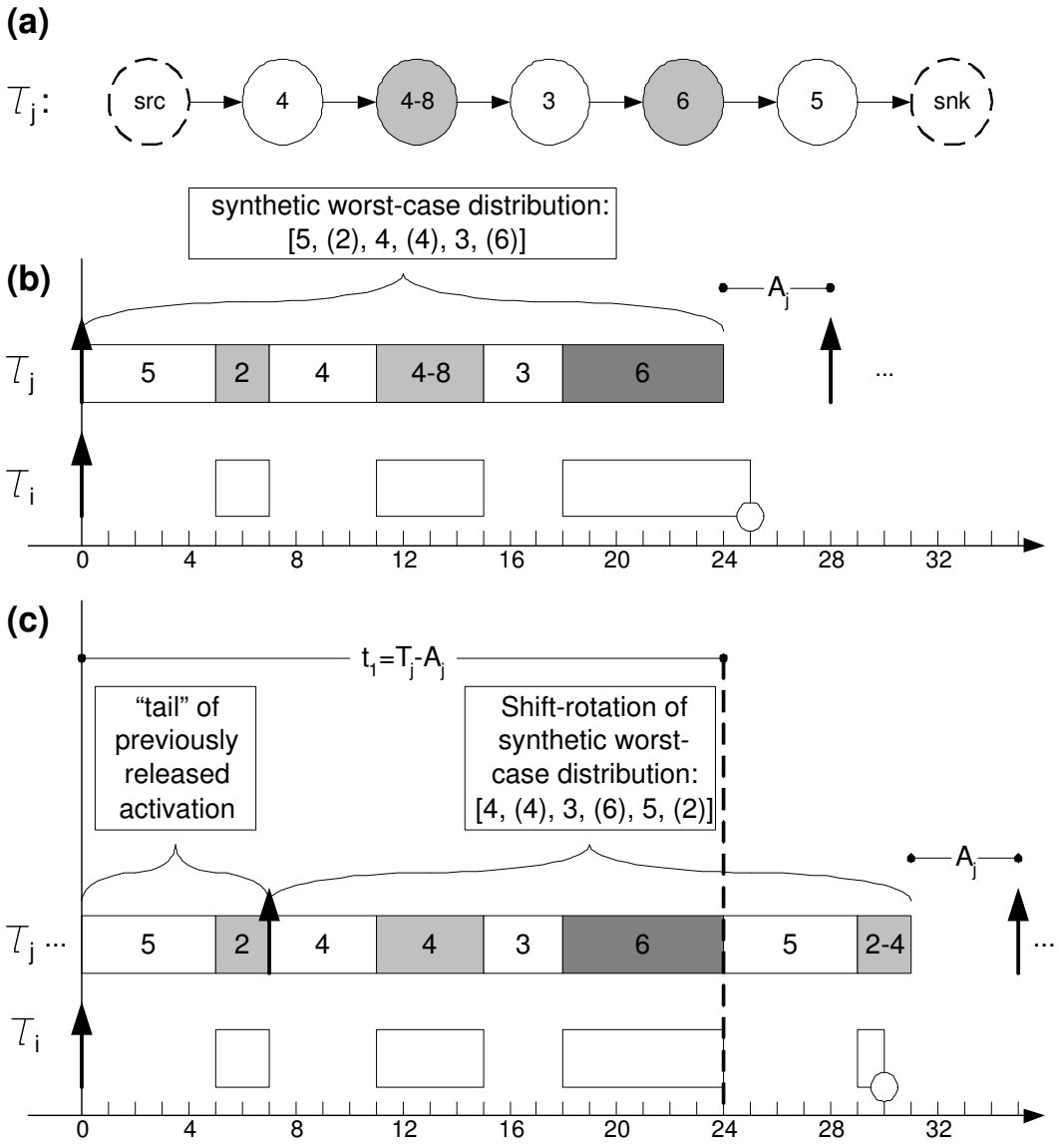

Figure 17: The variability in block placement needs to be accounted for by the analysis.

let its activations (or at least the one released at $t = 0$ - as subsequent releases of τ_j happen to occur after the termination of τ_i, their execution distributions are irrelevant) be characterised by the synthetic execution distribution for τ_j - which is $[5,\ (2),\ 4,\ (4),\ 3,\ (6)]$. This is a constructed scenario (i.e. not observable) under which interference exerted upon on τ_i cannot be less than what the interference from an activation of τ_j released **at** $t \geq 0$ may actually be, under any release offset. Under this scenario, τ_i suffers interference of 12 time units and terminates at $t = 25$.

Note that, despite the inclusion of the notional gap (which represents the minimum possible interval between the termination of any activation of τ_j and the release of the next one) into the synthetic worst-case execution of τ_j, there is still an interval of A_j time units between the termination of one activation of τ_j characterised by said distribution and the release of the next one. In our simple example (where no processes interfere with τ_j), this is entirely because, during the construction of the synthetic distribution, we specifically ask for (non-notional) gaps to be minimal, which tends to reduce the execution time of τ_j. Indeed, in our example, the sum of gap WCETs minus the sum of gap BCETs entirely accounts for A_j, which is 4.

Yet, Figure 17(c) illustrates an actually observable scenario, under which the interference suffered by τ_i exceeds that of the previous scenario because the "tail" of an activation of τ_j released **at** $t < 0$ gets to interfere with τ_i. That activation, released at $t = -21$ is characterised by the execution distribution $[4, (8), 3, (6), 5]$, which is actually observable for τ_j, as seen from its process graph in Figure 17(a). This means that the execution of its final local block, of length 5, is initiated at the same instant that τ_i is released (i.e. at $t = 0$). Moreover, this leaves the interval between the termination of that activation of τ_j and the release of the next one to $T_j - R_j = 2$ (i.e. the minimum possible). Subsequent activations of τ_j however, are characterised by the execution distribution $[4, (4), 3, (6), 5]$ (i.e. the gap of variable length now executes for as long as its BCET, instead of as long as its WCET, as in the activation released at $t = -21$). This results in interference of 17 time units for

τ_i and a response time of 30. As can be verified by exhaustive search (tractable for this simple example), this observable scenario also happens to be the absolute worst case, in terms of interference suffered by τ_i.

Upon closer inspection, we observe that the effects (i.e. interference) of the execution of τ_j upon τ_i happen to be the same as those that would be observed for a sequence of activations of τ_j synchronous with τ_i and characterised by its synthetic worst-case distribution, albeit with a release jitter of A_j (i.e. releases of τ_j at $t = 0,\ T_j - A_2,\ 2T_j - A_2,\ ...$). This "jitter" is akin to the spacing interval of τ_j encountered in the previous scenario being consumed between the first two interfering activations of τ_j (thus making them run "back-to-back"). It accounts for the maximum variability in block placement, relative to the process release. We will be referring to the term denoted by A_j as *worst-case synthetic jitter* throughout the remainder of this thesis.

Interference by τ_j on τ_i under this last scenario (i.e. assuming activations of τ_j characterised by the respective worst-case synthetic distribution and released at $t = 0,\ T_j - A_2,\ 2T_j - A_2,\ ...$) has been proven [18] to be an upper bound for the actual worst-case for such a two-process system. We thus call it the *synthetic worst-case scenario*. An upper bound for the WCRT of τ_i as a function of its (worst-case) execution requirement may be derived from Equation 11:

$$R_i = C_i + \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{11}$$

where ${}^{\xi}X_{j_k}$ denotes the length of the k^{th} local block of τ_j, as appearing in its synthetic worst-case distribution, ${}^{\xi}O_{j_k}$ is (a lower bound for) the offset of the k^{th} local block within an activation of τ_j characterised by the synthetic worst-case distribution (relative to the release of the activation).

For the offsets, the following values may be used:

$$
{}^{\xi}O_{j_k} = \begin{cases} 0 \quad \text{if } k = 0 \\ \\ \sum_{m=1}^{k-1} {}^{\xi}X_{j_m} + {}^{\xi}G_{j_m} \quad \text{otherwise} \end{cases} \tag{12}
$$

We have thus shown that, in our two-process system, interference exerted by τ_j on τ_i under the synthetic worst-case scenario (i.e. interference that would result from activations of τ_j characterised by the respective worst-case synthetic distribution and released at $t = 0,\ T_j - A_j,\ 2T_j - A_j,\ ...$) is an upper bound for interference in the actual worst-case. This finding was originally formulated in [18].

We proceed to discuss what holds for systems with more than two processes:

The variability in block placement (expressed by the jitter term) was, in our two-process system, entirely due to the variability in the execution time of gaps (since local blocks have to be maximal anyway for interference to be maximised). Thus we could use $A_j = G_j - \hat{G}_j$. Yet, in the general case (when there would be other processes, of even higher priority, interfering with τ_j), the variability in block placement would depend as well on the variability in interference suffered by blocks of τ_j. However (as also shown in [18]), it is safe (i.e. does not lead to optimistic bounds) to use, within the analysis, a jitter term of

$$
A_j = G_j - \hat{G}_j = \sum_{\substack{k \\ \tau_{j_k}:gap}} (C_{j_k} - \hat{C}_{j_k}) \tag{13}
$$

for each interfering process $\tau_j \in hp(i)$ (i.e. the same as in the case that it were the only interfering process). The justification is that whichever process interferes with τ_j (at an instant when an activation of entirely software-based process τ_i has not yet terminated) also interferes with τ_i at the same instant. Thus, Equation 11 is generalised to

$$R_i = C_i + \sum_{j \in hp(i)} \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{14}$$

We note that the validity of those bounds was proven for the case that τ_i is entirely software-based (i.e. has no gaps). However they are also valid (albeit more pessimistic, as already noted in Section 6 of Chapter 4 as Weakness #2) even in the case that τ_i has gaps. The same justification as the one provided by Corollary 1 in the context of the more basic model holds: gaps are immune to interference so interference can only be less, in reality, than what it would have been if τ_i was instead executing in software for the respective intervals.

The synthetic approach to the derivation of upper bounds to the WCRTs of processes in limited parallel systems improves on the simple approach described in Section 4 of Chapter 4 (i.e. the bounds derived by use of Equation 6) in two respects:

i) It breaks up the overall interference exerted by each higher-priority process τ_j (in its modelling of it), into multiple contributions (one from each interfering local block) which still add up to X_j but which occur spaced apart in time.

ii) The term acting as jitter, in Equation 6 is $C_j - X_j$, which for linear processes is equal to G_j. In Equation 14 however, it is reduced to $G_j - \hat{G}_j$.

Finally, in systems where only a subset of the processes are linear, it is possible to mix and match from the two approaches:

$$R_i = C_i + \sum_{\substack{j \in hp(i) \\ \tau_j : linear}} I_{j \to i} + \sum_{\substack{j \in hp(i) \\ \tau_j : non-linear}} I_{j \to i} \tag{15}$$

where

$$I_{j\rightarrow i} = \begin{cases} \sum_{\substack{k=1 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} & \text{if } \tau_j \text{ is linear} \\ \left\lceil \frac{R_j + C_j - X_j}{T_j} \right\rceil C_j & \text{otherwise} \end{cases}$$

5.2.1 In the presence of shared resources

In the case that shared software resources exist in the system, access to which is managed by the ICPP, upper bounds on worst-case blocking terms are still derivable by use of Equation 8. No changes are required to the synthetic analysis as formulated. Equation 15 is simply adapted to

$$R_i = C_i + B_i + \sum_{\substack{j \in hp(i) \\ \tau_j : linear}} I_{j\rightarrow i} + \sum_{\substack{j \in hp(i) \\ \tau_j : non-linear}} I_{j\rightarrow i} \tag{16}$$

The length of the notional gap is still derived as $T_j - R_j$, whether there are shared resources in the system or not.

5.2.2 Conceptual comparison with other work

Within our literature review (see page 56) we discussed the *multiframe process model* and its associated analysis, both of which introduced by Mok et al. [60, 61]. Much of that discussion focused on the concept of *accumulative monotonicity* [60, 61]. As shown [60, 61], worst-case interference exerted by any accumulatively monotonic N-frame process τ_j on some lower-priority process occurs under coincident release of τ_i with the release of a specific frame (from among the N) of τ_j.

However, in the general case, in a system consisting of multiframe processes, it is possible that not all of them will be accumulatively monotonic. Which is is why Mok

et al. [61] introduce a transformation which makes such process sets amenable to analysis using the techniques originally formulated in the context of accumulatively monotonic processes. Essentially, every non-accumulatively monotonic process is notionally replaced by an accumulatively monotonic one, derived under the above mentioned transformation (which we described in our literature survey, see page 59), for the purposes of analysis.

Likewise, for the purposes of bounding the interference exerted by a linear process in the worst case, we notionally substitute a process by one characterised by the worst-case synthetic execution distribution (and jitter) of the actual process; these are derived from the actual process via the transformation that we formulated earlier as part of our contributions. This is one conceptual similarity between the approach of Mok et al. and ours, although of course, the context differs.

Another similarity is that between the concept of accumulative monotonicity in the context of multiframe processes (a property met by the output of the transformation of Mok et al.) and the reordering of local code blocks by order of decreasing worst-case execution requirement for the purposes of the construction of a synthetic worst-case execution distribution. Both transformations involve a notional reordering of, respectively, releases of a process characterised by different WCETs (in the case of Mok et al.) and releases of local code blocks belonging to the same process (in our work).

However, an important difference is that for the derivation of a synthetic worst-case distribution we enforce strict monotonicity, not accumulative monotonicity. This is necessary because unlike the model of Mok et al., in the general case it is not only the worst-case execution requirement which varies between successive requests by a given process for the processor, but additionally, the minimum temporal separation between such requests. It is, ultimately, for this same reason that gaps in a synthetic execution distribution appear in order of increasing size.

5.3 Graph linearisation

In Section 1 of Chapter 5 it was implied a process would need to be linear in structure for it to be possible to use the synthetic analysis so as to bound the interference exerted by it. We will show how it is possible for some process graphs to be reduced to a linear form so as to enable the generation of a synthetic worst-case distribution for the respective process.

The reduction to a linear form (where possible) is performed by iterative transformation. In each step, either the *interchange*, the *sandwich* or the *bypass transformation* [1] (collectively called basic transformations and described later within this section) is applied. As before, node coloring denotes whether the respective code block executes in hardware or software.

We proceed to describe each of the basic transformations. The reader may find useful to also refer to Figure 18 at the same time, which provides visualisations of those transformations. Each of our transformations is a trivial application of established graph reduction theory (covered in detail in [1, 62], along with its associated terminology).

- **interchange transformation:** If there exist two sets of nodes, α, β, both of cardinality greater that one, and
  - nodes in α are the same color as each other,
  - nodes in β are the same color as each other,
  - for each node in β, the set of direct ancestors is α,
  - for each node in α, the set of direct descendants is β,

[1] Although these transformations are a trivial application of standard established graph reduction theory (covered in detail in [1, 62]), their names were coined by us and are not part of some standard terminology.

then the following transformation is applied:

Edges originally from nodes in α to nodes in β are removed. A notional node ν_0 (with both its BCET and WCET being 0) is inserted, as well as edges from every node in α to ν_0 and from ν_0 to every node in β.

Given that ν_0 has zero execution time, it can be subsequently treated as being of any color in subsequent transformations.

- **sandwich transformation:**

  For a set α of nodes of the same color, let the node ν_{icd} be the *immediate common dominator* (which we define here as the common dominator (see [1, 62] for definition of domination/postdomination) postdominated by all common dominators not in α) and the node ν_{icp} be the *immediate common postdominator* (defined as the common postdominator dominating all common postdominators not in α). If, for all possible paths from ν_{icd} to ν_{icp}, only nodes belonging to α are traversed and there is no direct edge from ν_{icd} to ν_{icp}, then α can be reduced to a single node ν, as follows:

  All nodes belonging to α and edges to or from them are eliminated. A node ν of the same color as the members of α and edges from ν_{icd} to ν_0 and from ν_0 to ν_{icp} are inserted. The BCET and WCET of ν are determined by path analysis of the original subgraph.

- **bypass transformation:** When a node ν_b has a single immediate ancestor ν_a and a single immediate descendant ν_d and there also exists a direct edge from ν_a to ν_d, then that edge may be eliminated and ν_b replaced by a node ν of the same color, with $\hat{C}_\nu = 0$ and $C_\nu = C_{\nu_0}$.

An example of a process being linearised by successive application of different kinds of transformations is presented in Figure 19. Not all process graphs are linearisable; a counter-example is depicted in Figure 20. For a graph to be linearisable there

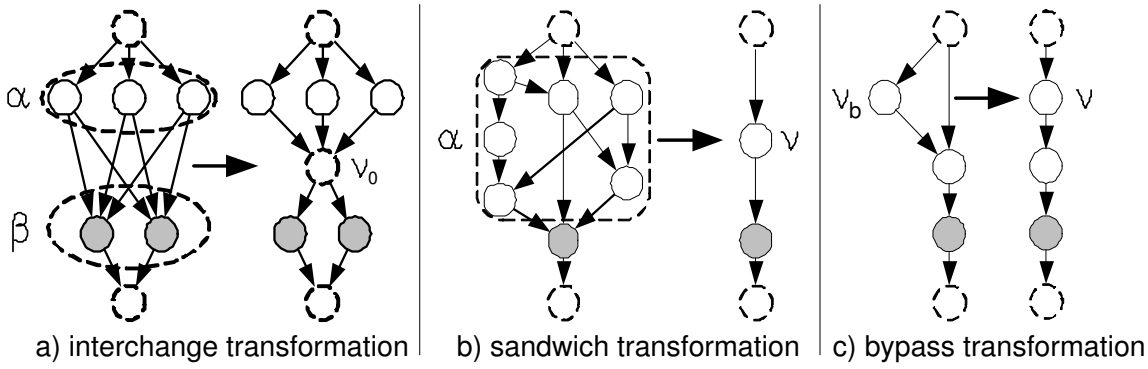

Figure 18: Depiction of the basic graph transformations

must exist a sequence of basic transformations which, if iteratively applied to it, outputs a linear graph where each node differs in color from its immediate ancestor and from its immediate descendant.

Iterative application of the basic transformations to a process graph preserves the upper and lower bounds on time spent executing on any specific processing processing element (thus, also upper/lower bounds on overall execution time). All execution distributions observable for the original graph (i.e. corresponding to any possible combination of control flow and associated block execution latencies) are also observable for the tranformed graph.

The importance of graph linearisation lies in that, if it is possible to linearise a process graph, then it is possible to model the worst-case interference exerted by that process upon lower-priority processes by means of a respective worst-case execution distribution and associated jitter term. This, in turn, reduces the pessimism in the calculation of upper bounds to the WCRTs of lower-priority processes (as the simple analysis for the limited parallel model would have to be used instead).

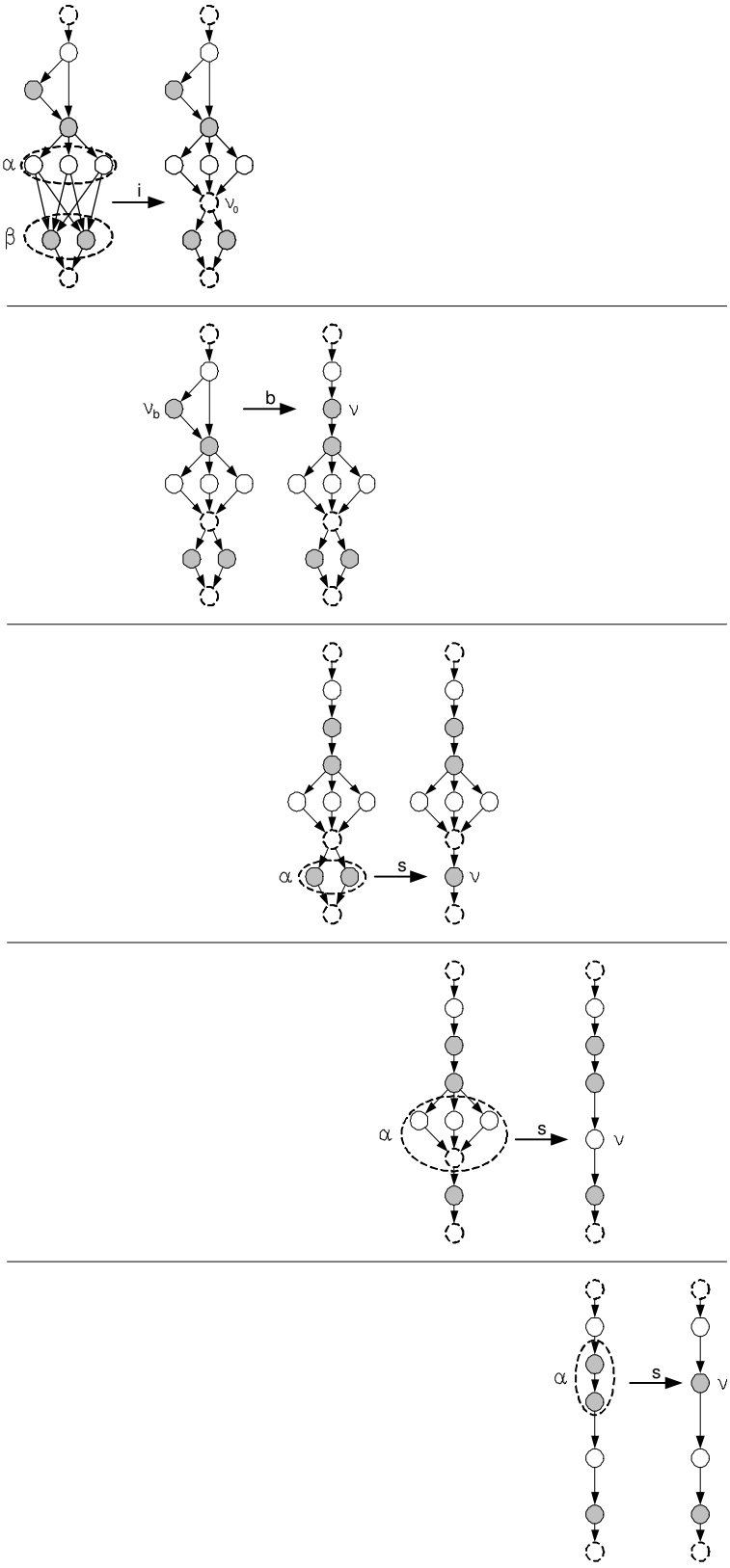

Figure 19: Linearisation of a process graph through successive transformations

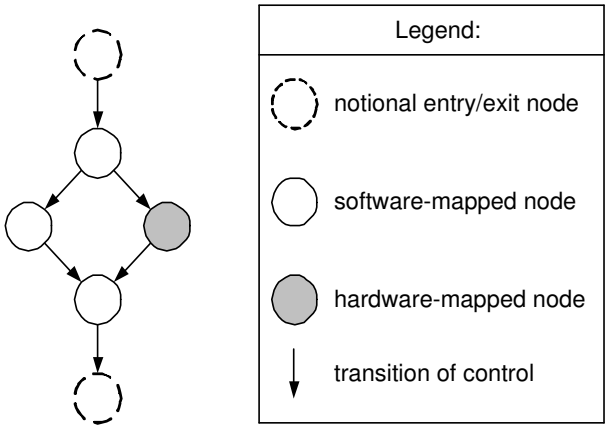

Figure 20: This simple graph is not linearisable. In fact, it is not possible for any of the basic transformations to be applied to it.

5.3.1 Linearisation algorithm

Neither an *optimal* algorithm which, given any process graph as input, tests it for linearisability nor an *optimal* algorithm for iteratively transforming a linearisable graph to its linear form are provided in this text. Their formulation is left as future work. However, we anticipate that process graphs will generally be coarse-granularity models of process structure, thus consisting of few nodes (i.e. at most 10 prior to linearisation). In that case, linearisation may be carried out by inspection.

Nevertheless, we outline an *exhaustive* algorithm which accomplishes the task of graph linearisation (if possible):

Consider the dominator tree of the process graph to be linearised (if possible). Let d be the sequence of nodes traversed by the path from the node corresponding to the source of the process graph to the node corresponding to the sink of the process graph, of length l^d. Then, it is possible to split the original process graph into $l^d - 1$ subgraphs with single entry semantics, the i^{th} subgraph having as source the node $d(i)$ and as sink the node $d(i+1)$. The original graph is linearisable if and only if all $l^d - 1$ subgraphs are linearisable. The (test for the) linearisation of each subgraph

is performed recursively, as for the original graph.

The recursion will not proceed any deeper whenever a subgraph is not further decomposable in the manner described (i.e. in its dominator tree, the node corresponding to its sink, is a direct descendant of the node corresponding to its source). Then, if not already linear, the subgraph is tested for linearisation according to the following procedure:

All possible sequences of successive transformations are tested. If the output of any of them is linear, then the subgraph is indeed linearisable as proven by construction. If, however none give linear output, then the original process graph is not transformable by our algorithm.

The possible sequences of successive transformations are generated by testing all possible (combinations) of subsets of nodes in the graph against the criteria for each of the transformations we introduced.

The complexity of the algorithm outlined is clearly exponential. However, if the node count for the original graph is low (i.e. at most 10 nodes, as per our earlier assumption), it should still be tractable.

5.4 Remaining issues addressed

Our synthetic analysis removes, for the most part, the source of pessimism identified as Weakness #1 in Section 6 of Chapter 4. In this section, we will provide analytical contributions which reduce the other main source of pessimism (identified in Section 6 of Chapter 4 as Weakness #2). In order to do so, we must first introduce extensive notation (which follows) and upon which we shall rely for the formulation of our analytical approach.

5.4.1 Notation and associated concepts

Before we proceed with our notation, we generalise two basic concepts in our terminology:

Until now, the term *release* has only referred to processes. We expand on that concept so that it covers (local or remote) code blocks as well: A block of code is said to be released when it is first ready to execute. Similarly, we expand on the concept of response time to refer to individual code blocks. The response time of a code block, part of some process activation, is the interval between the release (as previously defined) of said block and the completion of its execution. Just as with process response times, the response time of a code block accounts for time spent by that specific block executing, time spent preempted and time spent blocked on some shared resource.

Consider the following algebraic notation:

- $R(\tau_i, \Delta x)$ returns an upper bound on the WCRT of a local block of length Δx, with the priority of (and in place of) some process τ_i.

- $R(\tau_i, \mathbb{K})$ returns an upper bound on the WCRT of an activation of τ_i characterised by a given execution distribution $\mathbb{K}$. The second argument being an execution distribution rather than an execution length (i.e. a scalar) acknowledges the fact that gaps are immune to interference. Depending on the type of analysis employed, this realisation may be used to reduce the pessimism (or simply disregarded).

Which type of analysis is used to derive the bounds denoted by the above is an orthogonal issue. Wherever there is need to differentiate or wherever there would otherwise be ambiguity we will be using a left superscript: c for *classic* analysis (i.e. based on [54]), o for the *original* analysis targeted at the limited parallel model [9, 10]

(i.e. the simple analysis presented in Section 4 of Chapter 4) and ξ for the *synthetic* analysis.

By inspection of Equation 10, it is seen that the actual execution distribution for τ_j is disregarded and only its overall (i.e. irrespective whether in software in hardware) execution requirement is considered. Thus:

$$^{o}R(\tau_i, \mathbb{K}) = {}^{o}R(\tau_i, |\mathbb{K}|)$$

In other words, the simple analysis would output the same upper bound for the WCRT of any two activations of τ_i characterised by different execution distributions but which happen to have the same length [2].

Under that analysis:

$$^{o}R_i = {}^{o}R(\tau_i, C_i)$$

The same property, by inspection, holds for the classic uniprocessor analysis as well:

$$^{c}R_i = {}^{c}R(\tau_i, C_i)$$

In the case of a ranged distribution, say $\mathcal{K}$, which is essentially a set of exact execution distributions, we define $R(\tau_i, \mathbb{K})$ as

$$R(\tau_i, \mathcal{K}) = \max_{u} R(\tau_i, \mathbb{K}_u),\ \forall \mathbb{K}_u \in \mathcal{K}$$

[2] Actually, if two different execution distributions observable for the same process are characterised by a different number of transitions to local execution (either upon release or right after a gap), the number of potential blocks suffered by the process (hence also the respective worst-case overall blocking terms) would differ. Since, however, the analysis for the limited parallel model calculates worst-case per-process blocking terms using Equation 8 (wherein the scalar n_{τ_j} is a static property of the process - see Equation 9), this is not captured by the analysis.

Note that for a gap whose execution time ranges from $\hat{g}$ to g, no analysis is necessary (as gaps suffer no interference) and the WCRT of the gap is equal to its WCET (g in our case).

We proceed to prove the following set of corollaries which will prove handy later on:

Corollary 2

$\forall \Delta t_1,\ \Delta t_2 > 0$*:*

$$^{c}R(\tau_i, \Delta t_1 + \Delta t_2) \leq {}^{c}R(\tau_i, \Delta t_1) + {}^{c}R(\tau_i, \Delta t_2)$$

Proof:

Consider that process τ_i has a worst-case execution time of $\Delta t_1 + \Delta t_2$. Consider also some breakpoint inside the process code, reachable after at most Δt_1 units of execution. Once the breakpoint is reached, then the process may execute for at most Δt_2 time units before it terminates. By then replacing τ_i with two distinct processes τ_y, τ_z with respective WCETs of Δt_1, Δt_2 executing back to back (i.e. with no idle time between the termination of τ_y and the release of τ_z), scheduled at the same priority as τ_i, the overall response time (i.e. from the release of τ_y to the termination of τ_z) cannot decrease.

The conditions ensuring worst-case interference on each process individually are the same (a critical instant). If τ_y is released on a critical instant, its WCRT is indeed $R(\tau_i, \Delta t_1)$. However, with that as a given, τ_z may or may not be released under a critical instant.

- If it is, then its response time is $R(\tau_i, \Delta t_2)$. Then the response time for the whole sequence (by definition $R(\tau_i, \Delta t_1 + \Delta t_2)$, since τ_y is released on a critical instant) will be $R(\tau_i, \Delta t_1) + R(\tau_i, \Delta t_2)$.
- If not, then its response time will be some $R^{'} < R(\tau_i, \Delta t_2)$, thus $R(\tau_i, \Delta t_1 + \Delta t_2) < R(\tau_i, \Delta t_1) + R(\tau_i, \Delta t_2)$

In the above equations it was assumed (pessimistically) that τ_y, τ_z may block on the same set of resources. In the calculation of individual WCRTs it was also assumed (pessimistically again) that any of the processes encounters its worst-case blocking factor, irrespective of whether the other does. Thus the findings are not compromised by the presence of blocking.
□

The same proof (with the difference that the condition ensuring worst-case interference is not a critical instant, but instead the respective condition specific to the the simple analysis pertaining to the limited parallel model) can be used to prove the same property for limited parallel systems analysed under the simple approach:

Corollary 3
$\forall \Delta t_1,\ \Delta t_2 > 0$*:*

$$^{o}R(\tau_i, \Delta t_1 + \Delta t_2) \leq {}^{o}R(\tau_i, \Delta t_1) + {}^{o}R(\tau_i, \Delta t_2)$$

Likewise for the synthetic approach:

Corollary 4
$\forall \Delta t_1,\ \Delta t_2 > 0$*:*

$$^{\xi}R(\tau_i, \Delta t_1 + \Delta t_2) \leq {}^{\xi}R(\tau_i, \Delta t_1) + {}^{\xi}R(\tau_i, \Delta t_2)$$

In the general case, the processes for whose response times we wish to find an upper bound, are characterised by execution distributions with both local blocks and gaps. Under the synthetic analysis, as originally published in [18], this structure would be disregarded and the derived bounds would be a function solely of the overall WCET of the process in consideration. For some linear (or linearised) process τ_i (characterised by the ranged execution distribution $\mathcal{K}_i$), this property of the analysis may be expressed in mathematical terms (since $|\mathcal{K}_i| = C_i$) as:

$$R_i^{joint} = {}^{\xi}R(\tau_i, |\mathcal{K}_i|) = {}^{\xi}R(\tau_i, C_i) \tag{17}$$

However, another approach is possible. Another valid upper bound for the WCRT on τ_i may be computed as the sum of the individually computed WCRTs for each one of its constituent blocks. This bound (assuming that any adjacent LBs and any adjacent gaps are already merged - as mandated by Corollary 4 - which would then result in interleaved gaps/LBs) is then given by:

$$R_i^{split} = \sum_{\tau_{i_m} \in \tau_i} {}^{\xi}R_{i_m} = \sum_{\substack{\tau_{i_m} \in \tau_i \\ \tau_{i_m}:LB}} {}^{\xi}R_{i_m} + \sum_{\substack{\tau_{i_m} \in \tau_i \\ \tau_{i_m}:gap}} C_{i_m} \tag{18}$$

The question then arises: In the general case, does one of the two methods of deriving bounds (i.e. "joint" / "split") consistently outperform the other? The answer is: no. We are going to illustrate this fact by use of the example of Figure 21.

In Figure 21(a), lower-priority process τ_1 is characterised by execution distribution [5 , (3) , 4] (we chose invariant code block execution times for convenience). Higher-priority process τ_2 has a (worst-case) execution time of 6 and a period of 19 time units. We deliberately chose τ_1 to be sporadic so that any discussion about relative release offsets of the two processes becomes meaningless. We also chose the single higher-priority process τ_2 to consist of a single block which is local so that the analysis for the limited parallel model (simple or synthetic irrespective) and the classic uniprocessor analysis give the same results.

Using the "split" approach, we derive for each constituent block of τ_1 its WCRT. For the gap corresponding to τ_{1_2} no analysis is necessary since the respective WCRT is the same as the length of the gap itself. By adding up the individual WCRTs we come up with $R_1^{split} = 24$.

If, on the other hand, only the WCET of τ_1 is considered and we (pessimistically) assume that whenever τ_2 is released it immediately preempts τ_1 (which might how-

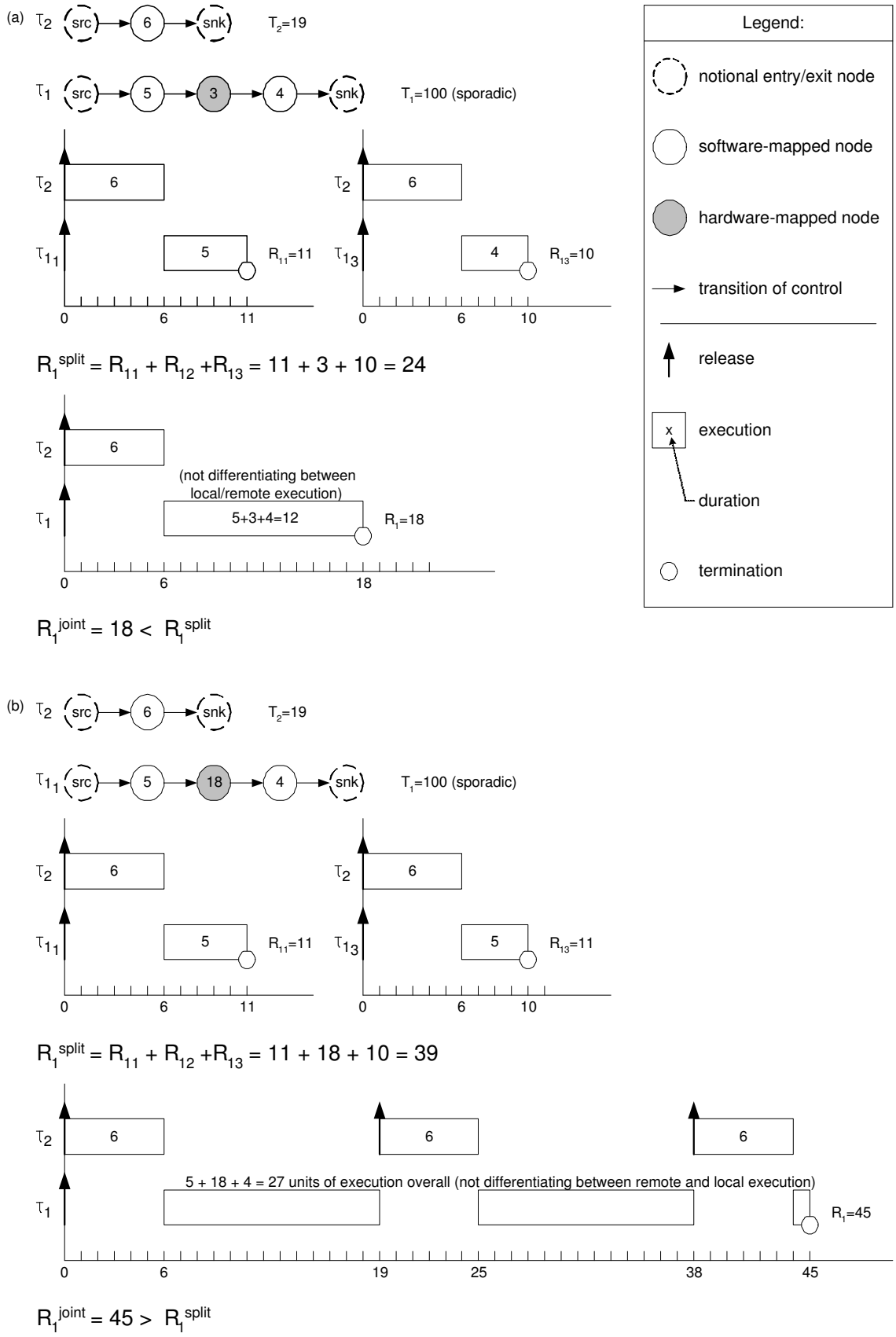

Figure 21: Different approaches to WCRT calculation

ever be executing in hardware on that instant, hence would not be preempted, immediately at least), the WCRT analysis will be identical to having an entirely software-based τ_1 with the same WCET as before. The output of the analysis is then an upper bound to the WCRT of τ_1 which is $R_1^{joint} = 18$.

Thus, for this particular example, the "joint" approach outperforms the "split" approach.

However if the gap of τ_1 is modified so that its length is increased to 18 time units, we obtain the example system of Figure 21(b). By analysis of the system, as before, using both approaches, we obtain $R_1^{split} = 39$ and $R_1^{joint} = 45$. Thus, in this example the "split" approach outperforms the "joint" approach (which is the opposite of what was observed for the previous example).

We observe that for any possible input (i.e. system to be analysed) both approaches do output valid (i.e. not optimistic) upper bounds for process WCRTs. A valid (but somewhat naive) approach could then be to independently calculate upper bounds for the WCRT of some process using each of the two approaches in turn, and then pick the least pessimistic (i.e. the smallest value). However, to eliminate pessimism to the greatest extend possible, one would have to consider all possible decompositions of the process activation into subsequences of code blocks. Consider another example, that of the process set of Figure 22:

By working as before, we derive $R_1^{split} = 126$ and $R_1^{joint} = 146$. Yet, if τ_1 is decomposed into subsequences $\tau_{1_{1\rightarrow 3}}$ (consisting of blocks τ_{1_1}, τ_{1_2}, τ_{1_3}), τ_{1_4} (standalone code block) and $\tau_{1_{5\rightarrow 7}}$ (consisting of blocks τ_{1_5}, τ_{1_6}, τ_{1_7}) and each subsequence is (recursively) analysed as previously, we derive

$R_{1_{1\rightarrow 3}} = 8$, $R_{1_4} = 100$ and $R_{1_{5\rightarrow 7}} = 8$,

the sum of which (i.e. 116) is a valid upper bound for the WCRT of τ_1 less pessimistic than either R_1^{split} or R_1^{joint}.

By extension, we have shown that, in the general case, to obtain the tightest possi-

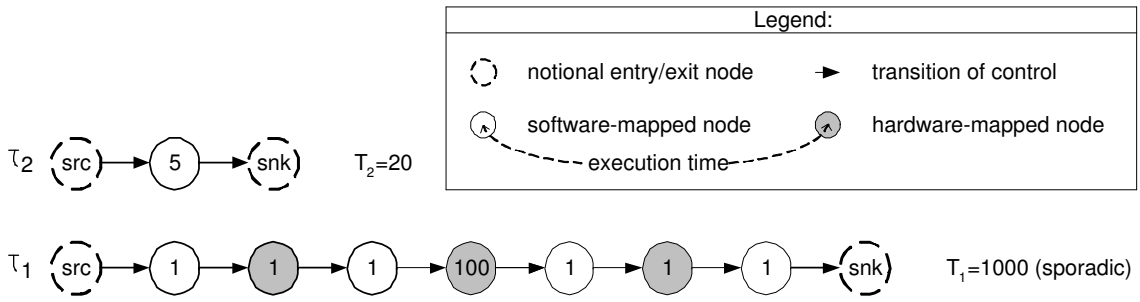

Figure 22: All possible decompositions of τ_1 would have to be considered during WCRT analysis for the elimination of pessimism.

ble bound on the WCRT of a linear process, one would have to consider all possible decompositions of the original process into sequences of code blocks and then derive, for these sequences, upper bounds on their respective WCRT. That may in turn require further decomposition into even shorter "chains" of code blocks. The algorithm is described in pseudocode in Figure 23.

The algorithm is clearly exponential in complexity to the block count of the process in consideration. Thus it is costlier in terms of operations than simply applying the "joint" approach (which is what was assumed by default in [18]). The algorithm removes to a certain degree what was identified in Section 6 of Chapter 4 as Weakness #2 (i.e. that the immunity of gaps to interference was not taken account of). However, while this analytical approach reduces the pessimism in offset-agnostic static analysis, the exact degree to which this is accomplished is highly input-dependent. Thus it may only be quantified by experimentation with real-world systems (possibly as future work). The same is also true of the additional computational complexity that would typically (i.e. in practice) be required for the calculation of these improved WCRTs. We note, however, that we expect linear processes to typically have few gaps (i.e. $1-3$) in the vast majority of cases (or else the model would be too fine-grained for codesign), which permits the algorithm to be tractable.

Note that there is nothing which prevents either of the two approaches ("joint" or

```
int wcrt_joint(int start, end)
{if ((start==end) && (is_remote_block(start)) //if examining a single RB
  return C[start]; //the gap length, as it never suffers interference
 else
 {int joint_length=0;
  for (int i=start;i<=end;i++)
   joint_length=joint_length+C[i];
  int joint_bound=r(current_process, joint_length);
  return joint_bound;
 }
}

int find_bound_for_wcrt(int start, int end)
{int bound=wcrt_joint(start, end);
 if (start!=end)
  for (int i=start; i<end; i++)
   bound=min(bound, find_bound_for_wcrt(start,i)+find_bound_for_wcrt(i+1,end));
 return bound;
}

int main() //entry point of the program
{return find_bound_for_wcrt(1, block_count_of(current_process);
}
```

Figure 23: The algorithm employing both the "split" and the "joint" approach for the derivation of upper bounds on process WCRTs (in C-like pseudocode)

"split") from being coupled with either the simple or the synthetic approach, with respect to the characterisation of interference from higher-priority processes. In practice though, we expect that the algorithm of Figure 23 will be used together with the synthetic approach.

Note also that, for convenience, we chose examples (i.e. those of Figures 21 and 22) without any shared resources when comparing the "split" and "joint" approaches but placed no assumption that, in the general case there will not be any. Equations 17 and 18 and the algorithm of Figure 23 are valid in any case, as can be seen by inspection.

5.5 A local optimisation

We present one last improvement to the synthetic analysis. This leads to the reduction, in specific cases, of the term used as a worst-case synthetic jitter in our analysis (which, in turn leads to tighter bounds on derived process WCRTs).

Consider the example of Figure 24. In Figure 24(a), the process graph for some process τ_j is shown. That graph may also be described as a ranged execution distribution: $\mathcal{K}_j = [5-10,\ (20-25),\ 5-10,\ (3-8)]$. We wish to derive, for that process, its synthetic worst-case execution and associated term acting as jitter (i.e. A_j) for the purposes of bounding the interference it exerts on other lower-priority processes. As processes are analysed in order of descending priority, at the time that the synthetic execution distribution for τ_j is constructed, an upper bound R_j for its WCRT has already been computed. Suppose that $T_j - R_j = 10$, which ensures at least 10 idle time units between the termination any activation of τ_j and the release of the next one. Let us then simulate the procedure which constructs the synthetic execution distribution for τ_j:

Figure 24(b) depicts the execution distribution that is derived by having the LBs/gaps of $\mathcal{K}_j$ be maximal/minimal respectively and by appending the notional gap of length

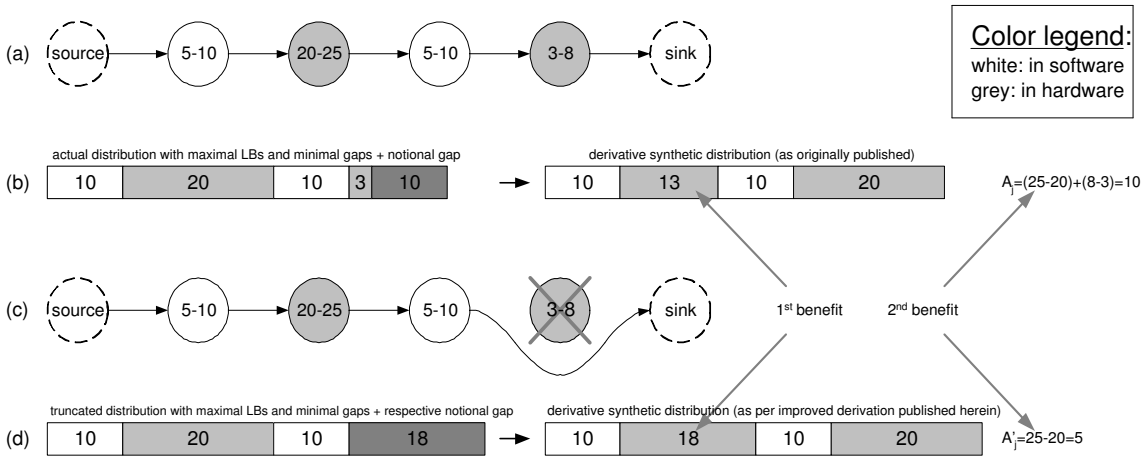

Figure 24: If remote, the last block of a process may be disregarded for the purposes of constructing its synthetic worst-case distribution, as shown. The analysis benefits from reduced pessimism as a result.

$N_j = 10$ to it - this merges with the neighboring gap to form a 13-unit long gap. If LBs and gaps are then shuffled so that, starting with a LB, they appear interleaved in order of decreasing/increasing length (respectively) we obtain the synthetic worst-case execution distribution, which turns out to be: $\mathbb{K}_j = [10,\ (13),\ 10,\ (20)]$. As for the synthetic worst-case jitter A_j, it is calculated as 10.

However, consider what the actual (i.e. in terms of scheduling decisions on the processor) impact would be if the last gap (i.e. the one whose length ranges from 3 to 8) was removed the from the process graph of τ_j: the modified process graph is depicted in Figure 24(c). In terms of interference on other processes or impact on scheduling decisions, the existence of the eliminated gap is inconsequential. However, the impact upon the output of the timing analysis for processes upon which τ_j exerts interference has to be examined. What is clear though is that upper bounds for the WCRTs of lower-priority processes derived *with* the gap and *without* the gap are both valid, since the analysis is safe for each case respectively, and the actual worst-case response times (unknown to us, as the analysis only provides upper bounds)

have to be the same. The reason is that, as already pointed out, one graph could be swapped for the other and scheduling decisions on the processor would remain the same.

Proceeding as before, albeit for the graph of Figure 24(c), we find that it corresponds to the ranged execution distribution $\mathcal{K}_j^{'} = [5-10,\ (20-25),\ 5-10,\ (3-8)]$. The length of the notional gap would now be $N_j^{'} = T_j - R(\tau_j, \mathcal{K}_j^{'})$ (which must be calculated). We observe that an upper bound for $R(\tau_j, \mathcal{K}_j^{'})$ is $R(\tau_j, \mathcal{K}_j) - G_j^{last} = R_j - G_j^{last}$, where G_j^{last} is the WCET of the gap that we removed. Otherwise, the "intact" τ_j could potentially exceed its WCRT (which is a contradiction). Hence, in the general case, $N_j^{'} > N_j$. In our example (where $G_{last}^{j} = 8$), $N_j^{'} = 18 > 10 = N_j$ and the derived synthetic worst-case distribution then is $\mathbb{K}_j^{'} = [10,\ (18),\ 10,\ (20)]$. This distribution is identical to $\mathbb{K}_j$ except for one gap (which increases from 13 to 18 time units). This impacts positively upon the schedulability analysis, because the LB which follows the gap is pushed to the right along the time axis. In terms of Equation 14 the offsets of any LBs to the right of the gap (a single LB in this case), increase accordingly. Moreover, the worst-case synthetic jitter A_j in that same equation is always reduced by $G_j^{last} - \hat{G}_j^{last}$ (the WCET minus the BCET of the eliminated gap). In our example, the worst-case synthetic jitter A_j for the initial graph (derived as per Equation 13) incorporates the variability in execution time of the last gap, even though it precedes no LB. Upon elimination, only the variability in the execution time of the remaining gaps need by accounted for, hence $A_j^{'} = 5 < 10 = A_j$.

In the general case, consider for some process τ_j ending with a gap, the sequence

$$^{\xi}g_j(1),\ ^{\xi}g_j(2),\ ...,\ ^{\xi}g_j(n(\tau_j))$$

(introduced in Section 2 of Chapter 5), returning the members of the set $^{\xi}\mathbf{G}_j$ in ascending order. The member of $^{\xi}\mathbf{G}_j$ associated with the notional gap will be returned by $^{\xi}g_j(a)$, for some $1 \leq a \leq n(\tau_j)$. Consider now the respective set $^{\xi}\mathbf{G}_j^{'}$

for the modified graph (i.e. with its final block, a gap, being removed). ${}^{\xi}\mathbf{G}_j^{'}$ will only differ in one element - the one corresponding to (or, in case τ_j also starts with a gap, incorporating) the notional gap. This element is then returned by ${}^{\xi}g_j^{'}(b)$, for some $1 \leq b \leq n(\tau_j)$ Obviously, ${}^{\xi}g_j^{'}(b) > {}^{\xi}g_j(a)$, thus, by necessity, $b \leq a$. We observe (assuming that $G_j^{last} \neq \hat{G}_j^{last}$ – otherwise ${}^{\xi}\mathbf{G}_j$ and ${}^{\xi}\mathbf{G}_j^{'}$ would be identical) the following three cases:

- $1 \leq b < a \leq n(\tau_j)$

  Then, ${}^{\xi}g_j^{'}(k) = {}^{\xi}g_j(b)$ for $1 \leq k < b$ and ${}^{\xi}g_j^{'}(k) \geq {}^{\xi}g_j(b)$ for $b \leq k \leq n(\tau_j)$. By inspection then of Equation 12 we observe that the offsets for the $(b+1)^{th}$ local block onwards, in the synthetic execution distribution of τ_j, improve (i.e. increase) – or, in mathematical terms: ${}^{\xi}O_{j_k}^{'} \geq {}^{\xi}O_{j_k}$ $\forall b < k \leq n(\tau_j)$. Additional improvement comes from the reduction of the synthetic worst-case jitter A_j, as already noted.

- $1 \leq b = a \leq n(\tau_j)$

  In this case, ${}^{\xi}g_j^{'}(k) = {}^{\xi}g_j(a)$ for $1 \leq k < a$ and for $a < k \leq n(\tau_j)$ and ${}^{\xi}g_j^{'}(a) > {}^{\xi}g_j(a)$. By inspection then of Equation 12 we observe that the offsets for the $(a+1)^{th}$ local block onwards, in the synthetic execution distribution of τ_j, improve (i.e. increase) – or, in mathematical terms: ${}^{\xi}O_{j_k}^{'} \geq {}^{\xi}O_{j_k}$ $\forall a < k \leq n(\tau_j)$. Additional improvement comes from the reduction of the synthetic worst-case jitter A_j, as already noted.

- $b = a = n(\tau_j)$

  In this case ${}^{\xi}g_j^{'}(k) = {}^{\xi}g_j(a)$ for $1 \leq k < n(\tau_j)$ and ${}^{\xi}g_j^{'}(n(\tau_j)) > {}^{\xi}g_j(n(\tau_j))$. We observe the $n(\tau_j)^{th}$ gap in the worst-case synthetic distribution, which is the only one to increase as a result of our transformation is to the right of any LB

of τ_j. Thus all offsets ${}^{\xi}O_{j_k}$ remain unaffected and the only improvement comes from the reduction of the synthetic worst-case jitter A_j, as already noted.

Thus we have shown, that our transformation (i.e the notional elimination of the last block of τ_j, if remote, before proceeding to construct its worst-case synthetic distribution) may improve the tightness of the analysis and may not, in any case, impact it negatively. No additional computational complexity is involved. Worst-case synthetic distributions are thus to be derived after the respective process has passed through this transformation (if applicable).

5.6 Evaluation

We next offer some evaluation of the analysis formulated within this chapter, first by comparing it to other approaches and then by using it to analyse an example process set.

5.6.1 Comparison with other analytical approaches

The synthetic analysis is more accurate than our basic analysis for the limited parallel model, which was formulated in Chapter 4 (which, in turn, was shown to outperform the uniprocessor analysis). Thus we will proceed with a summary of how it compares to the holistic approach (originally by Tindell et al. [79, 80]; refined by Palencia et al. [64]) and also discuss whether any meaningful comparison can be made with the analysis of Pop et al. [67].

With respect to the holistic analysis

With regard to the holistic analysis, our approach compares favorably on two respects:

- For a linear process, under the worst-case scenario of the holistic approach, **each** one of its constituent code blocks will encounter its worst-case interference (and, thus, also its worst-case individual response time); however the conditions which result in worst-case interference upon one code block might be mutually exclusive with those that result in worst-case interference upon some other code block of the same process.

  Conversely, under the worst-case scenario of our approach, it is the interference suffered **jointly** by all code blocks of the process analysed, for which an upper bound is derived. We explain:

  In Section 4 of Chapter 5 we discussed the "joint" and the "split" approach to worst-case response time calculation. The analysis of Tindell only relies on the "split" approach. Our analysis by contrast considers both the "joint" and the "split" approach, iteratively, for all combinations of code blocks that a process activation may be decomposed to.

- The worst-case scenario under the holistic approach, furthermore, assumes that the interference exerted (on the local code block under analysis) by **each** higher-priority local code block **individually** will be maximal. Again, this is too pessimistic. If two code blocks belong to the same interfering process, at most one of them may be released at the same time as the local block analysed (so that the interference exerted by it is maximised, as per the critical instant theorem of Liu and Layland [54]), not both of them.

  Conversely, our analysis derived a bound for the interference exerted **jointly** by all local code blocks belonging to the same interfering process.

Thus, our approach consistently outperforms the holistic analysis.

The holistic approach remains an invaluable tool for the analysis of distributed heterogeneous multiprocessor systems. Our approach is only applicable a very specific subclass of the class of systems that holistic approach is applicable to. However,

where applicable (i.e. on limited parallel systems), it is considerably more accurate, as shown.

With respect to the analysis of Pop et al.

Regarding the analysis of Pop et al. [67] for distributed embedded systems, we note that although it may be applied to architectures with co-processors, no meaningful comparison is to be made between that approach and ours. The reason for this could be summed up be stating that the analysis of Pop, based on a more general model and different assumptions, solves an entirely different problem. Under our process model (and respective assumptions) however, the problem solved by Pop et al. is a non-issue. We elaborate:

The semantics of process graphs under the two models are different. Process graphs under Pop et al. only specify a partial ordering of operations (see Figure 11(b) in page 86); thus, it is possible, under this model, for two or more operations (unless explicitly disallowed by some precedence constraint) to be active at the same time and (if mapped to the same processing element) compete for execution on the same processing element. Conversely, in our model, within any graph, no more than node may be active on any given instant; thus processes corresponding to nodes belonging to the same graph never compete with each other for a processing element.

The contribution of Pop et al. is that they manage to prove that some nodes may not be active at the same time as each other, due to the timing properties of the system (even when this is not directly specified by some precedence constraint). If so, then the derived worst-case scenario (unlike that of conventional analysis) does not invoke interference from one another (hence becomes less pessimistic). For example, consider the instance of their model in Figure 25.

There are three processors in that example (each denoted by a distinct color). The precedence constraints implied by the graph are as follows:

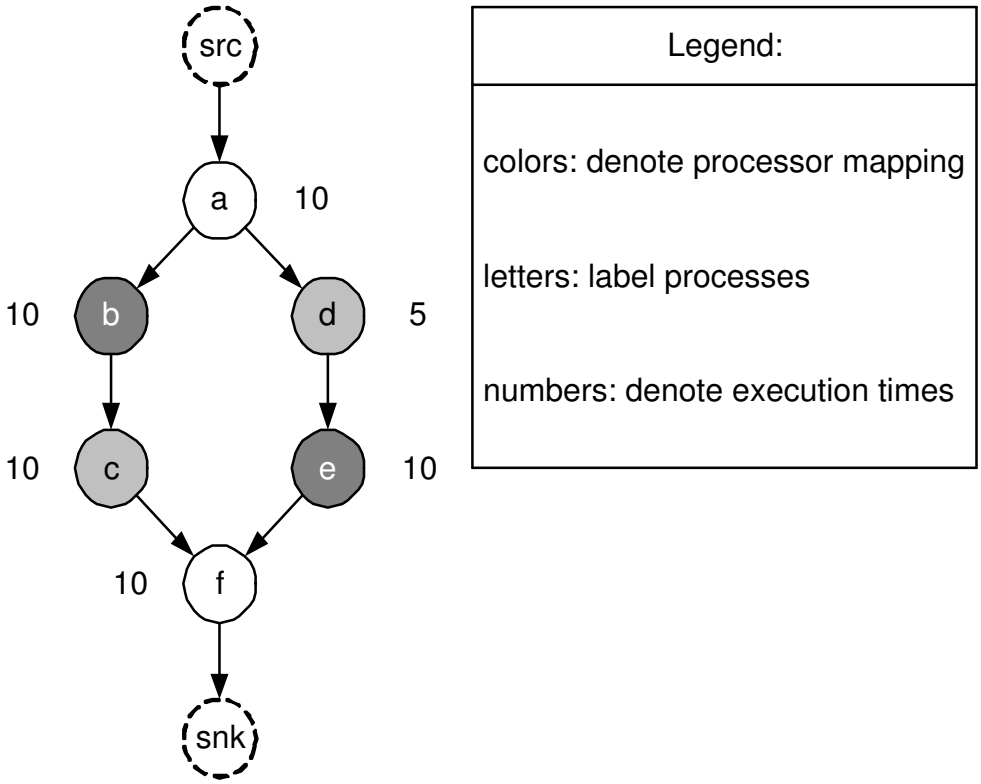

Figure 25: An instance of the process graph model of Pop et al. [67]

- a precedes every other node.
- b precedes c.
- d precedes e.
- f is preceded by every other node.

Thus, d and c being active at the same time is not ruled out. However, given the execution times shown (invariant, for our convenience), d will have terminated by the time c is activated. The same does not hold for b and e.

However, such behavior is not possible under our model, where no more than one node per given graph may be active on any given instant. This is a result of strict (as opposed to partial) precedence constraints in our model.

Pop et al. reduce the pessimism in the derivation of upper bounds on interference suffered by code blocks belonging to the same graph. In our model, such interference is zero anyway, though. Conversely, our approach reduces the pessimism in the derivation of upper bounds on interference suffered by code blocks belonging to other graphs. The approach of Pop et al. does not address this issue.

For process graphs which are instances of the model of Pop et al. but in which edges outgoing from the same graph are only activated under mutually exclusive conditions (an additional constraint so that such graphs are, at the same time, instances of our model as well), the analysis of Pop et al. reduces to the holistic approach (which our analysis outperforms, as shown).

Given that the approach of Pop et al. is more general and solves a problem not present under the additional constraints imposed by our model, a comparison with our approach would not be meaningful.

5.6.2 An example

We evaluate our analysis via an example. This example is an adaptation of the system of Table 2 (see page 98). More specifically, we constructed linear process graphs which fit the parameters of the processes of Table 2. These graphs are shown in Figure 26. Observe that while the execution times of software code blocks are highly variable, those of hardware functions are not; this reflects engineering reality. While it is not uncommon for the WCET of software code to be orders of magnitude greater than the respective BCET, hardware design libraries tend to balance paths. Remaining system parameters are shown in Table 6.

We proceed to calculate, for this system, upper bounds for process WCRTs under each of the following approaches: uniprocessor, holistic [3], our basic analysis, and the synthetic analysis. A comparison of those is provided in Table 8.

For illustration purposes, the synthetic worst-case execution distributions and jitters for the processes of our example (derived during the calculation), are given in Table 7.

While we note the improved accuracy in comparison to the uniprocessor analysis (${}^{c/u}R$) and the holistic analysis (${}^{c/h}R$), it is interesting that, for our specific example, the synthetic analysis (${}^{\xi}R$), despite being more detailed, derives the same numbers as our basic analysis (${}^{o}R$) for all processes but the lowest-priority one (for which it achieves a modest improvement). We can think of two interpretations, not necessarily mutually exclusive:

[3] Note that the original formulation of the holistic analysis [79, 80] (given zero communication latencies as here) derives the worst-case release jitter for a code block which is not the first within the process it belongs to as the sum of the individual WCRTs of the code blocks preceding it within the same process. However, this is in turn based on the assumption that the BCET of any code block may be arbitrarily small (i.e. zero). Since, however, we know the process BCETs, so as to be "fair", we will be subtracting from the above jitters the sum of the BCETs of the preceding code blocks.

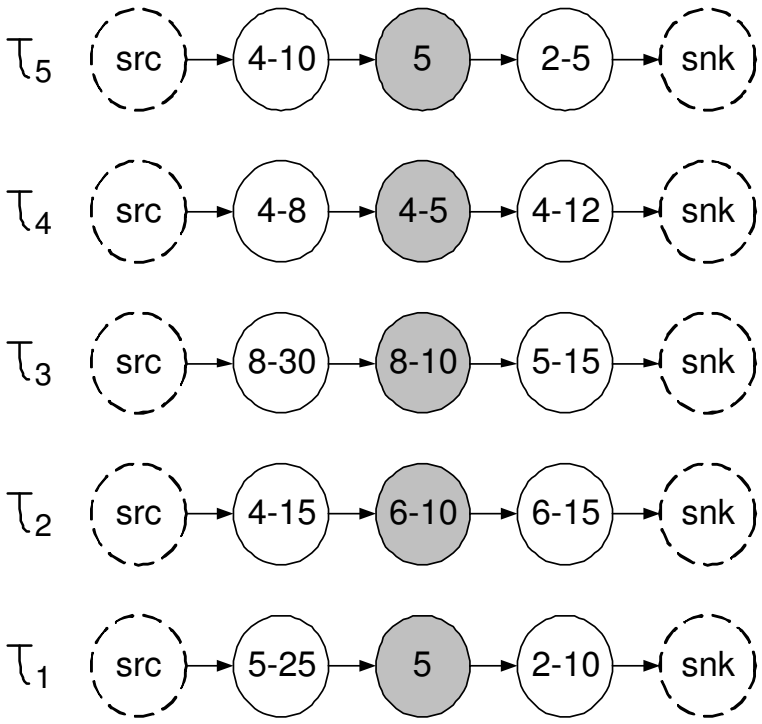

Figure 26: Respective process graphs for the set of linear mixed hardware/software processes of the example system of Table 6

process	priority	T
τ_5	5	50
τ_4	4	70
τ_3	3	300
τ_2	2	1000
τ_1	1	4000

Table 6: Scheduling parameters for the processes of Figure 26

process	worst-case synthetic execution distribution	worst-case synthetic jitter (A)
τ_5	[10, (5), 5, (30)]	0
τ_4	[12, (4), 8, (30)]	1
τ_3	[30, (8), 15, (140)]	2
τ_2	[15, (6), 15, (760)]	4
τ_1	(irrelevant)	(irrelevant)

Table 7: Worst-case synthetic distributions and jitters for processes of Table 6

process	priority	T	${}^{c/u}R$	${}^{c/h}R$	${}^{o}R$	${}^{\xi}R$
τ_5	5	50	20	20	20	20
τ_4	4	70	45	55	40	40
τ_3	3	300	245	210	160	160
τ_2	2	1000	890	370	240	240
τ_1	1	4000	2940	680	415	400

Table 8: Comparison of derived upper bounds on process WCRTs for the system of Table 6 under the various analytical approaches

1. Possibly, our basic approach is sufficiently accurate anyway, such that little room for improvement remains (and this is the reason that the synthetic approach offers only modest improvement for this specific example).

2. Possibly, in the general case, the less pessimistic a given analytical approach is, the more overwhelmingly complex (in the computational sense) and detailed a competing analytical approach would have to be to achieve even a modest further reduction in pessimism. Then, past a certain point, obtaining improved accuracy would be intractable. Given that determining the feasibility of a system is NP-hard as a problem [53], this appears likely.

In any case, so as to test either of the above conjectures, experimentation over a large set of systems would be necessary and, additionally, the exact process WCRTs would have to be available for comparisons. Since this is not possible, we proceed to test the degree of accuracy of our approaches for the specific system of our example.

Any observable (via simulation) response time for a process is a lower bound for the actual worst-case response time of the process (which, in the general case, is unknown to us). Thus, if R^{obs} is the maximum observed response time of a process and R is an upper bound, derived via analysis, for the worst-case response time of that process, then the overestimation of the worst-case response time is bounded by $R - R^{obs}$.

Figure 27 depicts two actually observable schedules for the system of our example. Figure 27(a) demonstrates that a response time of 40 is indeed observable for τ_4. Thus both the basic and the synthetic analysis derive the exact WCRT for the process. Figure 27(b) plots another observable schedule, under which the response times of τ_3, τ_2 and τ_1 are 138, 205 and 291 respectively. It is unclear if these are the exact WCRTs; we only inspected a few schedules. Even so, these observations permit the pessimism of the computed upper bounds on process WCRTs to be bounded (see Table 9).

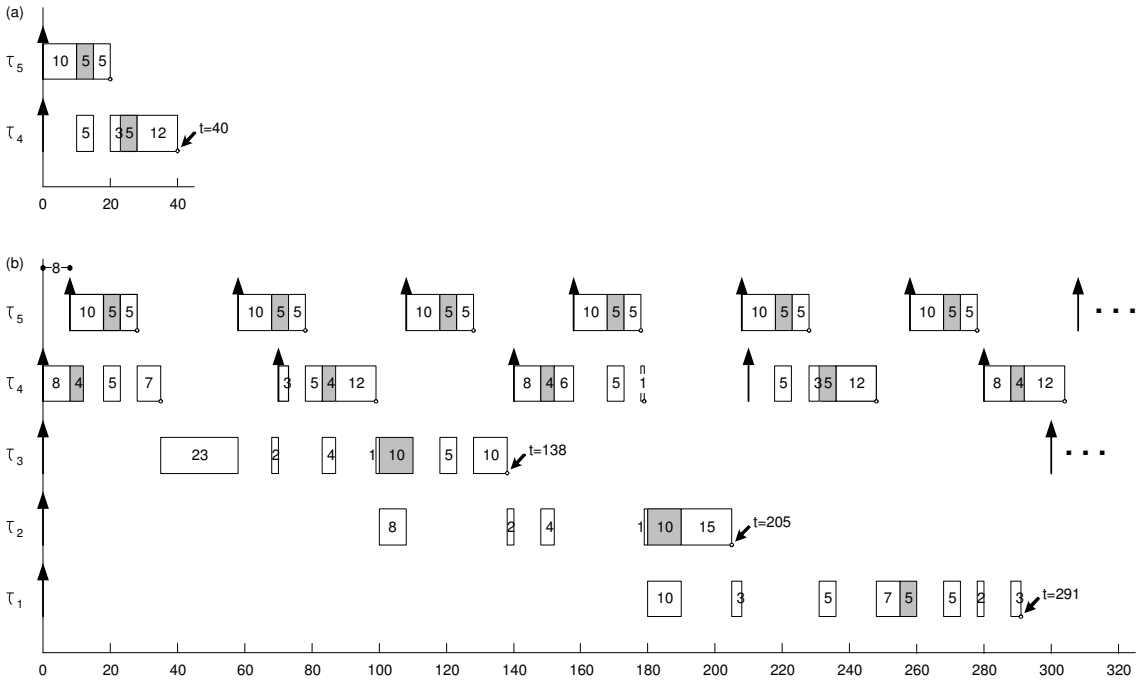

Figure 27: Two actually observable schedules for the system of Table 6, under different combinations of relative release offsets.

process	maximum observed response time	upper bound for pessimism in derived WCRT			
		uniprocessor	holistic	basic lim. parallel	synthetic
τ_5	20	+0%	+0%	+0%	+0%
τ_4	40	+13%	+38%	+0%	+0%
τ_3	138	+78%	+52%	+16%	+16%
τ_2	205	+334%	+80%	+17%	+17%
τ_1	291	+910%	+133%	+43%	+37%

Table 9: Upper bounds for the pessimism (in the derivation of upper bounds for WCRTs) of each analytical approach (when applied to the example of Table 6, given the observations of Figure 27)

It thus appears that, for the specific example, our approach is reasonably accurate. Indeed, it is likely that the pessimism is less than what is suggested in Table 6, if the actual WCRTs are found to be higher than those observed in Figure 27. In any case, the reduction in pessimism over the holistic and the uniprocessor analysis is considerable and more marked the lower the priority of the process in consideration.

While we offered just one example, we believe that it provides some useful indication of what may be expected in the general case.

5.7 Summary

Within this chapter, we formulated more accurate worst-case response time analysis for limited parallel systems (compared to the basic technique presented in Chapter 4). This analysis (termed *synthetic*) achieves improved accuracy in the analysis of systems where at least some of processes exhibit a linear structure (i.e their activations are always structured as a specific sequence of software code blocks and gaps) by reasoning about the patterns of software/hardware execution.

While some pessimism still persists, the improved accuracy achieved by our analysis in the characterisation of the worst-case timing behavior of limited parallel systems is notable. Previously, the only applicable approach to the analysis of limited parallel systems had been either the uniprocessor theory or the holistic approach, both of which were, as discussed, too pessimistic, when forced to this specific class of systems.

While the synthetic analysis is computationally more complex than our basic approach (which it supersedes), it is still tractable (thus deemed suitable for use within the inner loop of a codesign flow) if each process contains few gaps. Given that we expect hardware co-processors to be used for select complex, time-consuming functions, this is a reasonable assumption and consistent with current codesign practice.

For the purposes of achieving proper understanding of the timing behavior of limited

parallel systems, our worst-case response time analysis is complemented by our best-case response time analysis, which we will proceed to formulate in the next chapter.

6 Best-Case Response Time Analysis

Some of the same observations on which our synthetic worst-case analysis was based, will also form the basis for our best-case response time (BCRT) analysis, which we formulate within this chapter.

For many applications, knowledge of an upper bound for the response time of some processing activity does not suffice: knowledge of a respective lower bound is also necessary. Bate [16] describes, in the context of avionics, how a mechanical system controlled by software may become unstable (and incur damage) if output jitters exceed a certain threshold. Törngren provides the theoretical background in [82] for various categories of control systems and discusses the challenges in properly implementing them on computing systems. His conclusion is that jitters are the prime obstacle to the derivation of valid implementations of control algorithms.

Conversely then, so as to guarantee stability, a bound for jitter has to be found and compared to the threshold past which instability ensues. The tighter that such a bound is, the lower the possibility for a system to be erroneously deemed unstable. Additionally, a tight bound on that jitter would permit the designer to make the system operate closer the limit of its stability, for optimum performance [16].

Problems arising from jitter are not, however, limited to issues related to automatic control. It is possible that even the schedulability of a system may be jeopardised by jitter. Consider the example of Figure 28:

In this example, processes τ_a, τ_b, τ_c (in order of decreasing priority) execute on processor A under a fixed-priority scheme. Likewise for processes τ_d, τ_e, τ_f and processor B. All processes are periodic except for d, which is triggered whenever process c terminates. This means that the output jitter of τ_c determines the worst-case release jitter J_d of τ_d.

However, τ_d exerts interference on processes τ_e and τ_f. The worst-case response times of τ_e and τ_f increase as the worst-case release jitter of τ_d increases. If the jitter is

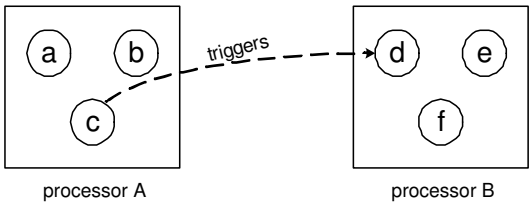

Figure 28: The output jitter of process c determines the release jitter of process d, which, in turn, impacts the schedulability of processes e and f.

past a certain threshold, it might then not be possible for all process deadlines to be met, in the worst case.

Of course, worst-case response time analysis may be deployed to determine if the processes assigned to processor B will meet their deadlines. One may use $J_d = R_c - \hat{R}_c$, where R_c is the an upper bound on the WCRT of τ_c and $\hat{R}_c$ respectively, on the BCRT of τ_c. This necessitates that analysis of the process set $\{\tau_a, \tau_b, \tau_c\}$ be carried out before the analysis of $\{\tau_d, \tau_e, \tau_f\}$ so as to obtain $\hat{R}_c$ and $\hat{R}_c$. However, the outcome of the analysis might still be pessimistic if these bounds are not tight enough.

It may well be that the actual output jitter of τ_c is such that τ_e and τ_f are still schedulable. However, since we will have to rely on upper and lower bounds (with some pessimism) for the response time of τ_c, the schedulability of τ_e and τ_f will be judged according to this pessimistic estimate of the output jitter of τ_c (see Figure 29). If the analysis used to derive R_c and $\hat{R}_c$ is not sufficiently accurate, the system may erroneously be deemed unschedulable. In the real world, this might in turn necessitate a modification of the design (for example, a costlier architecture, with an additional, dedicated processor for whichever process was erroneously found infeasible). We believe that this example sufficiently demonstrates the need for tight best-case analysis.

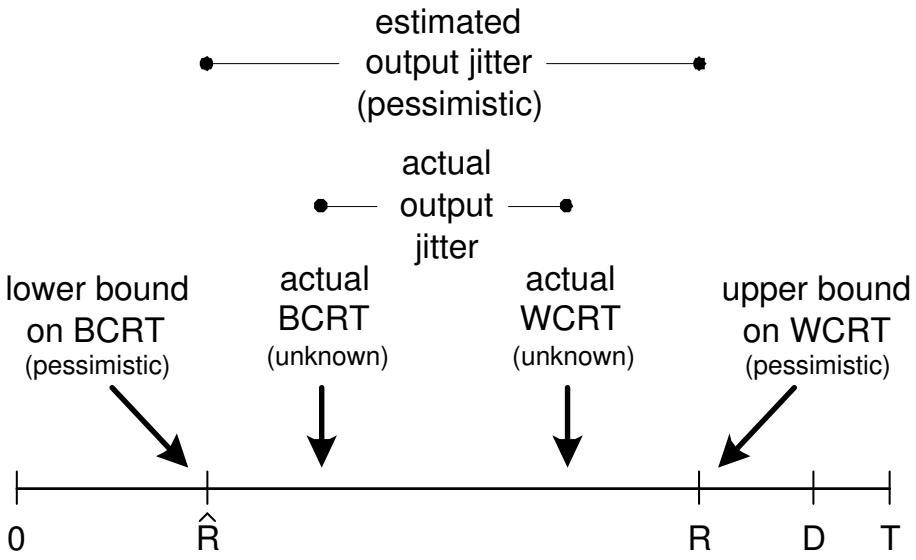

Figure 29: Tighter analysis reduces the degree of overestimation of the output jitter of a process.

6.1 Previously formulated approaches to BCRT analysis

In this section, we discuss existing approaches to BCRT analysis and identify some of their major shortcomings, which our approach has sought to address. Notable existing approaches are the one detailed by Palencia et al. in [63] and the subsequently published analysis of Redell et al. [73].

Until fairly recently, no attention was given to the problem of best-case response time analysis. Palencia et al. note in [63] that, until the time of writing, the response time of a process was treated in engineering practice as potentially being arbitrarily small, in the best case. Indeed, they present as a contribution in [63] the notion that, if a lower bound is known for the execution time of some process then that is also a lower bound for its best-case response time. The intuition is that, even if the process suffers no interference, it will still execute for at least that time – which we consider obvious. Indeed, Palencia et al. introduce this as a "trivial" approach to BCRT analysis before proceeding, within the same paper, with the formulation of a

more exact technique.

That technique mirrors the holistic WCRT analysis of Tindell et al [80] in that timing analysis is iteratively carried out separately for processing activities in each processing element; the bounds on response times derived by each iteration are then used to update release jitters for processes on other processors in the next iteration (until the solutions converge). As the terminology used by Palencia et al. is very different from ours, and so as not to create confusion, for any given concept we will be using our terminology (i.e. the one used within this document) to describe their approach.

Palencia et al. assume that processes are periodic and structured as linear transactions of code blocks (much as we do) - and that each code block is allocated to some processing unit in a distributed architecture. As this architectural model is more general than ours, their approach may be used for the analysis of limited parallel systems consisting of linear processes.

Palencia et al. identify that for a code block to encounter its BCRT, its execution time must be as short as possible (thus equal to its BCET) and the interference exerted upon it by higher-priority processes must be minimised. However, the approach used by them to derive a lower bound on that interference differs from our approach in one fundamental respect:

- Palencia et al. derive the lower bound on overall interference as the sum of the respective lower bounds for the interference exerted by each higher-priority *code block* individually.

- Our approach, by contrast, derives the lower bound on overall interference as the sum of the respective lower bounds for the interference exerted by each higher-priority *process*. This lower bound on the interference exerted by some higher-priority process is *not* computed as the sum of the respective lower bounds for the interference exerted individually by each one its constituent

code blocks - that would have been too pessimistic (i.e. resulting in lower bounds too low). Instead we use a more sophisticated best-case scenario.

The scenario, formulated in [63], for which interference is minimised, under Palencia et al., is based on the following reasoning (formulated here in the context of a limited parallel architecture, with code blocks either local or remote):

For the interference individually exerted by any higher-priority local block τ_{j_k} on some local block τ_i to be minimised, τ_{j_k} must terminate at the same instant that τ_i is released (i.e. $t = 0$) – as per Lemma 1 in [63]. With that as a given, the latest that the activation of τ_j to which that instance of τ_{j_j} belongs may have been released is at $t = -\hat{R}_{j_{1\to k}}$ – where $\hat{R}_{j_{1\to k}}$ is the best-case response time of the sequence of code blocks τ_{j_1} to τ_{j_k}. This "pushes" subsequent releases of τ_j as far to the right, along the time axis, as possible (thus, delaying associated interference as much as possible). The next release of τ_j will then occur at $T_j - \hat{R}_{j_{1\to k}}$ followed by one at $2T_j - \hat{R}_{j_{1\to k}}$ and so on. The latest then, that an activation of τ_{j_k} may be released, within each such activation of τ_j is $R_{j_{1\to k-1}}$ time units past the respective process release (where, if $k > 1$, $R_{j_{1\to k-1}}$ is the WCRT of the sequence of code blocks τ_{j_1} to $\tau_{j_{k-1}}$ and, if $k = 1$, it is zero). This is then equivalent to contributions of $\hat{C}_{j_k}$ time units of interference (i.e. the respective BCET) from each higher-priority local block τ_{j_k} at instants

$$t = zT_j - \hat{R}_{j_{1\to k}} + R_{j_{1\to k-1}}, \quad z = 1,\ 2,\ 3,\ ...$$

This permits the formulation of an equation which computes a lower bound on the BCRT of τ_{i_m}, which is:

$$\hat{R}_{i_m} = \hat{C}_i + \sum_{j \in hp(i)} \sum_k \left\lceil \frac{\hat{R}_{i_m} - (T_j + R_{j_{1\to k-1}} - \hat{R}_{j_{1\to k}})}{T_j} \right\rceil_0 \hat{C}_{j_k} \tag{19}$$

where the operator $\lceil \bullet \rceil_0$ is defined as:

$$\lceil u \rceil_0 = max(0, \lceil u \rceil)$$

The best-case scenario under Palencia et al. may be summarised (in the context of limited parallel systems) as:

A local block encounters its BCRT when its execution requirement is equal to its BCET and it is released at an instant when all higher-priority local blocks terminate, having encountered their respective BCRTs.

Even from its formulation, it becomes obvious how far from being exact this approach is. The above criterion would have the termination times of all higher-priority processes coincide, even those which belong to the same process – which is impossible. In fact, we observe that, if τ_{i_m} is released at the same instant that higher-priority block τ_{j_k} terminates, by the time of the next release of τ_{j_k}, if τ_{i_m} is still executing, it will have been preempted exactly once by every local block of τ_j other than τ_{j_k}: local blocks τ_{j_u}, $k < u$ belonging to the activation of τ_j containing the instance of τ_{j_k} which terminated at $t = 0$ and local blocks τ_{j_1} to $\tau_{j_{k-1}}$ from the next activation of τ_j.

Moreover, we note that, under any possible release offset, some local block τ_{i_m} may execute for U units at most, after being released, without being preempted by an activation of some local block belonging to τ_j, where U is either the maximum of the WCETs of the gaps of τ_j or $T_j - \hat{R}_j$ (whichever is greatest) [4].

These observations are later discussed in detail and form the foundation of our approach.

Redell et al [73] later came up with another BCRT scenario. They claimed that their approach was exact, unlike that of Palencia et al [63] which potentially under-

[4]This is true if the deadline of τ_j does not exceed its period. The approach of Palencia et al. is also applicable to systems with process deadlines possibly exceeding process periods. However, our approach does not cover such systems. We will be assuming that, for every process, $D < T$.

estimated the BCRTs of the processes in the system. Within the specific context that this claim was made in [73], the claim is true. However, that context is not the same as ours (i.e. a limited parallel model), as certain assumptions made in [73] no longer hold. We proceed to clarify the issue:

Unlike Palencia et al., who assume linear processes (i.e. code blocks sequentially activated) and a distributed architecture, Redell et al. assume a purely uniprocessor architecture. Using our terminology, they assume that processes are independent and consist of a single code block (hence there never exist any precedence constraints between code blocks and they may be arbitrarily phased). In that context, they derive lower bounds for process WCRTs which are exact, in the offset-agnostic sense (i.e. whichever the relative release offset between any two processes).

Still, the process model of Redell et al. accounts for potential jitter in the periodicity of processes. This then means that, if precedence constraints between code blocks are simply disregarded (which is a pessimistic assumption), systems analysable by the approach of Palencia et al. may then be analysed using the technique of Redell et al (and, among them, limited parallel systems). The only complication - hence our reference to jitters - is that the variation in the release time of local blocks (which depends on the completion time of the predecessor code block) will have to be specified as a jitter. This, in turn necessitates an iterative application of best- and worst-case analysis (with jitter values fed back in between iterations) until the solutions converge.

Surprisingly (given that the approach by Palencia et al. is aware of the precedence constraints), despite this pessimistic assumption, Redell et al. derive tighter bounds on BCRTs than Palencia et al. even for systems with precedence constraints between blocks (i.e. systems with linear processes). What must be noted, however is that these bounds are then simply valid but not exact (as we will proceed to show). The reason will become obvious upon formulation of the actual best-case scenario introduced in [73] - termed, by its authors, the “favourable instant”. Under Redell

et al (but, in the context of limited parallel systems and using our terminology):

A local block encounters its BCRT when its execution requirement is equal to its BCET and it terminates at the same instant that all higher-priority LBs are released, having encountered their respective worst-case jitter. Additionally, any activations of higher-priority LBs to have interfered with the LB in consideration must have executed for as long as their respective BCETs.

For a uniprocessor system, lower bounds on best-case response times derived under this scenario are then given by the following equation:

$$\hat{R}_i = \hat{C}_i + \sum_{j \in hp(i)} \left\lceil \frac{\hat{R}_i - J_j - T_j}{T_j} \right\rceil_0 \hat{C}_j \tag{20}$$

This assumes that processes are independent (i.e. that no precedence constraints exist). However, it is still possible to adapt this equation to a limited parallel system consisting of linear processes. Any LB τ_{j_k} within an interfering linear process may be (pessimistically) "translated" to one without precedence constraints, albeit with a worst-case release jitter of $J_{j_k} = R_{j_{1 \to k-1}} - \hat{R}_{j_{1 \to k-1}}$ (i.e. equal to the difference between the worst-case and the best-case completion time of its preceding code block relative to the release of the process it belongs to).

Then, a lower bound for the best-case response time of some local block τ_{i_m} is given by the equation

$$\hat{R}_{i_m} = \hat{C}_{i_m} + \sum_{j \in hp(i)} \sum_{k} \left\lceil \frac{\hat{R}_i - (T_j + R_{j_{1 \to k-1}} - \hat{R}_{j_{1 \to k-1}})}{T_j} \right\rceil_0 \hat{C}_{j_k} \tag{21}$$

At a first glance, by comparison of Equations 19 and 21, it would appear that, for the same system, bounds derived under Equation 19 would outperform (i.e. be numerically greater than or equal to) those derived under Equation 21. However, whereas Equation 19 is solved via a recurrence relation initiated by 0 (and increasing with every iteration until convergence), Equations 20 and 21 are instead solved via

a recurrence relation initiated by R_{i_m} (i.e. an upper bound for the **worst-case** response time of τ_{i_m}) and decreasing until convergence. In [73], it is proven how this approach outputs outperforms that of Palencia et al. In fact, for a uniprocessor system without precedence constraints and under the offset agnostic hypothesis, it outputs the exact BCRTs (otherwise it outputs valid respective lower bounds).

However, we proceed to identify some important shortcomings of this approach in the context of limited parallel systems made up of linear processes:

The approach of Redell et al., so as to be to applicable limited parallel systems with linear processes, completely disregards any precedence constraints between code blocks belonging to the same process. In doing so, its best-case scenario involves phasings which are impossible, thus underestimating interference. Specifically, under that scenario, local blocks belonging to the same higher-priority process, are to be released simultaneously, so that interference exerted by each one individually is minimised. However, this is not actually possible. If a phasing which minimises interference exerted by one specific local block is enforced, the phasing of every other local block belonging to the same higher-priority process will, as a result, have to be non-optimal. As with Palencia et al., Redell et al. assume minimal interference from each LB independently, instead of trying to bound interference on a per-process basis. Hence their best-case scenario is far more extreme than what can be actually be observed. And, as also observed for Palencia et al., the intervals during which any local block can be spared interference from a given higher-priority process are overestimated; the maximum such interval, for a given higher-priority process τ_j is either the maximum of the WCETs of the gaps of τ_j or $T_j - \hat{R}_j$ (whichever is greatest).

Thus, the accuracy of either of the two approaches examined, is (in the context of limited parallel systems) not satisfactory. This realisation directs us towards devising an alternative analytical approach, which we proceed to formulate.

6.2 Formulation of the problem in detail

We are first going to discuss the derivation of bounds for process BCRTs in uniprocessor systems (i.e. with no co-processors) because the findings will be relied upon for the subsequent formulation of BCRT analysis for limited parallel systems.

Consider thus a uniprocessor system with multiple, independent periodic processes, executing under a fixed priority scheme. We initially also assume that there are no shared resources, hence never any blocking.

Let τ_i be a process, bounds on whose response time we wish to calculate. For every higher-priority process τ_j, the following pattern would always be observed:

For an interval of variable duration R (ranging from $\hat{R}_j$, the BCET of τ_j, up to R_j, its WCRT) τ_j is either executing on the processor, or is preempted by processes of even higher priority. This interval is always followed by an interval of duration $T_j - R$ (thus ranging from $T_j - \hat{X}_j$ down to $T_j - X_j$ respectively) when τ_j is idle. This interval ends with the next release of τ_j and the cycle is repeated.

Requests by τ_i (and processes of even lower priority) for the processor do not affect the scheduling decisions for the set of higher-priority processes. τ_i may however suffer interference from those processes.

If there is no knowledge of the relative release phasings of the various processes in the system, the established analysis by Palencia has shown the following:

If each interfering activation of process τ_j is released as late as possible, relative to the release of τ_i (which has a minimal execution requirement, i.e. $\hat{C}_i$) and places as little demand as possible (i.e. $\hat{C}_j$) on the processor, the interference exerted by τ_j on τ_i is minimised. For the described phasing to occur, τ_i must be released at an instant when each interfering process τ_j has terminated, having encountered its respective BCRT, R_i. Intuitively, this maximises the idle time window, described earlier, before τ_j gets to request the processor again.

While this scenario may be observable for each interfering process individually, it cannot be observable for all of them at once (as at most one process may terminate on any given instant [5]. Nevertheless, this scenario (termed the "optimal instant" [63]) is a useful construct. By forcing it, for the purposes of analysis, on each $\tau_j \in hp(i)$, we derive a valid (if pessimistic) lower bound for the interference jointly from all $\tau_j \in hp(i)$ (hence also a lower bound for the BCRT of τ_i).

The "optimal instant" is depicted in Figure 31. (Compare with the classic "critical" instant of WCRT analysis in Figure 30.)

(We need not reiterate the best-case scenario of Redell et al. [73], because our contribution is actually based on the best-case scenario of Palencia et al.)

Consider now a system where processes are linear in structure and may issue remote operations on co-processors. Each process may be modelled as a linear graph where local and remote nodes appear interleaved (such as the one of Figure 15(a) in page 108). The blocks of code of any process are then always activated in the same order, within any activation of the process. This means that between successive releases of the same block of code (respectively, belonging to successive activations of the process it is a part of) all other code blocks belonging to the process will have been released exactly once (and will have terminated). We exploit this fact for our BCRT analysis, in a similar manner to how we did for the synthetic WCRT analysis.

Local blocks of code exert interference on local blocks belonging to lower-priority processes and suffer interference from (i.e. may be preempted by) local blocks belonging to higher-priority processes. Remote code blocks, by contrast, neither exert (as they do not cause preemption) nor suffer interference (as they may not be preempted).

Consider an entirely software-based (i.e. without gaps) process τ_i. The interfer-

[5]For a process to terminate on a given instant, it has to have been executing on the processor immediately before - and at most one process may be executing on the processor at a time.

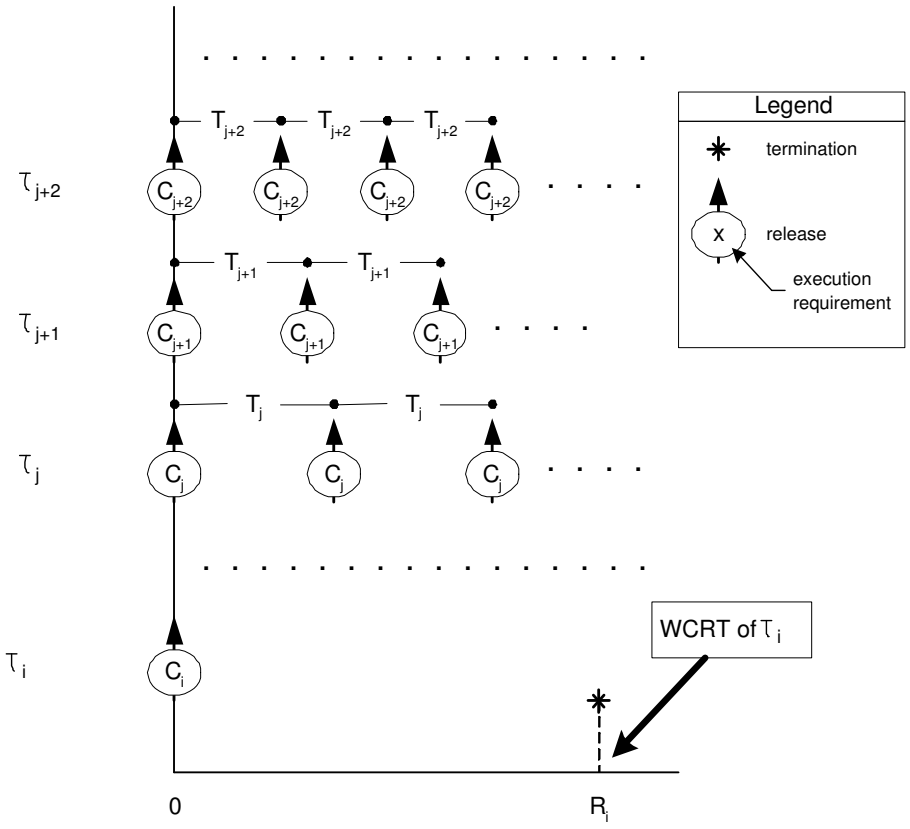

Figure 30: The worst-case scenario (critical instant) for uniprocessor systems

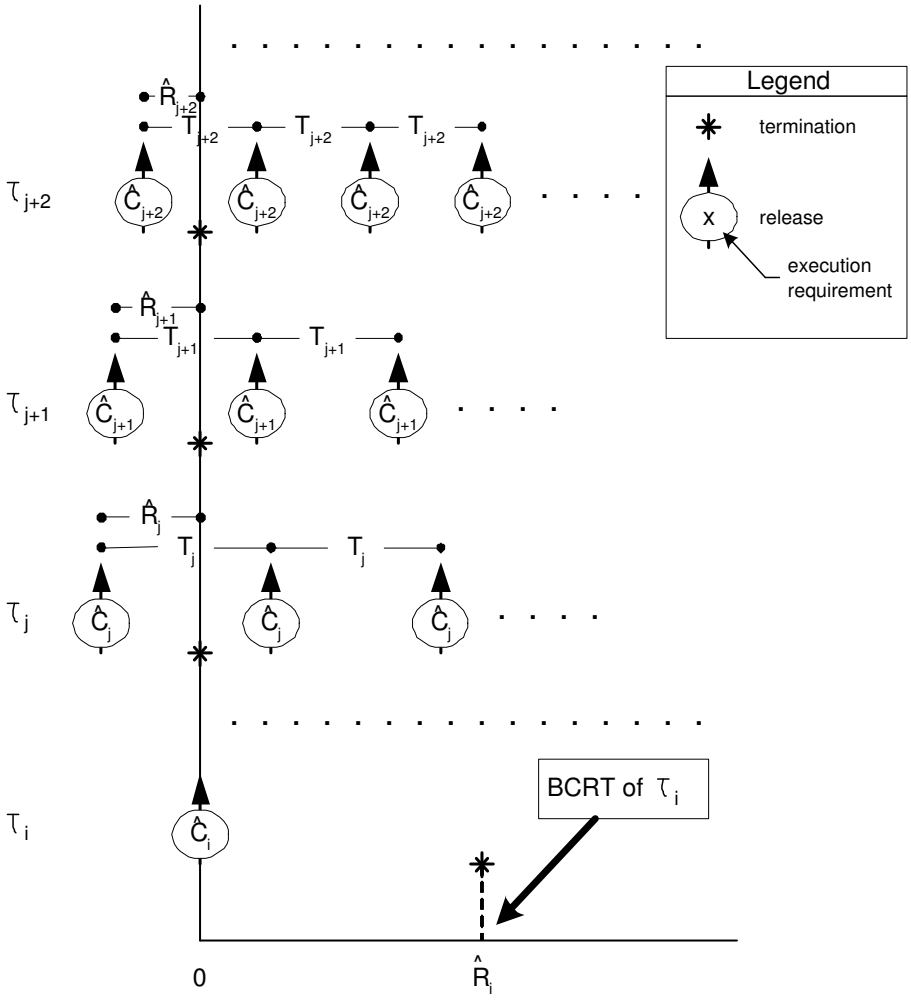

Figure 31: The best-case scenario (optimal instant) for uniprocessor systems under Palencia et al. [63]

ence suffered by some activation of τ_i due to activations of higher-priority processes released **after** τ_i is minimised when:

- **Condition 1**: Interfering (i.e. higher-priority) local-blocks execute for as short as possible (i.e. their respective BCET)
- **Condition 2**: Remote blocks belonging to higher-priority processes execute for as long as possible (i.e. their respective WCET). This spaces successive interfering LBs belonging to the same activation of a higher-priority process as far apart as possible within the execution distribution of that activation. This has the effect of delaying the next contribution of interference from said process as much as possible.

Palencia et al. [63] state as a corollary that for interference to be minimised, the release of the LB (using our terminology - not theirs) suffering the interference has to occur just after the termination of a LB of the interfering transaction. However, it is unclear which one, as all the corresponding phasings would have to be considered. We avoid the complexity resulting from having to consider different offsets by aiming for a lower bound on interference which is valid for all possible offsets.

We notice that for successive activations of τ_j for which conditions 1 and 2 hold, the time interval from the the termination of the last LB of one activation of τ_j and the release of the next activation of τ_j may not exceed

$$\hat{N}_j = T_j - \hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}G\ trunc})$$

where $\hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}\hat{G}\ trunc})$ is a lower bound on the response time of an activation of τ_j characterised by minimal LBs, maximal gaps and the final block of τ_j truncated if remote (as per Section 5 of Chapter 5). We use for such an execution distribution the symbol $\mathbb{K}_j^{\hat{X}G\ trunc}$. In turn, the BCRT for an activation of τ_j characterised by some execution distribution is to be derived as the sum of the BCRTs of the

individual code blocks in the distribution (with whatever execution requirements this distribution prescribes for them).

If condition 2 is relaxed, then the respective interval becomes

$$\hat{N}'_j = T_j - \hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}\hat{G}\ trunc})$$

where $\mathbb{K}_j^{\hat{X}\hat{G}\ trunc}$ is the execution distribution of τ_j characterised by minimal LBs and minimal gaps (and, if its last code block is remote, a truncation thereof).

Since $\mathbb{K}_j^{\hat{X}G\ trunc}$ and $\mathbb{K}_j^{\hat{X}\hat{G}\ trunc}$ differ only in the lengths they prescribe for the gaps and since each gap in the former may not be shorter than the respective gap in the latter:

$$\hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}G\ trunc}) \geq \hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}\hat{G}\ trunc})$$

It immediately follows that $\hat{N}_j < \hat{N}'_j$.

Considering the possibility that τ_j may start with a gap, the time interval from the termination of the last LB of an activation of τ_j characterised by $\mathbb{K}_j^{\hat{X}G\ trunc}/\mathbb{K}_j^{\hat{X}\hat{G}\ trunc}$ to the release of the first LB of the next activation of τ_j (removing any assumption regarding the distribution for that second activation) is then, respectively for each case:

$$\hat{\hat{N}}'_j = \begin{cases} \hat{N}'_j + G_{j_{first}} & \text{if } \tau_j \text{ starts with a gap} \\ & \text{whose WCET is } G_{j_{first}} \\ \hat{N}'_j & \text{otherwise} \end{cases}$$

$$\hat{\hat{N}}_j = \begin{cases} \hat{N}_j + G_{j_{first}} & \text{if } \tau_j \text{ starts with a gap} \\ & \text{whose WCET is } G_{j_{first}} \\ \\ \hat{N}_j & \text{otherwise} \end{cases}$$

Now consider the integer set ${}^{\hat{\xi}}\mathbf{X}_j$, of cardinality $|{}^{\hat{\xi}}\mathbf{X}_j|=\hat{N}(j)$, which consists of the BCETs of all LBs of τ_j. Consider also the integer set ${}^{\hat{\xi}}\mathbf{G}'_j$ which consists of $\hat{\hat{N}}'_j$ and WCETs of all the gaps of τ_j except the final one (if τ_j ends with a gap) and the first one (if it starts with a gap). Likewise, consider the integer set ${}^{\hat{\xi}}\mathbf{G}_j$ which consists of $\hat{\hat{N}}_j$ and the set WCETs of all the gaps of τ_j except the final one (if τ_j ends with a gap) and the first one (if it starts with a gap).

By necessity, $|{}^{\hat{\xi}}\mathbf{G}'_j| = |{}^{\hat{\xi}}\mathbf{G}_j| = |{}^{\hat{\xi}}\mathbf{X}_j| = \hat{N}(j)$

Each member of ${}^{\hat{\xi}}\mathbf{G}'_j$ corresponds to an upper bound for the maximum time interval between the termination of a given LB of τ_j and the release of the next LB of τ_j. Note that in the case of the last LB of τ_j, the next LB of τ_j to follow belongs to the next activation of τ_j. Similarly for the members of ${}^{\hat{\xi}}\mathbf{G}_j$, albeit with the added constraint that the activation of τ_j to which the terminating LB belongs must be following the execution distribution $\mathbb{K}_j^{\hat{X}G\ trunc}$.

Let ${}^{\hat{\xi}}x_j(1)$, ${}^{\hat{\xi}}x_j(2)$, ... , ${}^{\hat{\xi}}x_j(\hat{N}(j))$ be the sequence returning the members of ${}^{\hat{\xi}}\mathbf{X}_j$ in ascending order. Let ${}^{\hat{\xi}}g'_j(1)$, ${}^{\hat{\xi}}g'_j(2)$, ... , ${}^{\hat{\xi}}g'_j(\hat{N}(j))$ similarly be the sequence returning the members of ${}^{\hat{\xi}}\mathbf{G}'_j$ in descending order. Similarly, let ${}^{\hat{\xi}}g_j(1)$, ${}^{\hat{\xi}}g_j(2)$, ..., ${}^{\hat{\xi}}g_j(\hat{N}(j))$ be the sequence returning the members of ${}^{\hat{\xi}}\mathbf{G_j}$ in descending order.

Consider a software process τ_i (or equivalently a LB) of lower priority than some other process τ_j, released under some offset, relative to τ_j.

τ_i will be able to execute for no more than ${}^{\hat{\xi}}g'_j(1)$ time units without suffering a first preemption by τ_j. This holds whether there exist other higher-priority processes

besides τ_j or not. The interference exerted on τ_i by this initial preemption by τ_j may not be less than ${}^{\hat{\xi}}x_j(1)$. Once runnable again, τ_i will be able to execute for no more than ${}^{\hat{\xi}}g_j'(2)$ (i.e. the greatest member of ${}^{\hat{\xi}}\mathbf{G}_j'$ not already "expended") before again suffering a second preemption by τ_j. The interference exerted by τ_j on τ_i as a result of this preemption may not exceed ${}^{\hat{\xi}}g_j'(2)$ (i.e. the smallest member of ${}^{\hat{\xi}}\mathbf{X}_j$ not already "expended"). The procedure repeats itself until all members of ${}^{\hat{\xi}}\mathbf{G}_j'$ and ${}^{\hat{\xi}}\mathbf{X}_j$ have been expended.

Since we did not assume any particular release offset and since we did not place any restrictions regarding the interference exerted on τ_i by higher-priority processes other than τ_j, then the interference exerted by τ_j on τ_i up to this point may not be less than the the respective bound calculated if τ_j was the only higher-priority process and was released concurrently with τ_i and characterised by the execution distribution

$$[({}^{\hat{\xi}}g_j'(1)),\ {}^{\hat{\xi}}x_j(1),\ ({}^{\hat{\xi}}g_j'(2)),\ {}^{\hat{\xi}}x_j(2),\ ...,\ ({}^{\hat{\xi}}g_j'(\hat{N}(j))),\ {}^{\hat{\xi}}x_j(\hat{N}(j))]$$

Indeed, were τ_j to be characterised by any other execution distribution, any process τ_i not preempted more than once by any LB of τ_j (in other words, any process τ_i terminating before any LB of τ_j gets to be released for a second time), would suffer more interference than under the execution distribution constructed above. In simple terms, it would suffer more interference earlier. However, in the general case, a process may be preempted more that once by the same higher-priority LB, so we proceed to address that case as well:

Past the point where all members of ${}^{\hat{\xi}}\mathbf{G}_j'$ and ${}^{\hat{\xi}}\mathbf{X}_j$ have been "dealt", we proceed in a similar manner as before (replenishing the sets whenever they are fully expended) albeit "dealing" gaps from ${}^{\hat{\xi}}\mathbf{G}_j$ – not ${}^{\hat{\xi}}\mathbf{G}_j'$ as before. We justify this as follows:

Let ${}^{\hat{\xi}}g_j'(\alpha)$ be the member of ${}^{\hat{\xi}}\mathbf{G}_j'$ corresponding to $\hat{\hat{N}}_j'$.

As ${}^{\hat{\xi}}g_j'(\alpha) = \hat{\hat{N}}_j' \geq \hat{\hat{N}}_j$, it follows that ${}^{\hat{\xi}}g_j'(\beta) = {}^{\hat{\xi}}g_j(\beta) \bullet \forall\beta < \alpha$. Thus, up until

"dealing" the a^{th} gap of this new round, whether gaps are dealt from ${}^{\hat{\xi}}\mathbf{G}_j$ or ${}^{\hat{\xi}}\mathbf{G}'_j$, the sequence being constructed is the same. However, upon "dealing" of that a^{th} gap, the subsequence of $2\hat{N}(j)-1$ blocks preceding it consists of the members of ${}^{\hat{\xi}}\mathbf{X}_j$ (each appearing exactly once and corresponding to the respective LB of τ_j executing for its BCET) interleaved by those $\hat{N}(j)-1$ members of ${}^{\hat{\xi}}\mathbf{G}'_j$, ${}^{\hat{\xi}}\mathbf{G}_j$ which correspond to actual gaps of τ_j executing for their respective WCET and which are neither the first nor the last block of τ_j. We also note that the gap preceding this subsequence of $2\hat{N}(j)-1$ blocks is of length ${}^{\hat{\xi}}g'_j(\alpha) = \hat{\hat{N}}'_j$. We note two cases, depending on whether τ_j actually starts with a gap or not:

- If it does, then that idle interval of length $\hat{\hat{N}}'_j$ can only be observed at that position if the activation of τ_j whose LBs executed prior to that interval was characterised by the execution distribution $\mathbb{K}^{\hat{X}\hat{G}\ trunc}$ and the activation of τ_j whose LBs execute after that interval starts with a gap which executes for its respective WCET. If the release of that gap occurs at the time instant t^0, then the pattern of local and remote execution to the right of t^0 forms an execution distribution, say K^0_j for which the following property holds: The interleaved intervals of local and remote execution associated with it are the same as those in $\mathbb{K}^{\hat{X}\hat{G}\ trunc}$, albeit (possibly) in different placement (i.e. "shuffled ").

  Nevertheless, $\hat{R}(\tau_j, \mathbb{K}^0_j) = \hat{R}(\tau_j, \mathbb{K}^{\hat{X}G\ trunc}_j)$, as the offset agnostic BCRT of an execution distribution is derived as the sum of the offset agnostic BCRTs of its constituent blocks, local and remote (and addition is commutative). Thus, the a^{th} gap dealt during the second round will have to be of at most $T_j - \hat{R}(\tau_j, \mathbb{K}^{\hat{X}G\ trunc}_j) + G_{j_{first}}$ time units, because anything else would contradict the fact that τ_j is periodic. And $T_j - \hat{R}(\tau_j, \mathbb{K}^{\hat{X}G\ trunc}_j) + G_{j_{first}} = \hat{\hat{N}}_j = {}^{\hat{\xi}}g_j(\alpha)$

- If τ_j does not start with a gap, then the execution distribution K^0_j whose local and remote execution intervals are the same as those of $\mathbb{K}^{\hat{X}\hat{G}\ trunc}$ (albeit possibly reordered) is formed by exactly the above mentioned subsequence of

$2\hat{N}(j) - 1$ blocks. It then follows that the a^{th} gap dealt during the second round will have to be of at most $T_j - \hat{R}(\tau_j, \mathbb{K}_j^{\hat{X}G\ trunc}) = \hat{\hat{N}}(j) = {}^{\hat{\xi}}g_j(\alpha)$ time units.

The above reasoning applies for all subsequent dealings as well.

Interference by the above generated infinite sequence of interleaved gaps and LBs is already a lower bound for interference exerted by τ_j on τ_i. This interference would not in any case decrease if we only dealt gaps from ${}^{\hat{\xi}}\mathbf{G}_j$ but at the same time prepended the above infinite sequence by an additional $\hat{A}_j = {}^{\hat{\xi}}g_j'(\alpha) - {}^{\hat{\xi}}g_j(\alpha)$ units of processor idleness. (This would only result in shifting by $\hat{A}_j$ time units to the right all LBs left of the the a^{th} gap of the second round dealt. This does not increase the interference exerted.) This transformation would permit the formulation of the above infinite sequence as a periodic repetition of the execution distribution

$$[({}^{\hat{\xi}}g_j(1)), {}^{\hat{\xi}}x_j(1), ({}^{\hat{\xi}}g_j(2)), {}^{\hat{\xi}}x_j(2), ..., ({}^{\hat{\xi}}g_j(\hat{N}(j))), {}^{\hat{\xi}}x_j(\hat{N}_j)]$$

(which we call the *best-case synthetic distribution* for τ_j) occurring with a jitter of $\hat{A}_j$, which serves to delay all instances of it by the respective time interval. In other words, our best-case scenario calls for activations of every higher-priority process τ_j characterised by the respective synthetic best-case distribution (as defined above) and released at

$$t = \hat{A}_j,\ T_j + \hat{A}_j,\ 2T_j + \hat{A}_j,\ 3T_j + \hat{A}_j,\ ...$$

(where $t = 0$ corresponds to the release of τ_i).

6.3 Derivation of BCRT equations

The previous subsection contained the identification of a best-case scenario for limited parallel systems. Within this subsection, we cover how lower bounds on process

BCRTs are to be derived from that best-case scenario.

Consider again the entirely software-based (i.e. with no gaps) process τ_i, a bound on whose best-case response time we wish to find. For some process $\tau_j \in hp(i)$, its best-case synthetic distribution and its best-case synthetic jitter (i.e. $\hat{A}_j$) may be derived as previously detailed. Suppose that this best-case synthetic distribution is

$$[(^{\hat{\xi}}g_j(1)), ^{\hat{\xi}}x_j(1), (^{\hat{\xi}}g_j(2)), ^{\hat{\xi}}x_j(2), ..., (^{\hat{\xi}}g_j(\hat{N}(j))), ^{\hat{\xi}}x_j(\hat{N}_j)]$$

All this tells us is that, under the best-case scenario, τ_j will first request the processor $\hat{A}_j + ^{\hat{\xi}}g_j(1)$ time units after the release of τ_i. Since we assumed that τ_i is entirely software-based, if it has not completed its execution at that point (i.e. $t_1 = \hat{A}_j + ^{\hat{\xi}}g_j(1)$), it will have to be preempted. τ_j will be competing for the processor from t_1 onwards until it has accumulated $^{\hat{\xi}}x_j(1)$ time units of execution on it. As, during this interval, it may in turn be preempted by processes of even higher-priority than itself, the length of this interval will be some $r_{j_i} \geq ^{\hat{\xi}}x_j(1)$. Then, once instant $t_1 + r_{j_1}$ is reached, τ_j will issue a remote operation of length $^{\hat{\xi}}g_j(2)$. This means that, for the next $^{\hat{\xi}}g_j(2)$ time units, τ_i will be free from competition by τ_j for the processor. The next request by τ_j for the processor will occur at $t_2 = t_1 + r_{j_1} + ^{\hat{\xi}}g_j(1)$. That request will be for $^{\hat{\xi}}x_j(2)$ time units of execution on the processor. In turn (because of possible interference from processes of even higher priority), these additional $^{\hat{\xi}}x_j(2)$ units of execution time will be accumulated at $t = t_2 + r_{j_2}$, where $r_{j_2} \geq ^{\hat{\xi}}x_j(2)$. We proceed in a similar manner.

Thus, we have shown that requests by τ_j for the processor will occur:

- At $t_1 = ^{\hat{\xi}}g_j(1)$ for $^{\hat{\xi}}x_j(1)$ time units.
- Then, at $t_2 = t_1 + r_{j_1} + ^{\hat{\xi}}g_j(2)$ for $^{\hat{\xi}}x_j(2)$ time units.
- Then, at $t_3 = t_2 + r_{j_2} + ^{\hat{\xi}}g_j(3)$ for $^{\hat{\xi}}x_j(3)$ time units
- ...and so on.

By induction, for $1 < u \leq \hat{N}_j$, the k^{th} request for the processor (by the activation of τ_j characterised by $\hat{\mathbb{K}}_j$ and released at $t = \hat{A}_j$) will occur at

$$t_k = t_{u-1} + r_{j_{u-1}} + {}^{\hat{\xi}}g_j(k) = \hat{A}_j + {}^{\hat{\xi}}g_j(k) + \sum_{v=1}^{k-1}({}^{\hat{\xi}}g_j(v) + r_{j_v})$$

for ${}^{\hat{\xi}}x_j(k)$ time units, respectively.

Considering that releases of τ_j, in our best-case scenario, characterised by $\hat{\mathbb{K}}_j$, occur at $t = A_j + zT_j,\ \forall z = 0,\ 1,\ 2,\ ...$, the above expression may be generalised to

$$t_k^{(z)} = \begin{cases} zT_j + \hat{A}_j + {}^{\hat{\xi}}g_j(k) & \text{if } k = 1 \\ zT_j + \hat{A}_j + {}^{\hat{\xi}}g_j(k) + \sum_{v=1}^{k-1}({}^{\hat{\xi}}g_j(v) + r_{j_v}^{(z)}) & \text{if } 1 < k \leq \hat{N}_j \end{cases}$$

where the superscript in parentheses indicates that, since an interval $r_{j_{k-1}}$ also incorporates interference, it may vary for subsequent activations of τ_j.

Observing the above expression, we note that interference exerted by τ_j on τ_i would be minimised if intervals $r_{j_v}^{(z)}$ are maximised. Thus, we must derive upper bounds for the respective intervals. Recall what such an interval $r_{j_v}^{(z)}$ corresponds to:

It is the response time of a request by τ_j for the processor, for exactly ${}^{\hat{\xi}}x_j(v)$ time units. Thus, the upper bound that we seek is an upper bound on the WCRT of a local block of τ_j whose execution time is ${}^{\hat{\xi}}x_j(v)$ time units. This upper bound can be derived by use of our synthetic worst-case analysis as ${}^{\xi}R(\tau_j, {}^{\hat{\xi}}x_j(v))$.

We thus reiterate our findings:

Under our best-case scenario, the k^{th} $(1 \leq k \leq \hat{N}_j)$ request for the processor (by an activation of τ_j released at $t = \hat{A}_j + zT_j$ and characterised by $\hat{\mathbb{K}}_j$) will be for ${}^{\hat{\xi}}x_j(k)$ time units of execution and will occur at

$$t_k^{(z)} = \begin{cases} zT_j + \hat{A}_j + {}^{\hat{\xi}}g_j(k) \ \text{ if } k = 1 \\ zT_j + \hat{A}_j + {}^{\hat{\xi}}g_j(k) + \sum_{v=1}^{k-1} \left({}^{\hat{\xi}}g_j(v) + {}^{\xi}R\big(\tau_j, {}^{\hat{\xi}}x_j(v)\big)\right) \ \text{ if } 1 < k \leq \hat{N}_j \end{cases}$$

If we reformulate this expression so that times are expressed relative to each respective release of the process, as *offsets*, we then obtain:

$$ {}^{\hat{\xi}}O_{j_k} = \begin{cases} {}^{\hat{\xi}}g_j(k) \text{if } k = 1 \\ {}^{\hat{\xi}}g_j(k) + \sum_{v=1}^{k-1} \left({}^{\hat{\xi}}g_j(v) + {}^{\xi}R\big(\tau_j, {}^{\hat{\xi}}x_j(v)\big)\right) \ \text{ if } 1 < k \leq \hat{N}_j \end{cases} \tag{22}$$

Using these values, we are thus able to reach an equation which outputs a lower bound for the BCRT of τ_i:

$$ {}^{\hat{\xi}}\hat{R}_i = \hat{C}_i + \sum_{j \in hp(i)} \sum_{k=1}^{\hat{N}(j)} \left\lceil \frac{{}^{\hat{\xi}}\hat{R}_i - \hat{A}_j - {}^{\hat{\xi}}O_{j_k}}{T_j} \right\rceil_0 {}^{\hat{\xi}}x_j(k) \tag{23}$$

However note that, the above equation is valid only if (as per our initial assumption) τ_i is entirely software based (i.e. has no gaps). In the general case, where τ_i is structured as a linear transaction of interleaved local blocks and gaps, a lower bound on its BCRT may be computed as the sum of the respective lower bounds on the BCRTs of its constituent code blocks. These are to be computed separately, using the analysis we just provided.

For example, if some process τ_i is structured as

$$\tau_{i_1} \longrightarrow \tau_{i_2} \longrightarrow .... \longrightarrow \tau_{i_q}$$

a lower bound on its BCRT, by use of our synthetic best-case analysis, is computed as

$$ {}^{\hat{\xi}}\hat{R}_i^{split} = \sum_{m=1}^{q} {}^{\hat{\xi}}\hat{R}_{i_m}^{split}$$

We observe that, since gaps suffer no interference, the BCRT of a gap is equal to its BCET. Thus, Equation 23 may be updated to as

$$\hat{\xi}\hat{R}_i^{split} = \sum_{\substack{m=1 \\ \tau_{i_m}:\ local}}^{q} \hat{\xi}\hat{R}_{i_m} + \sum_{\substack{m=1 \\ \tau_{i_m}:\ remote}}^{q} \hat{C}_{i_m} \tag{24}$$

where, in turn, each term ${}^{\hat{\xi}}\hat{R}_{i_m}$ is computed as

$${}^{\hat{\xi}}\hat{R}_{i_m} = \hat{C}_{i_m} + \sum_{j \in hp(i)} \sum_{k=1}^{\hat{N}(j)} \left\lceil \frac{{}^{\hat{\xi}}\hat{R}_{i_m} - \hat{A}_j - {}^{\hat{\xi}}O_{j_k}}{T_j} \right\rceil_0 {}^{\hat{\xi}}x_j(k) \tag{25}$$

6.4 Towards even tighter bounds on process BCRTs

In our synthetic WCRT analysis, we detailed two approaches (termed "joint"/"split", respectively) which complement each other, in an effort to reduce pessimism. Likewise, for our BCRT analysis, having already formulated the respective "split" approach, we are going to introduce a "joint" approach as well. However, before doing so, we will demonstrate how not every property of our BCRT analysis is analogous to some property of the WCRT analysis. More specifically:

Perhaps, at this point, one could ask why Equation 23 might not, after all, be used to derive a lower bound for the BCRT of τ_i, even in the case that the latter contains gaps - after all, that would, at first, appear to be analogous with the so-called "joint" approach we of Chapter 5 Section 4, which was valid for our worst-case analysis. The answer is that such an approach would not output valid BCRTs – we demonstrate this by an example (see Figure 32).

Consider the simple process set of Figure 32(a) (we chose invariant block execution times for convenience). We want to compute a lower bound for the BCRT of τ_1. For this reason we construct the synthetic best-case distribution for τ_2 (see Figure 32(b)) and calculate the synthetic best-case jitter $\hat{A}_j$ and the offset for the only LB in that

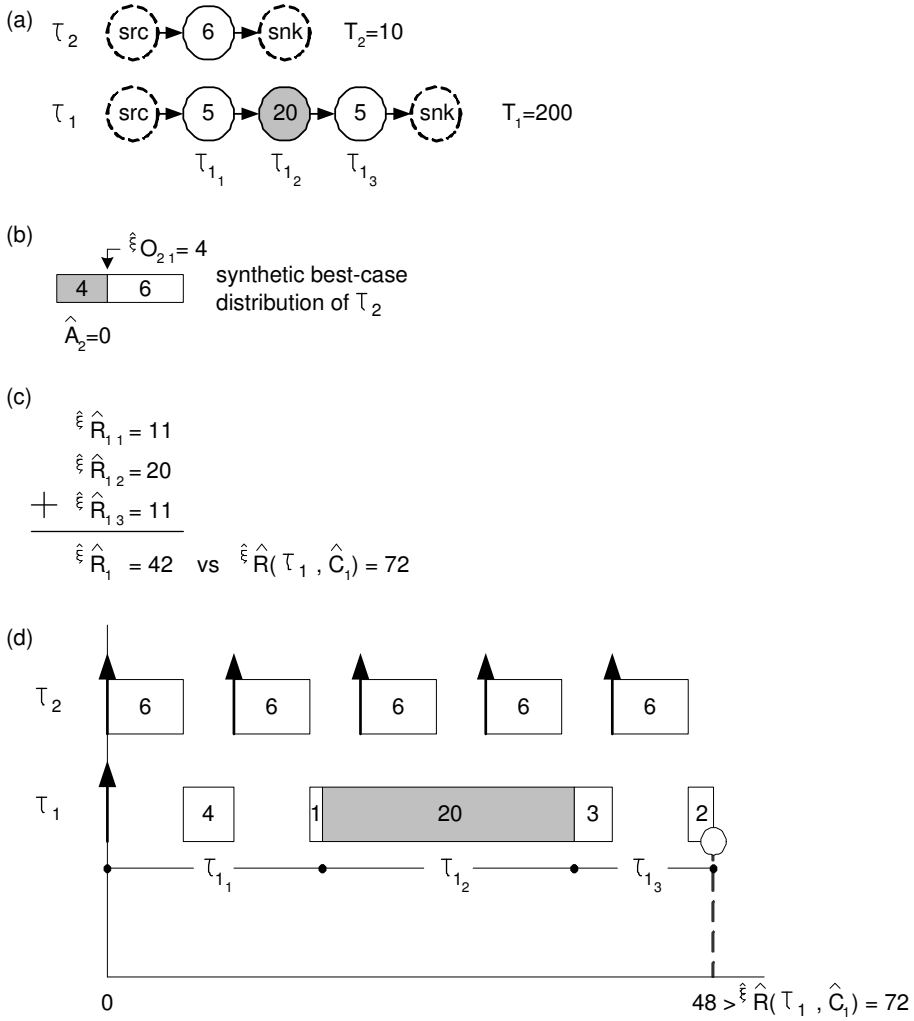

Figure 32: This example shows that Equation 23 is not applicable if τ_i may contain gaps

.

distribution. Using our approach, we then calculate lower bounds for the BCRTs of the individual code blocks of τ_1, which, if summed up, provide a lower bound of 42 time units for the BCRT of the process (see Figure 32(c)).

If, instead (erroneously, as we will show), one resorted to using Equation 23 and substituting the BCET for the whole process, the recurrence relation would converge at 72. This, however, would have only been a valid lower bound for the BCRT of τ_1, if τ_1 was a process executing entirely in software (i.e. with no gaps) – as we have noted, gaps are immune to interference but this is something that would not be accounted for (potentially resulting in an overestimation of interference, even for the best case).

To demonstrate this last issue, we arbitrarily choose a relative release offset for the two processes (in this case, the releases coincide) and simulate the scheduling decisions on the processor (see Figure 32). The response time of τ_1 in that case is 48 – hence 72, derived by the "naive" use of Equation 23 cannot be a lower bound for the BCRT of the process. Indeed, Equation 23 implicitly assumes that any execution of higher-priority processes, results in preemption for τ_i. But any execution in software of a higher-priority process occurring at a time when τ_i is executing remotely, will not exert interference. Hence, unless τ_i contains no gaps, the output of Equation 23 is not a lower bound for the interference suffered by τ_i.

Having disproven that "naive" attempt at a "joint" approach, we proceed to formulate the proper one. Again, we will use an example.

Consider a process set in a limited parallel system. Processes are linear and may have gaps. Let τ_i be the process a lower bound on whose BCRT we wish to derive. If the BCET and the WCET of each block belonging to a process of higher priority than τ_i is known, along with process periods, then the best-case and worst-case synthetic distribution of each such higher-priority process may be derived. Without loss of generality, assume that τ_i consists of a single local block of length $2x + 1$ (i.e. its execution distribution is $[2x+1]$) and that the synthetic distributions of the

higher-priority processes are such that

$$^{\hat{\xi}}\hat{R}(\tau_i, x) = x \tag{26}$$

and

$$^{\hat{\xi}}\hat{R}(\tau_i, 2x+1) = 2x+1+\delta > 2x+1 \tag{27}$$

What Equation 26 translates to, in simple terms, is that a (hypothetical) local block of length x belonging to τ_i would, in the best-case, suffer no interference whereas Equation 26 states that if the respective length is $2x+1$ (as is the case), it would, even in the best-case, suffer at least δ time units of interference.

Now assume that we instead had a τ_i whose execution distribution is $[x, (1), x]$, not $[2x+1]$). The overall execution time will remain the same, at $2x+1$ time units. However, even this small change (in, fact, the smallest possible) to the execution distribution for τ_i has a big impact to the derived lower bound on the BCRT of τ_i, if the "split" approach is used:

$$^{\hat{\xi}}\hat{R}_i^{split} = {}^{\hat{\xi}}\hat{R}(\tau_i, x) + 1 + {}^{\hat{\xi}}\hat{R}(\tau_i, x) = 2x+1 < 2x+1+\delta$$

We have thus come across a pathogenic corner case for our analysis. The derived lower bound is clearly an underestimation. Suppose that ϵ is the smallest integer such that

$$^{\hat{\xi}}\hat{R}(\tau_i, x+\epsilon) = x+\zeta > x \tag{28}$$

There are two possibilities: either $\epsilon = 1$ or $\epsilon > 1$.

- If $\epsilon = 1$, this then means that, under our best-case scenario, a release of a higher-priority block would then occur simultaneously with the release of the

gap of τ_i. But by having a gap of length 1 though, where there was previously (i.e. for the distribution $[2x+1]$) software, at most 1 time unit of interference is spared until the local block of τ_i which follows the gap is released.

- If, instead, $\epsilon > 1$, then whether the execution distribution of τ_i is changed from $[2x+1]$ to $[x, (1), x]$ or not has no effect on the interference suffered by τ_i under our best-case scenario.

These findings direct us towards formulating a "joint" best-case analytical approach which does not suffer from such corner cases and, if used in conjunction with the "split" approach already described, achieves tighter lower bounds on best-case interference.

Assume an activation of some process τ_i with an execution time of c time units, of which x in software and g remotely. If a lower bound on the cumulative time spent by higher-priority processes executing in software during the time interval $[0,\ c]$ is $\hat{W}^{(0)}$, then a lower bound for the interference suffered by τ_i during the same interval is $max(0, \hat{W}^{(0)} - \gamma^{(0)}$, where $\gamma^{(0)}$ is the cumulative time spent by τ_i executing remotely during the interval $[0,\ c]$ (as no more than $min(\gamma^{(0)}, \hat{W}^{(0)}$ time units of higher-priority execution may overlap with remote execution of τ_i). Since g is an upper bound for $\gamma^{(0)}$, a lower bound for the interference suffered by τ_i during $[0,\ c]$ is

$$\hat{I}^{(0)} = max(0, \hat{W}^{(0)} - g)$$

So as to find a lower bound for the interference suffered by an activation of τ_i, we would repeat the procedure for the interval $[0,\ c + \hat{I}^{(0)}]$ to find a new bound, $\hat{I}^{(1)}$ and would keep repeating, if necessary, until for some z: $\hat{I}^{(z+1)} = \hat{I}^{(z)}$. Then $c + \hat{I}^{(z)}$ is a lower bound for the BCRT of τ_i. What we just described with some abstraction is a recurrence relation, which is given more formally as:

$$^{\hat{\xi}}\hat{R}^{joint}(\tau_i, \mathbb{K}) = |\mathbb{K}| + max\left(\sum_{j \in hp(i)} \sum_{k=1}^{\hat{N}(j)} \left(\left\lceil \frac{^{\hat{\xi}}\hat{R}(\tau_i, \mathbb{K}) - \hat{A}_j - {}^{\hat{\xi}}O_{j_k}}{T_j} \right\rceil_0^{\hat{\xi}} x_j(k) \right) - \mathcal{G}(\mathbb{K}), 0 \right) \tag{29}$$

were $\mathbb{K}$ is the exact execution distribution of the activation of τ_i a lower bound on whose BCRT we wish to find and $\mathcal{G}(\mathbb{K})$ is a shorthand for the time units of remote execution in $\mathbb{K}$.

Now that we have both a "split" and "joint" approach to the derivation of lower bounds for process BCRTs, we may use the two approaches in conjunction, in a similar manner as in our previously formulated WCRT analysis. The algorithm accomplishing this is presented in pseudocode in Figure 33 and is a straightforward adaptation of the corresponding algorithm (see Figure 23 for pseudocode) described earlier for our WCRT analysis.

6.5 Evaluation

For the purposes of evaluating our BCRT analysis we will examine a simplified variant of the system first introduced in Table 6 (see page 151). This system, which consists of the two highest-priority processes from that original example and an entirely software-based lower-priority process, is depicted in Figure 34.

The execution time of τ_1, the lowest-priority process is left unspecified because it will be used as an additional parameter in our experimentation. We will be deriving lower bounds on the BCRT of τ_1 (under both the analysis of Redell et al. and the one we introduced) given different values for the BCET of τ_1.

Respective lower bounds for the BCRTs of τ_3, τ_2 under the two approaches happen to coincide:

$$^{Rdl}\hat{R}_3 = {}^{\hat{\xi}}\hat{R}_3 = \hat{C}_3 = 11$$

```
int bcrt_joint(int start, end)
{if ((start==end) && (is_remote_block(start)) //if examining a single RB
  return Ĉ[start]; //the gap length, as it never suffers interference
 else
 {int joint_length=0;
  for (int i=start;i<=end;i++)
   joint_length=joint_length+Ĉ[i];
  for (int i=start;i<=end;i++)
   if (is_remote_block(i))
    joint_gap_length=joint_gap_length+Ĉ[i];
  int joint_bound=r(current_process, joint_length, joint_gap_length);
  return joint_bound;
 }
}

int find_bound_for_bcrt(int start, int end)
{int bound=bcrt_joint(start, end);
 if (start!=end)
  for (int i=start; i<end; i++)
   bound=max(bound,
             find_bound_for_bcrt(start,i)+find_bound_for_bcrt(i+1,end));
 return bound;
}

int main() //entry point of the program
{return find_bound_for_bcrt(1, block_count_of(current_process);
}
```

Figure 33: The algorithm employing both the "split" and the "joint" approach for the derivation of lower bounds on process BCRTs (in C-like pseudocode)

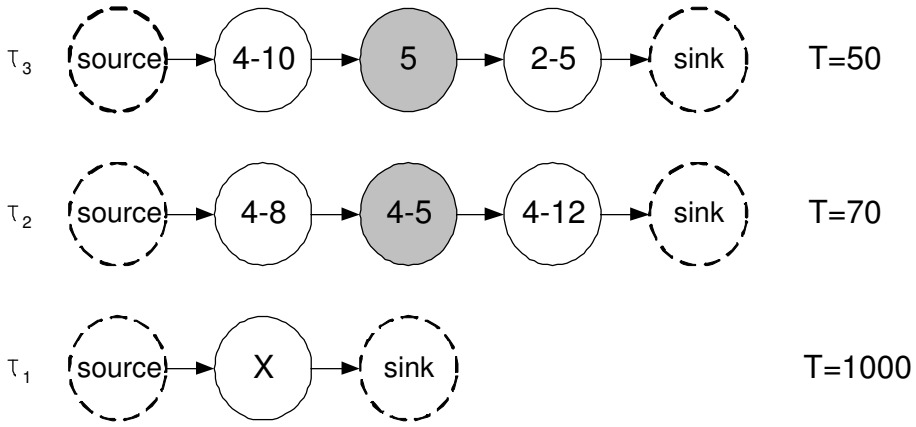

Figure 34: The process set used for the evaluation of our BCRT analysis

.

proc.	best-case synth. distribution	corresponding offsets	best-case synth. jitter ($\hat{A}$)
τ_3	[(39), 2, (5), 4]	${}^{\hat{\xi}}O_{3_2} = 39,\ {}^{\hat{\xi}}O_{3_4} = 44$	0
τ_2	[(57), 4, (5), 4]	${}^{\hat{\xi}}O_{2_2} = 57,\ {}^{\hat{\xi}}O_{2_4} = 76$	1
τ_1	(irrelevant)	(irrelevant)	(irrelevant)

Table 10: Best-case synthetic distributions, jitters and offsets for the processes of Figure 34

$$^{Rdl}\hat{R}_2 = {}^{\hat{\xi}}\hat{R}_2 = \hat{C}_2 = 12$$

We thus focus on the BCRT of τ_1. The best-case synthetic execution distributions, offsets and jitters derived for τ_2, τ_3 under our approach (which are used for the derivation of a lower bound on the BCRT of τ_1) are displayed on Table 10. For the approach of Redell et al., the jitter terms for each of the local code blocks τ_{3_1}, τ_{3_3}, τ_{2_1}, τ_{2_3} (which are treated as independent processes) are, respectively given in Table 11.

Derived lower bounds on the BCRT of τ_1 under the two approaches are represented

LB	period (T)	best-case execution time ($\hat{C}$)	worst-case jitter (J)
τ_{3_1}	50	4	0
τ_{3_3}	50	2	6
τ_{2_1}	70	4	0
τ_{2_3}	70	4	20

Table 11: Parameters used under the analysis of Redell et al. for the derivation of lower bounds on the BCRTs of the processes of Figure 34

in Figure 35 for values of $\hat{C}_1$ ranging from 40 to 100. For $39 \leq \hat{C}_1$, both approaches output a lower bound of $\hat{R}_1 = \hat{C}_1$ so we only display outputs for $\hat{C}_1 \geq 40$. We have the following observations:

- It is only when $\hat{C}_1$ reaches values of comparable magnitude to the shortest of the periods of the interfering processes that any of the two approaches is able to prove that some interference will be suffered by τ_1 even in the best case. Given however, that process BCETs are typically an order of magnitude or more smaller than the respective WCETs, such values for the BCET of a process would be atypically high (by the standards of most real-world systems). Whether, though, the BCRT analysis techniques are actually accurate either for small or for large values of $\hat{C}_1$, remains to be seen (via testing).

- Our approach outperforms that of Redell et al. by a modest margin. The numerical improvement ranges from 0 to 6 time units and the greatest relative improvement is for $\hat{C}_1 = 45$ (where the approach of Redell et al. outputs $\hat{R}_i = 45$, which is 12% less than 51, the output of our approach).

- The numerical improvement realised by our approach over that of Redell et al. does not match the degree of improvement earlier realised by Redell et al. over the "trivial" approach ($\hat{R}_1 = \hat{C}_1$). This reinforces our conjecture (made

earlier in the context of WCRT analysis) that each increase in the complexity of timing analysis only brings diminishing returns.

Having compared the analytical techniques to each other, we now attempt to derive bounds on the accuracy attained by each. For this reason, we construct actually observable schedules, for the system in consideration, under different execution requirements of τ_1 (see Figure 36. The response time observed for τ_1 in each case, is then an *upper bound* on what its BCRT may be (given the respective execution requirement for the process). We proceed to use these observed response times for the evaluation of the respective derived (via analysis) lower bounds on the BCRT of τ_1.

- From the schedule of Figure 36(a), we deduce that the BCRT of τ_1 is equal to its BCET if that BCET is 39 or less.
- Similarly, the BCRT of τ_1 may be no more than 48/58/66/89/96/127 when its BCET is, respectively, 46/52/56/45/80/107 (Figures 36(b) to (g)).

We compile this information into Table 12, which provides an overall comparison.

As can be seen, both our analysis and that of Redell et al. (which our approach outperforms) are very accurate; in many cases, they are even exact. However, even the trivial approach is reasonably accurate and especially so for small values of $\hat{C}_1$. Given though that process BCETs are in practice typically much shorter than the periods of interfering processes, it is likely that the improved accuracy made possible by our contribution will not be of much consequence in the average case. We note that this is not because our analysis fails to be exact (on the contrary, it provides unprecedented accuracy) but because, in the average case, it appears that even the trivial approach (which uses the BCET of a process as a lower bound on it BCRT), will be fairly accurate anyway. While we believe that our example provides a good demonstration of the potential of each approach, such a conclusion would

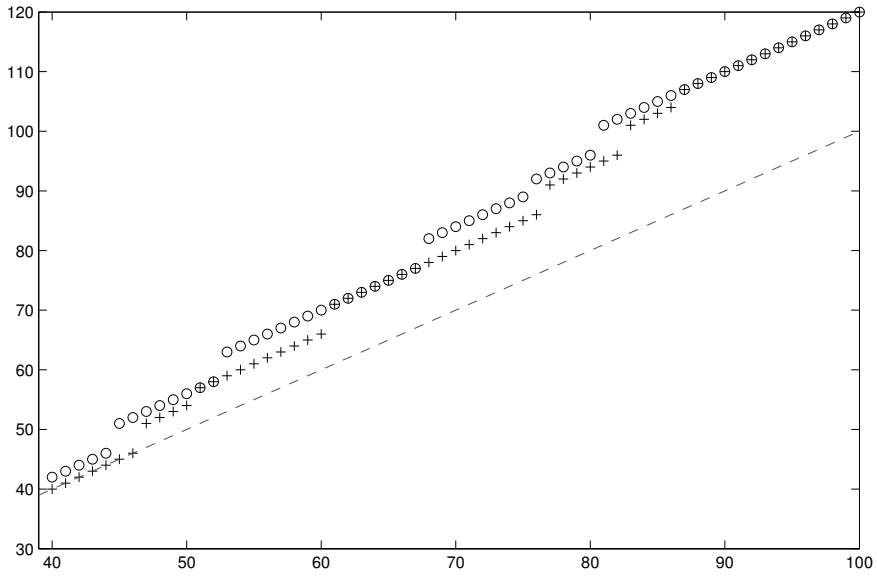

Figure 35: Lower bounds on the BCRT of τ_1 (as a function of its BCET) derived by our approach (o) and that of Redell et al. (+). The dashed line corresponds to $\hat{R}_1 = \hat{C}_1$ (the trivial approach).

$\hat{C}_1$	${}^{triv}\hat{R}_1$	${}^{Rdl}\hat{R}_1$	${}^{\hat{\xi}}\hat{R}_1$	observed
39	39 (exact)	39 (exact)	39 (exact)	39 (exact)
44	44 (-8% to -4% off)	44 (-8% to -4% off)	46 (-4% to -0% off)	48 (+0% to +4% off)
52	52 (-10% off)	58 (exact)	58 (exact)	58 (exact)
56	56 (-15% off)	62 (-6% off)	66 (exact)	66 (exact)
75	75 (-16% off)	85 (-4% off)	89 (exact)	89 (exact)
80	80 (-16% off)	86 (-10% to -11% off)	94 (-2% to -0% off)	96 (+0% to +2% off)
107	107 (-16% off)	127 (exact)	127 (exact)	127 (exact)

Table 12: Performance comparison by use of observed response times as a benchmark.

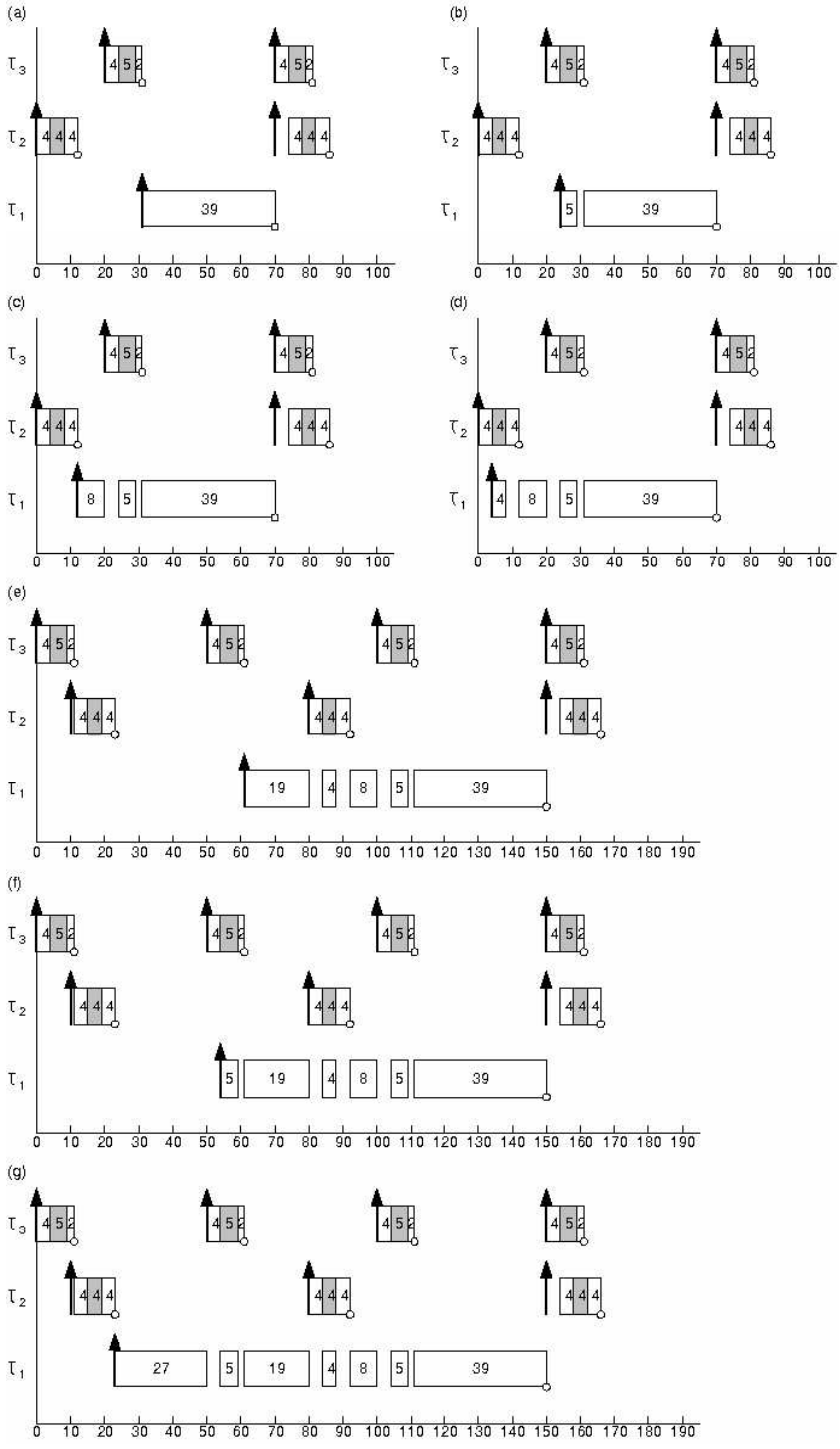

Figure 36: Observable response times for τ_1 for different values of $\hat{C}_1$.

also be consistent with the findings of Palencia et al [63] who note only marginal improvement from the use of their more detailed analysis over the use of the trivial approach.

As a final comment then, we make the following observation:

Since (in most practical contexts) for some process τ_i, $\hat{C}_i << C_i$, by necessity $\hat{R}_i << R_i$. Then, the pessimism in the derivation of an upper bound for the output jitter of τ_i (computed as $R_i - \hat{R}_i$) will be dominated by the pessimism in the derivation of R_i (especially given that BCRT analysis appears to be more accurate than WCRT analysis, in the general case).

Nevertheless, since our approach is tractable (and since any improvement in accuracy is, in principle, beneficial), we conclude that it a useful tool available to the designers of embedded real-time systems.

6.6 In the presence of blocking

Neither in the BCRT analysis of Palencia et al. [63] nor that of Redell et al. [73] is there any discussion of the effects that blocking in the system may have on the BCRT of a process. However, it cannot be deduced whether this was a conscious choice of the respective authors or due to an implicit assumption that the addition of shared resources, all other things being equal, could only result in increased response times (and thus, the approaches formulated could be put to use to derive valid lower bounds even for systems with shared resources). In the case of Palencia et al., our interpretation is the latter since, in the same paper [63], they advocate the joint use of WCRT and BCRT analysis so as to derive bounds on output jitters and the WCRT analysis employed for this purpose does account for blocking (hence they intend for their BCRT analysis to be applicable to systems with shared resources as well). In the case of Redell et al., we are less sure. Elsewhere, in the context of WCRT analysis, Redell does deal with the blocking effects of blocking (for example see [72,

74]) but in the paper on BCRT analysis, there is no stated assumption regarding the existence (or lack) of any shared resources. This ambiguity has prompted us to investigate the effects of blocking on the BCRT processes in a system.

We have established that the above conjecture (i.e. that the introduction of shared resources to a system, all other things remaining equal may not drive down process BCRTs) indeed holds, except under a pathogenic corner case, which we will proceed to characterise. This corner case is illustrated by the following example.

Consider a uniprocessor system consisting of two processes. The higher-priority process τ_2 has a period of $T_2 = 10$ and its execution requirement is 5 (for our convenience, let that be invariant, i.e. $C_2 = \hat{C}_2 = 5$). For the lower-priority process, respectively, $T_1 = 100$ and $C_1 = \hat{C}_1 = 6$. Nothing is known regarding the relative release offset of the two processes. However, (under either the best-case scenario of Palencia et al. or that of Redell et al.) it would be preempted at least once, even in the best case; both approaches output 11 as lower bound on its BCRT. Our approach (which in the context of uniprocessor systems reduces to that of Palencia et al.) outputs the same.

Now, let us introduce a single shared resource into the system. The critical section in τ_1 guarding that resource has a worst-case length of $B = 2$. Let us also assume that this critical section is located at the very end of the activation of τ_1 (i.e that τ_1 only releases the resource upon termination). Then, the following behavior (see Figure 37) may be observed:

τ_1 is released at $t = 0$, when τ_2 is idle. At $t = 4$ it proceeds to access the shared resource. At $t = 5$, τ_2 is released and immediately attempts to access the shared resource, which however is already in use by τ_1, hence τ_2 is blocked. τ_2 only gets to execute, after τ_1 terminates (because only then, in our carefully crafted example, does it release the resource). Thus, by maintaining exclusive access to the shared resource up until its termination, τ_1 manages to avoid interference from higher-priority process activations (i.e. the activation of τ_2) released after exclusive access

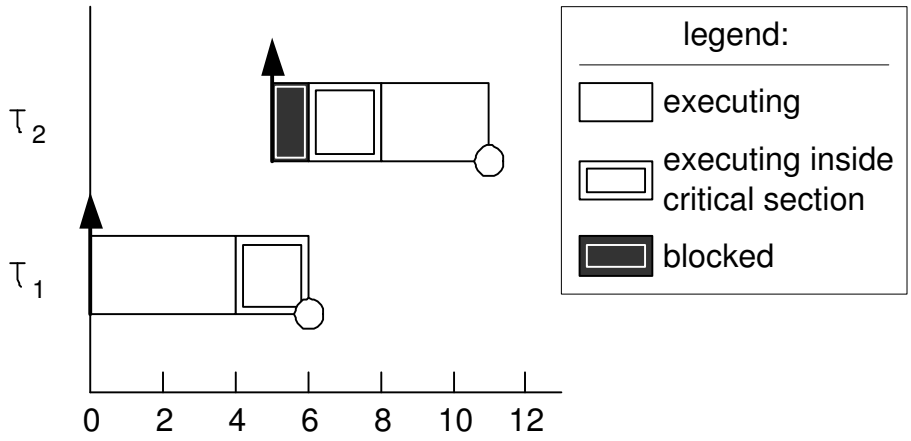

Figure 37: By maintaining exclusive access to the shared resource until the very end of its execution, τ_1 is able to evade any interference at all.

to the resource by τ_1 has been obtained. Note that the response time of τ_1 is, in this case, 6 time units, which is less than 11, the respective lower bound in the absence of blocking (under any of the approaches to BCRT analysis already discussed).

Suppose however, that τ_1 would not wait until the very end of its execution to release the the shared resource (in other words, that, upon releasing the shared resource, τ_1 would have at least one time unit of outstanding computation). Then, τ_1 would immediately be preempted by τ_2 and would only be able to resume execution after demand for the processor by τ_2 is satisfied.

Based on those observations, we generalise our findings (always in the context of a uniprocessor system):

- If a process may be executing with exclusive access to some resource (shared with some higher-priority process) during the last time unit of its execution, then its response time might potentially be less than the respective lower bound derived under either Palencia et al. or Redell et al. for a version of the system where (all other things being equal) there is no resource sharing.

- Otherwise, the analytical approaches mentioned derive valid lower bounds for the BCRT of the process. Although the respective intervals during which the process is preempted/executing may be affected by the blocking of higher-priority processes, the end result (i.e. the overall preemption suffered by the activation of the process under consideration) is the same as in the case that (all other things being equal) there is no resource sharing.

In the context of a limited parallel system, what applies above for a process (in the context of a uniprocessor system) would apply for each LB separately. That is, the analysis would only output a valid lower bound for the BCRT of the process if the remaining execution of each of its LBs, upon exit from a critical section from either process is at least one time unit. This applies to our BCRT analysis as well.

For those cases then where analysability is important, compliance to this requirement should thus be enforced by the designer.

6.7 Summary

Within this chapter we formulated our analytical approach to the best-case response time analysis of limited parallel systems. This analysis overcomes some of the pessimistic aspects of existing best-case analysis techniques (which were not originally formulated in terms of the limited parallel model) and achieves better accuracy. Our best-case response time analysis complements our work on worst-case response time analysis. When used in conjunction, they permit the derivation of bounds on process output jitters considerably more accurate than previously possible.

7 Priority Assignment

So far, we have focused on timing analysis for limited parallel systems scheduled under a fixed-priority scheme but have taken the assignment of priorities to processes for granted. We thus proceed to formulate a priority assignment algorithm, optimal in the presence of blocking, which uses the WCRT analysis formulated earlier within this thesis as a feasibility test. We originally formulated this algorithm in [19].

In fixed-priority preemptive scheduling [11], feasibility analysis determines if, for any given priority assignment to a process set, all process deadlines can be met. This is, of course, important to assert for hard real time systems. A priority assignment algorithm is *optimal* if it is guaranteed to output a feasible priority assignment if one exists.

For the discussion of priority assignment algorithms, we will initially limit our scope to purely uniprocessor systems (i.e. with no limited parallelism). This is both to facilitate the formulation of our contribution but also because our contribution is also of value in the context of uniprocessor systems. We will then discuss its applicability to limited parallel systems.

7.1 Background

A distinction exists between *synchronous* and *asynchronous* periodic systems. In the former, periodic process releases coincide once every system hyperperiod (defined as the least common multiple of process periods). Systems not meeting this criterion are termed asynchronous.

For synchronous systems and in the absence of blocking (caused by the existence of shared resources, access to which must be synchronised), the *Deadline Monotonic Priority Ordering* (abbreviated as DMPO) was proven optimal [53] if process deadlines do not exceed their periods. DMPO assigns priorities (from highest to

lowest) by order of increasing process deadline. An iterative algorithm, optimal for asynchronous systems alike, was later formulated by Audsley [7].

However, no known optimal priority assignment algorithm existed for process sets with shared resources (and thus blocking). Indeed the only known method for determining the existence of a feasible priority ordering for such a set of n processes, prior to our contribution [19], had been to (exhaustively) test all $n!$ possible orderings for feasibility. The optimality of DMPO for systems with synchronous systems with shared resources managed under the Priority Ceiling Protocol [68] had been de facto assumed, but was first proven in [19] as part of the work leading to this thesis.

We will proceed to reiterate here that original proof of the optimality of DMPO for synchronous systems under the PCP. We will then formulate another algorithm, optimal for asynchronous systems alike and (ultimately) optimal for limited parallel systems as well.

7.2 Terminology and assumptions

For a process set $\Delta = \{\tau_1,\ \tau_2,\ ..., \tau_n\}$ we assume that:

1. Process deadlines do not exceed respective process periods.
2. Processes may share resources, access to which is managed by the PCP.
3. Processes do not voluntarily suspend.

Λ^Δ is the set of all possible priority orderings over Δ. Each such ordering $P \in \Lambda^\Delta$ is a set of n tuples (τ_i, k), one per process, where k denotes the priority assigned to τ_i. Priorities are numbered 1 (lowest) to n (highest). Thus $|\Lambda^\Delta| = n!$ (the count of all possible permutations over $n!$ elements). For any $P \in \Lambda^\Delta$ we define:

- $pri(P, \tau_i)$ – gives the priority assigned to τ_i under P.

- $proc(P, i)$ – gives the process assigned to priority i under P.
- $procs(P, i, j)$ – gives the set of processes assigned to priorities $i..j$ under P.
- $hp(P, \tau_i)$ – gives the set of processes assigned higher priorities than $pri(P, \tau_i)$ under P.
- $feasible(P, i) \rightarrow \{true, false\}$ – $true$ if and only if τ_i is feasible under P.
- $feasible(P) \rightarrow \{true, false\}$ – $true$ if and only if $feasible(P, i) = true$, $\forall i \in \{1, .., n\}$.

7.3 Optimal priority assignment for synchronous systems with shared resources managed under the PCP

Theorem 3 *DMPO is optimal for synchronous systems with shared resources managed under the PCP.*

Proof: If Δ is schedulable under some priority ordering W, it suffices to show that it is also schedulable under DMPO. Let τ_y, τ_z be two processes in Δ with adjacent priorities under W such that $pri(y, W) = P_y > P_z = pri(z, W)$ and, for their respective deadlines, $D_y > D_z$.

We restate here, for convenience, some relevant equations: Upper bounds on process response times under the PCP are derived as

$$R_i = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + B_i \tag{30}$$

$$where \;\; B_i = max_{m=0}^{M}(usage(m, i)\ b(m)) \tag{31}$$

As defined in [77], b(m) is the worst-case length of the m^{th} critical section in the system whereas $usage(m, i) = 1$ if that critical section belongs to a process of priority

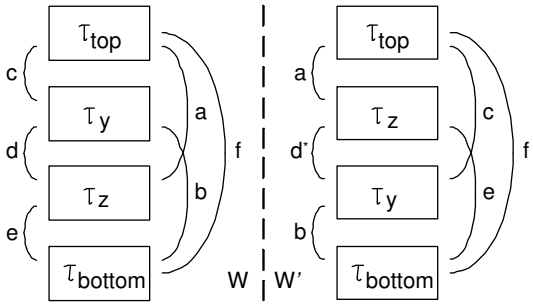

Figure 38: Effects of the priority swap on blocking [19]

greater than or equal to $pri(P, \tau_i)$ (but excluding τ_i) and the guarded resource is used by at least one process with priority less than $pri(P, \tau_i)$ and at least one process with priority greater or equal to $pri(P, \tau_i)$ (including τ_i). Otherwise $usage(m, i) = 0$.

Let $W^{'}$ be a priority ordering derived from W by swapping the priorities of τ_y, τ_z. Then $pri(z, W^{'}) = P_z^{'} = P_y > P_z = P_y^{'} = pri(y, W^{'})$. By inspection, the WCRTs of processes in $hp(W^{'}, j) = hp(W, i)$ and any with lower priorities than those of τ_y, τ_z are unaffected. Thus only the WCRTs of τ_y, τ_z need be examined.

Figure 38 highlights the relation between process priorities and blocking under both W and $W^{'}$. The process set is broken down in 4 priority bands, from highest to lowest priority. τ_{top} consists of processes with higher and τ_{bottom} of processes with lower priorities than both τ_y, τ_z. Each of τ_y, τ_z is the only process in its respective band. A letter denotes the longest critical section belonging to the lowest of the two bands joined by the respective arc and guarding a resource shared by at least one process in each band. Note that in the general case, d (the longest critical section of τ_z potentially causing τ_y to block under W) might differ in length from d^* (the longest critical section of τ_z potentially causing τ_y to block under $W^{'}$), hence the different symbol.

From Equation 31, respective blocking terms for τ_y, τ_z under W, $W^{'}$ are $B_y = max(a, b, d, f)$, $B_z = max(b, e, f)$ and $B_y^{'} = max(b, e, f)$, $B_z^{'} = max(c, d^*, e, f)$. The

blocking term of τ_z may only increase if $max(c, d^*) > max(b, e, f)$. Thus if any increase occurs, it is in either case due to blocking caused by τ_y (either executing c or d^*). Thus any increase in the blocking term is (in absolute value) less than the decrease in interference due to the priority swap (as τ_y interferes at least once with τ_z under W and $C_y \geq max(c, d)$. Thus $R_z^{'} \leq R_z < D_z$.

Thus it suffices to show that $R_y^{'} < D_y$. Since $B_y^{'} = max(b, e, f) = B_z = max(b, e, f)$, by inspection $R_y^{'} = R_z$. This is true because under both W, $W^{'}$ the same amount of computation has been completed with the same amount of interference from higher-priority processes and with the same time spent blocked by the respective process in consideration. τ_z is released only once during R_z under $W^{'}$ (as $R_z < T_z$) hence interferes only once with τ_y. It follows that $R_y^{'} = R_z \geq D_z < D_y$.

Thus both τ_y, τ_z remain feasible after a priority swap. By iteratively swapping adjacent processes which are in the "wrong" order under $DMPO$, the priority ordering can be transformed to $DMPO$ with schedulability preserved.
□

For the special case that process deadlines equal their periods, the same proof shows that Rate Monotonic [54] ordering is also optimal.

Note that the originally published proof [19] implicitly assumed that $d = d^*$ but the algorithm is optimal even in the general case (when, possibly, $d \neq d^*$).

7.4 Optimal priority assignment for asynchronous uniprocessor and limited parallel systems

In this section we will initially consider asynchronous uniprocessor systems and subsequently discuss the applicability of our model to limited parallel systems.

Upper bounds on blocking terms under the PCP described by Equation 31 also apply to asynchronous uniprocessor systems. However, there is no general equation which

may derive bounds on process WCRTs; these are determined by the construction a schedule as long as the least common multiple of the period of the process in consideration and the periods of any higher-priority processes. Nevertheless, for such systems, derived upper bounds on WCRTs depend on the set of higher-priority processes for the process in consideration (or, equivalently, lower-priority processes, as these sets are disjoint and one determines the other), not their relative priority orderings.

DMPO is not optimal for asynchronous systems even in the absence of blocking as the counter-example in [53] demonstrates. We introduce a priority assignment algorithm which is optimal for such systems when shared resources additionally exist (managed by the PCP). Proof and some complexity analysis then ensue.

Theorem 4 *The algorithm given as pseudocode in Figure 39 is optimal for asynchronous uniprocessor systems with shared resources managed by the PCP.*

Proof: Feasible priority orderings are identified by traversal of the permutation tree for the set of processes in consideration. An example of such a tree is that of Figure 40. Each path from the root to a leaf ($n!$ in total) corresponds to one of the possible distinct priority orderings: each non-root node corresponds to a process; its depth corresponds to its assigned priority.

A non-root node τ_i of depth β in the tree may form part of multiple distinct paths from the root to a leaf, each corresponding to a distinct possible priority ordering. Let $\Lambda^{'}$ be the set of such priority orderings. Then:

- $\forall P \in \Lambda^{'} \subset \Lambda^{\Delta} : \ pri(\tau_i) = \beta$
- $\forall P, P^{'} \in \Lambda^{'} \subset \Lambda^{\Delta}, P \neq P^{'} : \ hp(P, \tau_i) = hp(P^{'}, \tau_i)$

Combining these two properties with the fact that R_i is determined by hp(i):

```
int traverse_orderings(process_index t, int priority, stack
lower_priority_processes)
{if (test_process_feasibility(t, priority, lower_priority_processes))
 {push(process(t), copy_of(lower_priority_processes));
  if (priority+1==process_set_cardinality) //reached leaf
   print(lower_priority_processes); //outputs a feasible ordering
  else for (int j=1;j<=process_set_cardinality;j++)
   if (process(j) not in lower_priority_processes)
    traverse_orderings(j, priority+1, lower_prior_processes);
 }
}

int main() //entry point for the program
{for (i=1;i<=process_set_cardinality;i++)
  traverse_orderings(i, 1 ,new_empty_stack());
}
```

Figure 39: The branch-and-bound optimal algorithm in C-like pseudocode [19]

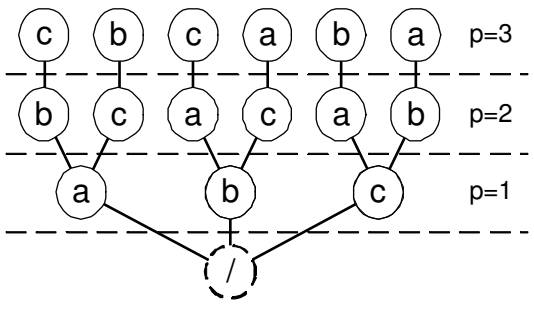

Figure 40: Permutation tree for $\{\tau_a, \tau_b, \tau_c\}$ [19]

$$\forall P, P^{'} \in \Lambda^{'} \subset \Lambda^{\Delta}, P \neq P^{'} : feasible(P, i) = true \Leftrightarrow feasible(P^{'}, i) = true$$

In other words:

- A single feasibility test determines the feasibility of the process τ_i under all the priority orderings that belong to $\Lambda^{'}$.
- If τ_i is found infeasible, then so is $\Lambda^{'}$ as a whole. Orderings corresponding to paths in the tree which share the node in consideration are infeasible.

This property allows us to disqualify multiple orderings as infeasible with a single test and is used in our algorithm:

The permutation tree is traversed depth-first and a process feasibility test is conducted per node. If it returns $false$, then the traversal of that subtree is aborted (the tree is pruned) as all paths (orderings) sharing the given node will be infeasible. A feasible ordering is identified whenever a leaf is reached which tests $feasible$. This *branch and bound* algorithm always terminates and finds all feasible orderings (or terminates on the first one found, if any suffices).
□

7.4.1 Complexity considerations

In the worst case, the whole tree is traversed. However, at most one process feasibility test is carried out per node. The number of nodes is $n(n-1)(n-2)..1 = n!$, thus the worst-case complexity is $O(En!)$, where E is the (non-polynomial [7]) complexity of the feasibility test.A brute force testing of all $n!$ possible priority orderings (at n process feasibility tests per ordering) has a complexity of $O(E(n! * n))$=O(E((n+1)!-n!)) so there is improvement even in the worst-case. However, since it is a branch and bound algorithm, we expect it to fare much better for most inputs. If there is

no blocking, no backtracking occurs if a feasible ordering exists, thus the algorithm reduces to the one in [7].

The original formulation of the algorithm calls for priority levels to be traversed from lowest to highest. However the algorithm works even if the traversal is from highest to lowest priority level. In that case, the permutation tree would have higher priority levels closer to the tree node (i.e. a greater node height would correspond to a higher corresponding priority level and the tree leaves would correspond to the lowest priority level).

The worst-case complexity is identical, whichever direction of traversal is chosen. However, we chose the bottom-up traversal as default because it is likelier to lower the average complexity: Given that any process is more likely to be unschedulable at the lowest priority level than at the highest, more prunings occur earlier under this scheme than under the inverse one.

7.4.2 Applicability to limited parallel systems

If processes in a limited parallel system are analysed according to the simple approach (i.e. the one described in Section 4 of Chapter 4) then the algorithm is directly applicable and (exactly as formulated) optimal. If however, the analysis is carried out according to the synthetic approach, then the permutation tree will have to be constructed with the highest-priority levels closer to the root (and traversed accordingly). This is the inverse of what was specified in the original formulation. The reason is that, for the derivation of an upper bound to the worst-case response time of any process, the synthetic worst-case execution distribution of each interfering process has to be first constructed. In turn, the construction of the synthetic worst-case distribution of any process requires knowledge (via prior calculation) of an upper bound to its respective worst-case response time. This, however, cannot be known if processes are analysed in order of ascending priority. The issue is resolved

if the priority levels are traversed from highest to lowest.

Since our WCRT analysis (which is used for the feasibility test) is offset-agnostic, the priority assignment algorithm is optimal in the offset-agnostic sense. In other words, the algorithm is guaranteed to find a priority ordering given which the system is feasible under any possible combination of relative process release offsets, if such an ordering exists.

7.5 Summary

Within this chapter, we presented our contributions to the area of priority assignment. Some of those contributions were not directly related to limited parallel systems. We proved the optimality of the Deadline Monotonic Priority Ordering scheme for synchronous uniprocessor systems with shared resources managed under the PCP. Subsequently, we formulated a branch and bound algorithm which achieves optimal priority assignment for asynchronous uniprocessor systems alike (again, when shared resources are managed by the PCP). However, we then showed that the same algorithm may be used for priority assignment in limited parallel systems (using our WCRT analysis as the feasibility test). In that context, our algorithm is optimal in the offset-agnostic sense.

8 Conclusion

Within this chapter, we are going to provide a summary of each of our contributions and highlight how these, in conjunction, validate our main hypothesis. Subsequently some possible directions for future work are identified.

8.1 Summary of contributions

In Chapter 1, through our survey of approaches to the hardware/software codesign of embedded real-time systems, we identified timing analysis as an area of concern within current codesign practice. In particular, we noted the following:

Timing analysis was usually resorted to at a late stage in the codesign process, so as to validate the behavior (with respect to its timing constraints) of a largely finalised candidate design. In our understanding, timing analysis should instead be continuously performed throughout the codesign process so as to guide the flow towards modifications in the right direction upon the candidate design. However, this in turn necessitates that the analysis employed then not only be sufficiently accurate but also fast enough for integration into the inner loop of a codesign flow (without considerably slowing it down).

In Chapter 2, with our review of timing analysis techniques, we noted that the techniques available for the analysis of architectures with co-processors where rather too pessimistic. In our understanding, this was because these techniques were not originally conceived with the characteristics of mixed hardware/software implementations in mind (such as the potential for parallelism). As a result, when put to use for the analysis of such systems, they do not perform as well as in their original context. Even an architecture consisting of a single general-purpose processor and multiple application specific hardware co-processors (the most studied one in codesign) is poorly addressed by established timing analysis if the various processing

units may operate in parallel.

These observations served as motivation for the formulation of our contributions.

In Chapter 3 we defined in detail the *limited parallel model* and its underlying assumptions. This computational model is based on the above discussed simple architecture. System functionality is structured in terms of mixed software/hardware processes. These processes are scheduled according to a fixed priority scheme on the processor but may also issue operations implemented in hardware. During such operations, the process in consideration notionally migrates to execution in hardware, relinquishing the software processor to other processes competing for it. A process executing in software may then be advancing in computation in parallel with multiple other processes, each of which is executing on some co-processor.

Chapter 4 features our basic *worst-case* response time analysis for the limited parallel model, which is based on the observation that execution in hardware does not contribute to interference upon lower priority processes. Our analysis is then able to compute overall interference only as a function of the execution in software of higher-priority processes while accounting for the fact that the location of software and hardware computation, within the activation of any process, may vary. Within the same chapter we also quantify the effects of blocking, if any shared resources in the system are managed by the Priority Ceiling Protocol [69] (as formulated in the context of uniprocessor systems). We show how this basic analysis for the limited parallel model outperforms the established uniprocessor analysis at no added complexity.

In Chapter 5, we formulated our *synthetic* worst-case response time analysis, a more detailed (and more accurate) analytical technique, evolved from our basic approach. As long as at least a subset of the mixed software/hardware processes in a system exhibit a certain linearity in structure, this linearity is exploited to reduce the overestimation in both the interference exerted and the interference suffered by such a linear process. We were able to show that the synthetic analysis will al-

ways outperform the holistic analysis [80] (a technique conceived for the analysis of distributed systems) in the context of limited parallel systems. The synthetic analysis is computationally considerably more complex than its basic counterpart but still tractable under our assumption that the partitioning of process code into software and hardware operations is coarse grained. This assumption is consistent with current engineering practice and enforceable during codesign. Thus, the synthetic approach is suitable for use with codesign.

In Chapter 6 we introduced *best-case* timing analysis for limited parallel systems. This analysis was formulated by analogy to the synthetic worst-case response time analysis. We discussed its merits relative to approaches originally formulated in the context of uniprocessor systems (namely those by Palencia et al. [63] and Redell et al. [73]) but also reached the conclusion that the pessimism in the derivation of upper bounds on the worst-case process output jitter is dominated by the pessimism in the worst-case (not the best-case) response time analysis.

In Chapter 7 we addressed the issue of priority assignment in the context of limited parallel systems. The timing analysis techniques formulated as part of our previous contributions had taken the priority assignment for granted. However, in this chapter, we introduce an optimal offset-agnostic priority assignment algorithm for limited parallel systems with shared resources managed under the Priority Ceiling Protocol. This algorithm belongs to the class of branch and bound algorithms and uses our worst-case response time analysis as the feasibility test. As a tangential contribution, we also prove the optimality of the Deadline Monotonic Priority Ordering [53] for uniprocessor systems with shared resources managed under the Priority Ceiling Protocol.

At this point we restate our main hypothesis:

Static timing analysis for both the worst and the best case can accurately characterise the timing behavior of limited parallel systems.

Given that as part of contributions we delivered both worst-case and best-case analysis targeted at limited parallel systems, the above hypothesis may in turn be tested by evaluation of said analysis techniques.

Our basic worst-case analysis technique has the same complexity as uniprocessor analysis yet considerably outperforms the later in the context of limited parallel systems. In the contrived case that no hardware operations exist, this analysis reduces to the uniprocessor one. However, both the the uniprocessor analysis and the holistic analysis may be used to analyse limited parallel systems and one does not consistently outperform the other, as there exist corner cases where each fares badly. However, our more detailed technique, the synthetic analysis, manages to always outperform both and is largely free of such idiosyncrasies.

One could argue that our analysis is still pessimistic, as it assumes the worst regarding the combination of relative process release phasings. We contend that, on the contrary, such offset-agnosticism was purposely designed into the analysis, given that limited parallel systems are likely to contain sporadic processes, offsets for which may not be relied upon and which invalidate any reasoning regarding offset combinations of the remaining processes. The worst case is *not* for a given combination of relative process release offsets, it is over *all* such combinations. This makes the timing behavior of the system resilient to change. Our analysis provides an accurate estimate of that offset-agnostic worst case, while at the same time remaining tractable (under our stated assumptions), thus suitable for use within codesign.

Regarding the best-case analysis of limited parallel systems, we have shown how our technique outperforms the earlier established best-case response time analysis techniques. Given that these approaches appear to have been fairly accurate anyway, for typical inputs, we can state with confidence that our analysis is accurate in the characterisation of the best-case behavior in limited parallel systems. Equipped with accurate best-case and worst-case timing analysis, one may then in turn derive accurate bounds on process output jitters.

Finally, one could argue that the value of our analysis would have been diminished if it was to be applied on systems with a suboptimal assignment of priorities to processes. We addressed this concern by formulating an optimal offset-agnostic priority assignment algorithm for limited parallel systems. This algorithm uses our worst-case timing analysis for the limited parallel model as the feasibility test (which is fairly accurate). While its worst-case complexity is exponential, it is a branch and bound algorithm, which ensures that its average complexity for typical inputs will be tractable.

Thus all our of contributions combine into a coherent analytical approach which accurately characterises the timing behavior of limited parallel systems.

8.2 Future work

We identify the following directions for future work which could enhance the value of our contributions:

- Throughout this thesis, we have assumed that there is no contention for hardware co-processors (as each of those is monopolised by a single process). However it would be useful to the system designer if co-processors could potentially be shared by two or more processes, so as to avoid hardware replication. Our model does not currently permit that. Given that hardware is not preemptible, some mechanism must control access to shared co-processors. Perhaps work in this area could be combined with other improvements to the original Priority Ceiling Protocol (which would account for the limited parallelism and further limit any pessimism), as a coherent future contribution.

- Another assumption throughout the thesis was that of a single general-purpose processor, whereas co-processors could be multiple. Yet there is no fundamental reason why the number of processors should be limited to one. Thus multiprocessor extensions of the limited parallel model and its associated analysis

are one possible direction for future work. Some of preliminary work in that direction is presented as an appendix to this thesis. However, any multiprocessor extension of our work will require an appropriate resource management scheme which will bound blocking suffered per process and prevent deadlock (again, conceived with limited parallelism in mind, so as not to be unnecessarily pessimistic). We expect that such a scheme would be based as well on the concept of Priority Inheritance. Some discussion of the challenges related to the design of such a protocol is also provided in the Appendix.

- Finally, our timing analysis has been conceived with hardware/software codesign in mind. We have shown its suitability for integration into a codesign loop. We would like to see that happen, via the development of a proof-of-concept codesign flow, implementing our timing analysis and employing it in the manner that we have been advocating.

A Appendix: Multiprocessor Architectures with Limited Parallelism

So far we have only considered architectures with a single general-purpose instruction set processor and multiple application-specific co-processors. However, variants of the above architecture, with multiple general-purpose processors are possible. The extension of the limited parallel model to multiprocessors has been identified within this thesis as a direction for future work. Within this appendix, we briefly discuss some preliminary activity towards that direction.

We examine a variant of the architecture assumed throughout the thesis, where the single instruction set processor is replaced by an array of N identical processors, which execute in parallel and have common address space and a common scheduling queue. Processes may use any processor to execute in software and may even migrate to a different processor to execute. Figure 41 depicts such an architecture, which may equivalently be described as a Symmetric Multiprocessor (SMP) architecture augmented by application-specific co-processors. The N highest-priority processes among those competing for execution in software are executing on the array any given instant. Other processes may be executing in hardware in parallel. Consider the example of Figure 42, which simulates the scheduling decisions in a system based on such a multiprocessor architecture with co-processors.

In that example, there are 5 processes in the system (τ_A to τ_E, by order of increasing priority). The processor array consists of two processors ($CPU1$ and $CPU2$). There also exist two co-processors ($HW1$ and $HW2$). Figures 42(a) and (b) are equivalent representations of processing and scheduling activity, however Figure 42(a) depicts this as Gannt charts per processing element whereas Figures 42(b) as Gannt charts per process.

- At $t = 0$, τ_C is released and immediately requests a processor (which it is

granted, as there is no competition). Without loss of generality, the processor granted to τ_C is $CPU1$.

- At $t = 2$, τ_E is released and likewise, immediately gets to execute on the remaining free processor ($CPU2$).

- At $t = 3$, τ_A is released and requests a processor but since its priority is less than that of either τ_C, τ_E, it is immediately preempted.

- At $t = 4$ though, τ_D is released and requests a processor as well. At that instant, four processes (τ_A, τ_C, τ_D, τ_E) are each competing for a processor (from among the two processors in the system). Of those, τ_D and τ_E have the highest priorities, thus τ_C is preempted and τ_D gets to execute on $CPU1$.

- At $t = 6$, τ_E terminates so one process less now competes for a processor. Thus, τ_C gets to resume execution, albeit on $CPU2$ (not $CPU1$, where it was previously executing). (Note that on any occasion where the set of processes executing on the array changes, which process gets to execute on which actual processor is not significant. We assume though that the process swapped in replaces the process swapped out on the respective processor, because this does not involve unnecessary process migrations.)

- At $t = 7$, τ_B is released but immediately preempted because it is not among the two highest-priority processes requesting a processor on that given instant.

- At $t = 9$, τ_C terminates and τ_B gets to execute on $CPU2$.

- At $t = 10$, τ_D issues a hardware operation, temporarily freeing $CPU1$, which is then granted to τ_A.

- On completion of that hardware operation though, at $t = 14$, three processes (τ_D, τ_B and τ_A) are competing for execution on the processor array. Thus τ_B continues to execute on $CPU2$ whereas τ_A is preempted and $CPU1$ is granted once again to τ_D.

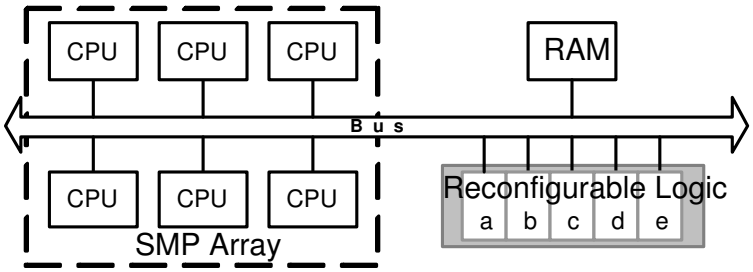

Figure 41: A multiprocessor architecture with application specific co-processors

- τ_B issues a hardware operation at $t = 16$, which frees up $CPU2$ for τ_A (which thus resumes its execution on a processor other than the one it started executing on).

- τ_A terminates at $t = 19$, while τ_B is still executing in hardware.

We require system resources to be global and to be accessed symmetrically by all processing elements. One could argue in favor of local memories attached to specific processors, which would then effectively constitute a *Non Uniform Memory Access* (NUMA) architecture, such as the one depicted in Figure 43. Local memory in NUMA architectures is accessed through the local (to the processor in consideration) I/O controller whereas remote memory is accessed through two additional hops: across the system bus and through the I/O controller of the remote processor to which the memory is attached. Obviously, access latency is greater for remote memory. We explain why the NUMA approach would be problematic:

First, as processes are not attached to any specific processor (and may even migrate during their activation), there is no incentive to having any resource be local to any particular processor. Moreover, having process state reside in local memory would introduce long delays (due to data transfer) upon process migration. By having all memory resources be global and symmetrically accessed, we can assume migration costs to be negligible. Finally, it would be impossible to determine during offline

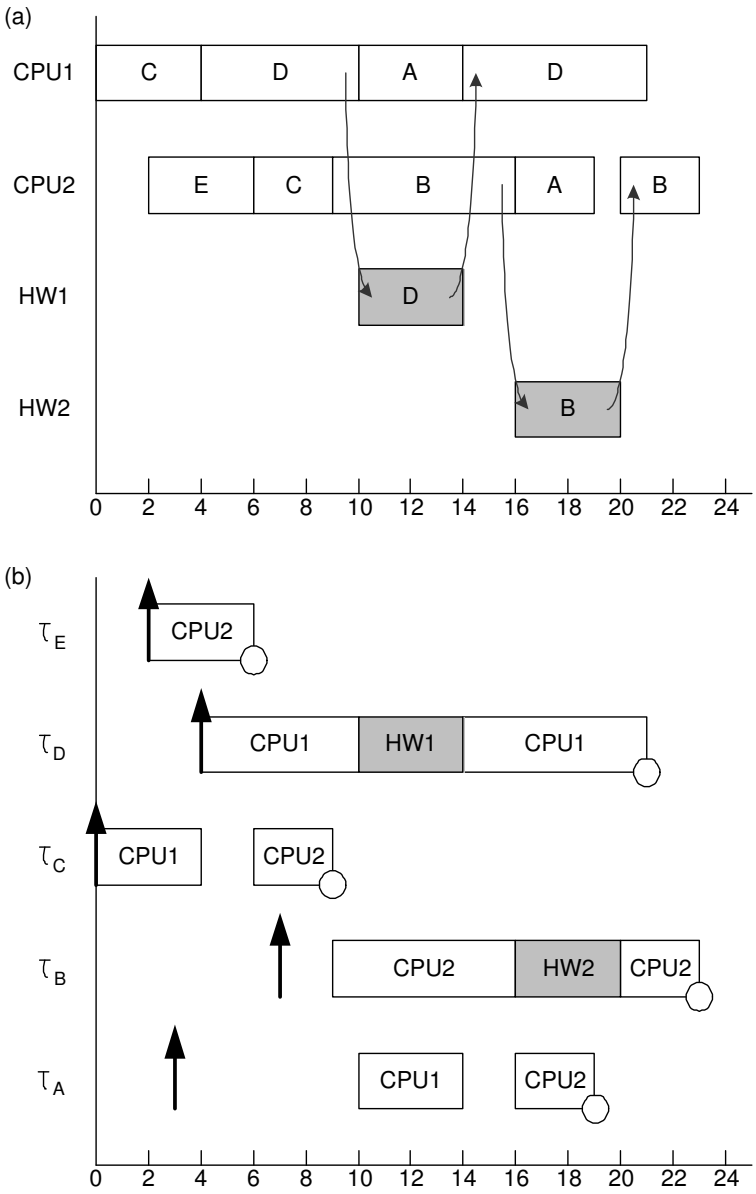

Figure 42: Simulation of scheduling decisions in a multiprocessor limited parallel system. In this example, the processor pool consists of 2 CPUS.

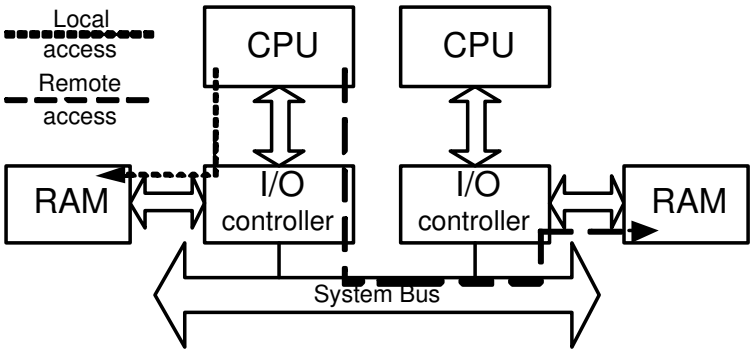

Figure 43: A typical NUMA architecture

WCET analysis whether any particular memory access would be local or remote, thus the worst would then have to be assumed.

The issue then of access to the shared bus system arises. To cope with that, we assume that it is possible for memory accesses to be blocked upon accessing the bus for very short intervals. However, we also assume that bus access arbitration is handled in a fair manner by the communication controllers of the processors. The resulting increased memory latencies are then an aspect of the architecture/implementation and, as such, are factored in during the WCET analysis of the code (something which lies beyond the scope of our work).

Before proceeding with analysis for such multiprocessor architectures, we place one additional restriction: no resources are to be shared between processes. We will eventually revisit this restriction and examine if it may be removed and how.

A.1 Timing analysis for multiprocessor architectures

Intuitively, the existence of $N > 1$ processors would increase availability and reduce the interference suffered by processes. We proceed to quantify this effect and calculate upper bounds for the WCRTs of processes in such systems.

For some process τ_i, the term W_i denotes an upper bound for the time spent by all

higher-priority processes executing on the processor array, during an activation of τ_i. (In the context of multiprocessor systems, the term *local execution* will be referring to execution locally to the array, not to any particular processor, as processors are pooled).

W_i is measured in processor ticks. If there is only one processor in the system (i.e. $N = 1$, our familiar architecture), then $W_i \geq I_i$ (if τ_i contains gaps) or $W_i = I_i$ (if not). In other words, whenever τ_i gets to compete for the (single) processor with any higher-priority process, it is denied the processor and suffers interference. However, with $N > 1$ processors, τ_i could only be denied a processor to execute on (thus suffering interference) when there are N or more higher-priority processes also competing for a processor on the given instant (thus, the N processors are granted to the N highest-priority processes among them).

An upper bound for the cumulative time for which this condition may hold true for a given process τ_i over a time window of length equal to its WCRT, is thus an upper bound for the worst-case interference suffered by τ_i. We will show that such a bound may be derived as

$$I_i = \begin{cases} 0 & \text{if } \tau_i \text{ is among the N highest-priority processes} \\ \\ \left\lfloor \frac{W_i}{N} \right\rfloor & \text{otherwise} \end{cases} \tag{32}$$

Proof:

If τ_i is among the N highest-priority processes, it follows that it will always be granted a processor immediately upon request and that it will never have to be preempted. If not: Assume that the worst-case interference for τ_i is $\breve{I}_i = I_i + a$, a being a positive integer and I_i being given by Equation 32. There would then be at least $\breve{I}_i$ instants from the release of some activation of τ_i until its termination during which all processors would be busy hosting higher-priority processes. Thus

the cumulative time spent by higher-priority processes executing on the processor array would have been at least

$$N\breve{I}_i = N(I_i + a) = Na + N\left\lfloor \frac{W_i}{N} \right\rfloor \geq Na + N\frac{W_i}{N} - 1 = Na - 1 + W_i > W_i$$

which is impossible (as per the definition of W_i).
□

Thus, upper bounds on interferences derived for any given process set are inversely proportional to N (the number of processors in the array) for those processes that do suffer interference – the N highest-priority processes do not suffer any interference, as already shown.

Regarding the calculation of W_i, either the our basic approach (the one presented in Chapter 4) or the synthetic approach (introduced in Chapter 5) may be used. The equation which outputs an upper bound for the WCRT of some local block of length Δx of process τ_i then is

$$R(\tau_i, \Delta x) = \begin{cases} \Delta x & \text{if } \tau_i \text{ is among the N highest-priority processes} \\ \\ \Delta x + \left\lfloor \frac{W(\tau_i, \Delta x)}{N} \right\rfloor & \text{otherwise} \end{cases} \tag{33}$$

where $W(\tau_i, \Delta x)$ is a placeholder for

$$W(\tau_i, \Delta x) = \sum_{j \in hp(i)} \left\lceil \frac{R_i + C_j - X_j}{T_j} \right\rceil X_j \tag{34}$$

if the basic analysis is used or

$$W(\tau_i, \Delta x) = \sum_{\substack{j \in hp(i) \\ \tau_j : nonlinear}} \left\lceil \frac{R_i + C_j - X_j}{T_j} \right\rceil X_j + \sum_{\substack{j \in hp(i) \\ \tau_j : linear}} \sum_{\substack{k=0 \\ R_i > {}^{\xi}O_{j_k}}}^{n(\tau_j)} \left\lceil \frac{R_i - {}^{\xi}O_{j_k} + A_j}{T_j} \right\rceil {}^{\xi}X_{j_k} \tag{35}$$

if the synthetic analysis is used. If so, in the general case (when τ_i may contain gaps), the decomposition algorithm formulated earlier (see Figure 23 in page 140 for the pseudocode) which combines the "split" and "joint" analytical approach in the context of limited parallel systems with a single general-purpose processor ($N = 1$) is also applicable in the context of multiprocessor limited parallel systems ($N > 1$) without modification (as can be established by inspection).

Note that for $N = 1$, Equation 33 is reduced to the familiar (from Chapters 4 and 5) corresponding equations (under the basic/synthetic analysis, respectively) for limited parallel architectures with a single general-purpose processor:

- If the basic analysis is used, it reduces to Equation 6.
- If the synthetic analysis is used, it reduces to Equation 15.

Moreover, note that, in the absence of any hardware co-processors, the architecture is reduced to an N-way Symmetric Multiprocessor array – which is then also addressed by the above analysis.

A.2 Observations regarding the effect of additional processors on the worst-case synthetic distribution of a process

Since the synthetic worst-case execution distribution of a given process is a construct useful for bounding the cumulative time spent by the process executing in software

during a given interval, its derivation in the context of multiprocessor variants (i.e. with $N > 1$) of the limited parallel architecture is performed in the same manner (i.e. using the same construction algorithm) as in the context of the original model (i.e. with $N = 1$).

It should be noted though that the worst-case synthetic distribution for some process derived for a system with $N = 1$ may, in the general case, be different than the one derived when, all other things remaining equal, $N > 1$ (i.e. when the architecture is modified via the addition of one or more pooled processors). We elaborate:

The length of what we referred to as the *notional gap* for a process τ_j (during the construction of its synthetic worst-case execution distribution) is $N_j = T_j - R_j$. By increasing N (all other things remaining equal), interference suffered by τ_j in the worst-case cannot increase as a result. Thus R_j cannot increase either, which in turn determines that $N_j = T_j - R_j$ cannot decrease. We have already shown (in the context of the local optimisation described in Section 5 of Chapter 5) how an increase in the length of the notional gap (all things remaining equal) of some process $\tau_j \in hp(i)$ may not cause the upper bound on the cumulative execution time of $hp(i)$ during an activation of τ_i (derived under our worst-case scenario) to increase (in fact, it may decrease). Thus, W_i does not increase if more processors are added. Consequently, since an upper bound for the worst-case interference suffered by τ_i is $I_i = \left\lfloor \frac{W_i}{N} \right\rfloor$ (which is decreasing with increasing N), increasing the number of processors in the pool never causes the upper bounds on process WCRTs derived under the synthetic analysis to increase (which would have been an anomaly).

A.3 Priority assignment

In Chapter 7 we described an optimal priority assignment for systems with limited parallelism, scheduled under a fixed-priority preemptive scheme and analysed under our synthetic approach. Those systems had only one processor (and multiple

co-processors). We will show that the same algorithm is applicable to the multiprocessor extension of limited parallel system described above, as long as there are no shared resources and the scheduling is fully preemptive.

By inspection of Equations 33, 35 and 34, it is seen that the derived upper bound for the WCRT of some process τ_i is monotonically increasing as C_i increases. Moreover, it is monotonically non-decreasing if, for a given partial priority ordering of the remaining tasks, its priority decreases. Thus, we deduct that the analysis is safe from any anomalies such as upper bounds on response times increasing with either increasing priority or with increasing execution time.

In the past, Graham had provided examples of anomalous behaviors in multiprocessor systems in [44] but these were observed in systems which were non-preemptive and, additionally, which were scheduled under the *Earliest Deadline First* scheme. Those findings do not apply to our model.

A.4 Obstacles to resource sharing

Both variants (i.e OCPP/ICPP) of the Priority Ceiling Protocol, as formulated for uniprocessor systems are able to bound the time that any process spends blocked, while preventing deadlock at the same time. Worst-case blocking experienced by a process activation is a function of the duration of critical sections of lower-priority processes. Since critical sections are typically short (in comparison to the execution times of processes), worst-case blocking terms are short as well and, by this criterion, the protocol is fair and efficient.

However, if applied (under its original formulation unaltered) to multiprocessor systems, the same protocol, while still preventing deadlock (by virtue of the strict priority ordering of critical sections), has problematic timing behavior. For example, consider a process τ_i with multiple non-overlapping accesses to shared resources. In a uniprocessor system, the worst-case blocking for the process would have been

at most the length of one specific critical section (which one depends on the process and which resources are shared by which processes). However, in a multiprocessor system there is nothing which prevents some other process τ_j (irrespective of its priority), executing in parallel on some other processor, from potentially obtaining exclusive access to a resource just before τ_i attempts to access it. (In a uniprocessor system, a lower-priority process would have been preempted, unless exclusive access to the shared resource was obtained prior to the release of τ_i). Thus, τ_i could potentially block upon each and every access to a shared resource. Moreover, there is nothing which then prevents τ_j from being preempted by higher-priority processes or from being in turn blocked by a third process τ_k executing on some other processor (a behavior termed *chained blocking*).

It is thus not clear if worst-case blocking per process may be bounded at all under the Uniprocessor PCP in multiprocessor systems. What is obvious however is that worst-case blocking terms may be unacceptably long.

Rajkumar et al. provide in [71] other examples of problematic behavior under the Uniprocessor PCP in the context of multiprocessor systems as motivation for the development of a variant of the protocol appropriate for multiprocessor systems. However, these examples assume that, in the multiprocessor system in consideration, processes are statically assigned to specific processors. This contravenes our assumption of a shared scheduling queue to a pooled array of processors (and resulting potential migration of processes). Our examples, on the other hand, demonstrate problematic behaviors that are observable under either partitioned multiprocessor systems (as assumed by Rajkumar et al.) or under processor pooling (as per our model).

The same assumption of a fixed partitioning of processes to processing elements is propagated as a fundamental assumption in the formulation of those resource management protocols (based on the concept of priority inheritance) which are targeted at systems with multiple processors (namely, the Distributed Priority Ceiling Proto-

col and variants of the Multiprocessor Priority Ceiling Protocol) [71, 68]). Separate bounds for worst-case *local blocking* (i.e. on resources shared only by processes mapped to the same processor) and *remote blocking* (i.e. respectively, by processes mapped to different processors) are then derived. The protocols are engineered to ensure that the execution of some process outside a critical section or even inside a local critical section may not contribute to the time that a higher-priority process spends blocked on a global (i.e not local to any processor) shared resource.

Given the assumption of a fixed allocation of processes to processors, none of the the above extensions to the PCP are directly applicable to our model. Moreover, it is unclear how any of them is to be modified so as to target a multiprocessor architecture with pooled processors and a shared scheduling queue. As an additional requirement some of the above protocols depend on one or more dedicated *synchronisation processors* (for the execution of global critical sections; instead of execution on the host processor of the respective process). This option is not accounted for by our model either. Given though that we expect limited parallel systems to be a product of codesign (and not forced upon a predetermined architecture), such a feature, in principle, could be introduced if deemed necessary for the incorporation of shared resources without compromising the behavior of the system with respect to its real-time constraints. However, as noted earlier, no existing resource management scheme known to us is appropriate.

The development of such an appropriate shared resource management protocol is a task that exceeds the scope of this thesis. For our original limited parallel architecture (with a single general-purpose processor), we were able to successfully apply an existing protocol, the PCP, originally formulated in the context of uniprocessor systems, for the purpose of introducing shared resources. In the absence of an appropriate protocol for our multiprocessor extension, our timing analysis is limited to systems without shared resources. While this greatly limits its immediate usefulness, we expect that in the future the formulation of our multiprocessor analysis

will trigger the development of a resource management scheme to match it.

A.5 Summary

Within this appendix we introduced a multiprocessor extension of the limited parallel architecture. Under this extended model, instead of a single general purpose processor, there is a pool of N processors. All system resources are symmetrically accessible by all processors. There is a single scheduling queue; a process is not bound to any specific processor. On any given instant, the N highest-priority processes among those competing for a processor, may be executing in software (each one some processor from the pool). On the same instant, other processes may be executing remotely in parallel, each on some hardware co-processor. In the absence of hardware co-processors, this model is reduced to a Symmetric Multiprocessor (SMP) architecture (similarly to how the basic limited parallel architecture is reduced to a uniprocessor architecture, in the absence of co-processors).

We formulated worst-case analysis for the above multiprocessor extension of our model which is free from anomalies and scalable with respect to N (the processor count) – per process worst-case interference augmented by unity is at worst inversely proportional to N. The issue of optimal priority assignment in the context of such systems was also addressed. However, in the absence of an appropriate resource management protocol for our model, we had to disallow resource sharing, which limits the applicability of both this model and its associated analysis until such a protocol is formulated.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Tools and Techniques*. Addison-Wesley, 1986.

[2] I. Alston and B. Madarar. From c to netlists: hardware engineering for software engineers? *Electronics and Communication Engineering Journal*, 14(4):165–173, aug 2002.

[3] Altera Corporation. *Altera Product Information : http://www.altera.com/products*, 2005.

[4] ARM Ltd. *Arm Products and Solutions: http://www.arm.com/products/*, 2007.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, Atlanta, GA, USA, 1991.

[6] N. C. Audsley. The synchronous computational model for safety-critical systems: A timing analysis perspective. Technical report, Dependable Computing Systems Centre, Dept. of Computer Science, University of York, 1995.

[7] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[8] N. C. Audsley, I. J. Bate, and M. Ward. Mapping Concurrent Real-Time Software Languages to FPGA. In *Proceedings of the 3rd UK ACM SIGDA Workshop on Electronic Design Automation*, page (not applicable), 2003.

[9] N. C. Audsley and K. Bletsas. Fixed Priority Timing Analysis of Real-Time Systems with Limited Parallelism. In *Proc. Euromicro Conference on Real-Time Systems*, pages 231–238, 2004.

[10] N. C. Audsley and K. Bletsas. Realistic Analysis of Limited Parallel Software / Hardware Implementations. In *Proc. 10th Real Time Applications Symposium*, pages 388–395, 2004.

[11] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Eng. J.*, 8(5):284–292, 1993.

[12] A. Baghdadi, N. Zergainoh, W. Cesrio, and A. Jerraya. Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems. *Information Processing Letters*, 28(9):83–86, Sept. 2002.

[13] C. M. Bailey, A. Burns, A. J. Wellings, and C. H. Forsyth. A Performance Analysis of a Hard Real-Time System. *Control Engineering Practice*, 3(4):447–464, 1995.

[14] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems - The POLIS Approach.* Springer, 1997.

[15] F. Barat, R. Lauwereins, and G. Deconinck. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. *IEEE Transactions on Software Engineering*, 28(9):847–862, Sept. 2002.

[16] I. J. Bate. *Scheduling and Timing Analysis for Safety-Critical Real-Time Systems.* DPhil. Thesis YCST/99/04, Department of Computer Science, University of York, UK, 1999.

[17] G. Berry and G. Gouthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[18] K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited paralellism. In *Proc. Embedded and Real-Time Computing Systems and Applications Conference (RTCSA)*, pages 525–531, 2005.

[19] K. Bletsas and N. C. Audsley. Optimal priority assignment in the presence of blocking. *Information Processing Letters*, 99(3):83–86, Aug. 2006.

[20] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the 23rd Real-Time Systems Symposium*, pages 269–278, Austin, Texas, Dec. 2002. IEEE.

[21] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, Apr. 1994.

[22] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[23] G. C. Butazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*. Springer, 2nd edition, 2005.

[24] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *Proceedings of the 39th Design Automation Conference*, pages 789–794, June 2002.

[25] K. S. Chatha and R. Vemuri. Hardware-software codesign for dynamically reconfigurable architectures. In *Proc. 9th International Workshop on Field-Programmable Logic and Applications*, pages 175–184. Springer-Verlag, Berlin, 1999.

[26] K. S. Chatha and R. Vemuri. Magellan: Multiway hardware/software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proc. 9th International Symposium on Hardware-Software Codesign (CODES 2001)*, pages 42–47, Apr. 2001.

[27] G. De Micheli, R. Ernst, and W. Wolf. *Readings in Hardware / Software Codesign*, chapter 1 – Introduction, pages 1–4. Morgan-Kaufman, 1993.

[28] G. De Micheli and R. K. Gupta. Hardware/Software Co-Design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[29] M. Dertouzos. Control robotics: the procedural control of physical processes. In *Proceedings of IFIP Congress*, pages 807–813, 1974.

[30] S. Dey, D. Panigrahi, L. Chen, C. N. Taylor, K. Sekar, and P. Sanchez. Using a Soft Core in a SoC Design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, 2001.

[31] A. Doboli and P. Eles. Scheduling under data and control dependencies for heterogeneous architectures. In *Proceedings of the International Conference on Computer Design - ICCD*, pages 602–608, 1998.

[32] S. A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[33] EECS Department, University of California, Berkeley. *Ptolemy II Design Document*, 2007.

[34] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Journal on Design Automation for Embedded Systems*, 2:5–32, 1997.

[35] R. Ernst. Codesign of Embedded Systems: Status and Trends. *IEEE Design and Test of Computers*, 15(2):45–54, Apr. 1998.

[36] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, Dec. 1993.

[37] K. Fowler. Mission-critical and safety-critical development. *IEEE Instrumentation and Measurement Magazine*, 7(4):52–59, Dec. 2004.

[38] A. B. G. Bernat and A. Llamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, Apr. 2001.

[39] D. Gajski and F. Vahid. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design and Test of Computers*, 12(1):53–67, 1995.

[40] D. Gajski, F. Vahid, S. Narayan, and J. Gong. Specsyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. *IEEE Transactions on Very Large Scale Integration Systems*, 6(1):84–100, 1998.

[41] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[42] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 2nd edition, 1997.

[43] K. Goosens, S. Pestana, J. Dielssen, O. Gangwal, J. Meergergen, A. Radulescu, E. Rijpkema, and P. Wielage. Service-based design of systems on chip and networks on chip. In book: *Dynamic and Robust Streaming in and between Connected Consumer Electronic Devices*, pages 37–60, Springer, 2001.

[44] R. L. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mar. 1969.

[45] R. K. Gupta and G. De Michelli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design and Test of Computers*, 10(3):29–41, 1993.

[46] R. K. Gupta and Y. Zorian. Introducing Core-Based System Design. *IEEE Design and Test of Computers*, 14(4):15–25, 1997.

[47] A. Jerraya, H. Tenhunen, and W. Wolf. Multiprocessor Systems-On-Chips. *IEEE Computer*, 38(7):36–40, Jul. 2005.

[48] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming models and HW-SW Interfaces Abstraction for Multi-Processor Soc. In *Proceedings of the 43rd Design Automation Conference*, pages 280–285, Jul. 2006.

[49] A. A. Jerraya and W. Wolf. Hardware/Software Interface Codesign for Embedded Systems. *IEEE Computer*, 38(2):63–69, Feb. 2005.

[50] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29(5):390–395, October 1986.

[51] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 13 May 1983.

[52] J. Y. T. Leung and M. L. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Inf. Pro. Letters*, 11(3):115–118, 1980.

[53] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Perf. Eval. (Netherlands)*, 2(4):237–250, 1982.

[54] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. of the ACM*, 20(1):40–61, 1973.

[55] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of International Workshop on Hardware-Software Codesign*, 1993.

[56] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. Technical Report UCB/ERL M93/48, EECS Department, University of California, Berkeley, 1993.

[57] B. Madahar and I. Alston. How rapid is rapid prototyping? *Electronics and Communication Engineering Journal*, 14(4):155–164, aug 2002.

[58] P. Marwedel. Guest editorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1281–1282, Nov. 2001.

[59] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[60] A. K. Mok and D. Chen. A Multiframe Model for Real-Time Tasks. In *Proceedings of the IEEE International Real Time Systems Symposium (RTSS)*, pages 22–29, Dec. 1996.

[61] A. K. Mok and D. Chen. A Multiframe Model for Real-Time Tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, Oct. 1997.

[62] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan-Kaufmann, 1997.

[63] J. C. Palencia, J. J. Gutierrez-Garcia, and M. Gonzalez-Harbour. Best-case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time

Systems. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, pages 35–44, June 1998.

[64] J. C. Palencia-Gutierrez, J. J. Gutierrez-Garcia, and M. Gonzalez Harbour. On The Schedulability Analysis For Distributed Hard Real-Time Systems. In *Proc. Euromicro Conf. on Real-Time Systems*, pages 136–143, 1997.

[65] P. Pop. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. PhD thesis, Dept. of Computer and Information Science, Linköping University, Linköping University, SE-581 83 Linköping, Sweden, 2003.

[66] P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES '99)*, pages 178–182, 1999.

[67] P. Pop, P. Eles, and Z. Peng. Performance Estimation for Embedded Systems with Data and Control Dependencies. In *8th Int. Symp. on Hardware/Software Codesign*, pages 62–66, 2000.

[68] R. Rajkumar. *Synchronization In Real-Time Systems - A Priority Inheritance Approach*. Kluwer, 1991.

[69] R. Rajkumar. Real-time synchronisation protocols for shared memory multiprocessors. *Proceedings 10th IEEE International Conference on Distributed Computing Systems*, pages 116–123, 28 May - 1 June 1990.

[70] R. Rajkumar, L. Sha, and J. P. Lehoczky. On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment. In *Proceedings IEEE Real-Time Systems Symposium*, pages 2–11, 1987.

[71] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronisation protocols for multiprocessors. *Proceedings IEEE Real-Time Systems Symposium*, pages 259–269, December 1988.

[72] O. Redell. Analysis of Tree-Shaped Transactions in Distributed Real-Time Systems. In *Proc. Euromicro Conference on Real-Time Systems*, pages 239–248, 2004.

[73] O. Redell and M. Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *Proceedings of 14th Euromicro Conference on Real-Time Systems, ECRTS 2002*, pages pp. 165–172. IEEE Computer Society Press, June 2002.

[74] O. Redell and M. Törngren. Calculating Exact Worst Case Response Times for Static Priority Scheduled Tasks with Offsets and Jitter. In *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium*, pages 164–172, Sept. 2002.

[75] L. Sha, T. Abdelzaher, K. E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Journal of Real Time Systems*, 28(2/3):101–155, 2004.

[76] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. *Proceedings IEEE Industrial Electronics Conference (IECON)*, pages 909–915, 1987.

[77] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–85, Sept. 1990.

[78] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. Butazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[79] K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. YCS 197, Department of Computer Science, University of York, April 1993.

[80] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[81] K. Tindell, H. Hansson, and A. Wellings. Analysing real-time communications: Controller Area Network (CAN). *Proceedings IEEE Real-Time Systems Symposium*, pages 259–265, December 7-9, 1994.

[82] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Journal of Real-Time Systems*, 14:219–250, 1998.

[83] F. Vahid and T. Givargis. *Embedded System Design - A Unified Hardware/Software Approach*. Wiley, 2002.

[84] F. Vahid, R. Lysecky, C. Zhang, and G. Stitt. Highly configurable platforms for embedded computing systems. *Microelectronics Journal*, 34(11):1025–1029, Nov. 2003.

[85] F. Vahid, S. Narayan, and D. Gajski. SpecCharts: A VHDL front-end for embeded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, Jun. 1995.

[86] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan-Kaufmann, 2001.

[87] W. Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan-Kaufmann, 2006.

[88] Xilinx Corporation. *Xilinx Product Information : http://www.xilinx.com/products*, 2005.

[89] A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*, chapter Preface, pages vii–xi. Kluwer, 2000.

[90] T. Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, Dec. 1998.