FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Scheduling Parallel Real-Time Tasks in Multiprocessor Platforms

Cláudio Roberto Ribeiro Maia

Doctoral Program in Electrical and Computer Engineering

Supervisor: Luís Miguel Pinho Nogueira

Co-Supervisor: Luís Miguel Rosário da Silva Pinho

November 6, 2018

© Cláudio Roberto Ribeiro Maia, 2018

Scheduling Parallel Real-Time Tasks in Multiprocessor Platforms

Cláudio Roberto Ribeiro Maia

Doctoral Program in Electrical and Computer Engineering

November 6, 2018

Resumo

No passado era suficiente aumentar a frequência de execução em processadores mononúcleo, de modo a adicionar novas funcionalidades de *software*. No entanto, devido a limitações no meio físico, os fabricantes de processadores deixaram de fabricar processadores mononúcleo em favor de processadores multinúcleo. Enquanto, no âmbito geral, esta adaptação é benéfica para a indústria de *software*, dado que permite a inclusão de funcionalidades mais exigentes e complexas nas aplicações, a utilização de sistemas comerciais de *hardware* (conhecidos como COTS - *Commercial-Off-The-Shelf*) em ambientes de tempo-real apresenta um desafio ainda em aberto. De facto, em sistemas de tempo-real, a previsibilidade é considerada mais importante do que o desempenho e é um requisito para a exactidão, num domínio que é sobejamente conhecido pelos seus rigorosos requisitos temporais.

Podemos apontar duas razões principais para este desafio. Em primeiro lugar, as arquitecturas COTS são desenhadas para sistemas em que o desempenho no caso médio é importante, e por conseguinte, recursos como componentes de memória (*i.e.,* memória principal, memória *cache*, *etc.*), periféricos e barramentos, são partilhados entre os vários núcleos do sistema. Consequentemente, se não houver cuidado, desvios temporais, daqueles que foram estimados em tempo de desenho da aplicação, podem ocorrer (devido a interferência) sempre que núcleos diferentes acedem simultaneamente aos recursos partilhados do sistema. Em segundo lugar, a plataforma de *hardware* não só suporta execução concorrente ao nível dos núcleos, mas também suporta execução paralela ao nível da plataforma. Por conseguinte, o objectivo principal para a comunidade de sistemas de tempo-real é encontrar soluções eficientes para lidar com o comportamento paralelo e inerente à plataforma, e ao mesmo tempo, assegurar a previsibilidade das aplicações, tendo em consideração os recursos partilhados da plataforma. Nesta dissertação, este objectivo é dividido em dois problemas distintos que são abordados de forma independente, nomeadamente: (i) o problema de escalonamento de tarefas paralelas com restrições temporais em plataformas multiprocessador; (ii) o problema da partilha de recursos em plataformas multiprocessador.

O primeiro problema (o problema de escalonamento de tarefas paralelas com restrições temporais em plataformas multiprocessador) é coberto usando duas perspectivas diferentes. A primeira perspectiva foca-se no tempo de resposta de tarefas paralelas e de tempo-real utilizando o modelo síncrono de tarefas paralelas. O modelo considerado tem como alvo tarefas com prioridades fixas, compostas por vários segmentos, em que cada segmento é composto por um número arbitrário de unidades de execução independentes e que podem ser executadas em paralelo. Para alcançar este objectivo, novos conceitos, tais como decomposição do *carry-out* e janela deslizante, são introduzidos de modo a derivar um cenário no pior caso que permita computar o pior tempo de resposta de cada tarefa que executa no sistema. Na segunda perspectiva, o problema é analisado considerando uma abordagem mais dinâmica. Uma abordagem multifase é apresentada para analisar a escalonabilidade das tarefas de tempo-real seguindo um modelo *fork-join* antes e durante o tempo de execução. Esta abordagem tem a particularidade de, durante o tempo de execução, utilizar o algoritmo de *work-stealing* para reduzir o tempo médio de resposta das tarefas de tempo-real.

O segundo problema (o problema da partilha de recursos em plataformas multiprocessador) é abordado considerando uma plataforma na qual o barramento é partilhado entre os vários núcleos e, por conseguinte, é uma fonte de interferência sempre que pedidos de memória são feitos em simultâneo pelos vários núcleos do sistema. Para resolver este problema, o modelo de 3 fases é utilizado. Em primeiro lugar, uma análise empírica é realizada para comparar o desempenho de diferentes políticas de atribuição de prioridades com uma implementação da política de escalonamento *global Earliest Deadline First* (EDF) que considera interferência entre tarefas. De seguida, um teste de escalonabilidade para o modelo de 3 fases é proposto, tendo em consideração a interferência no barramento e a interferência entre tarefas.

Abstract

In the past, increasing the frequency in single-core processors was enough to accommodate new software features. However, due to physical limitations, processor manufacturers stopped releasing single-core processors in favour of multicore ones. While this move is beneficial for the software industry overall, as it allows the inclusion of more complex and demanding features into applications, the use of Commercial-Off-The-Shelf (COTS) multicore platforms in real-time systems still remains a challenge. In fact, in the real-time systems domain, predictability is considered more important than performance, and is a requirement for correctness in a domain well-known for their stringent timing requirements.

Two main reasons can be identified for such a challenge. First, COTS multicore architectures are designed for average-case performance and due to this, resources, such as memory components (*i.e.,* main memory, memory caches, *etc.*), peripheral devices, and buses, are shared among the different cores. Consequently, if care is not taken, timing deviations, from the ones estimated at design time, may occur due to interference whenever different cores simultaneously access shared resources. Second, the platform not only supports concurrent execution at a core level but it also supports parallel execution at the platform level. Therefore, the major goal for the real-time systems community is to find efficient ways of dealing with the inherent parallel behaviour of the platform, and at the same time, be able to ensure application predictability by taking into account the shared resources in the platform. In this dissertation, this goal is divided into two distinct problems which are dealt independently from each other: (i) the problem of scheduling parallel real-time tasks in multiprocessor platforms; and (ii) the problem of sharing resources in multiprocessor platforms.

The first problem (scheduling parallel real-time tasks in multiprocessor platforms) is covered from two different perspectives. In the first one, we focus on the response-time of synchronous parallel real-time tasks. The model under consideration targets tasks with fixed priorities, composed of several segments, each with an arbitrary number of parallel and independent units of execution that can be executed in parallel. New concepts such as carry-out decomposition and sliding window are introduced to derive a worst-case scenario that allows one to compute the worst-case response-time of each task executing in the system. In the second perspective, the problem is analysed considering a more dynamic approach. A multi-stage approach is presented to analyse the schedulability of fork-join real-time tasks before and during runtime. The particularity of this approach is that during runtime the work-stealing algorithm is used to reduce the average response-time of real-time tasks.

The second problem (sharing resources in multiprocessor platforms) is addressed by considering a platform where the memory bus is shared among cores. Consequently, it is a source of interference whenever simultaneous memory requests are issued by the cores in the platform. To solve this problem, the 3-phase task model is used. First, an empirical analysis is performed where the performance of different priority assignment policies is compared against an implementation of the global Earliest Deadline First (EDF) scheduling policy that considers inter-task interfer-

ences. Then, a schedulability test for the 3-phase task model is derived by taking into account the bus interference and task interference.

Acknowledgements

Many people contributed to this endeavour in many different ways. Thus, I would like to take this opportunity to show my respect and gratitude to all of them.

First of all, I am very grateful to my advisors, **Luís Miguel Pinho** and **Luís Miguel Nogueira**, for everything that they did for me. In the first place, for guiding me in the selection of a research topic and afterwards, for the time, support and long discussions about the research work that led to dissertation you are reading. In addition, for all the time and effort spent reviewing my work, including this dissertation. Finally, for teaching other valuable things related to life. Their friendship and professional collaboration meant a lot to me.

A PhD degree cannot be obtained without family support. For that, I have to thank my family in general, but specially my parents, **Carlos** and **Maria**, my wife **Joana** and our newborn **Dinis**, which also contributed to this research work in its own way, and finally, my brother **Luís and his family**.

To the people that collaborated with me, namely: **Marko Bertogna**, **Daniél Gracia Perez**, **Patrick Yomsi** and **Geoffrey Nellisen**. The hours we spent together discussing complex problems were very valuable to me. I learned several different things with all of you and I felt that not only I was improving as a researcher but also as a human being.

Finally, I would like to thank **CISTER Research Centre** as an institution for the opportunity and trust they gave me in order to pursue this degree. Moreover, as a research centre is just a hollow space without people, I also want to thank and give a warm appreciation to my co-workers at CISTER for sharing their insights, ideas, different visions and cultural perspectives, not only about my work but about life in general. A special thanks goes to **David Pereira**, **André Pedro** and **José Fonseca** for their friendship and support.

This research work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

Cláudio Maia

"The loneliness of a PhD student is equivalent to the loneliness of a long distance runner.
"

C. Maia, 2013

List of Publications

The following publications were developed in the scope of the research activities presented in this dissertation.

Journals

- Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. Real-time semi-partitioned scheduling of fork-join tasks using work-stealing. *EURASIP Journal on Embedded Systems*, 2017(1):31, Sep 2017b. ISSN 1687-3963. doi: 10.1186/s13639-017-0079-5

Conferences (in chronological order)

- Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Supporting real-time parallel task models with work-stealing, March 2012. Research Poster at The Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2012) (DATE Workshop)
- Cláudio Maia, Luís Nogueira, Luís Miguel Pinho, and Marko Bertogna. Response-time analysis of fork/join tasks in multiprocessor systems. In *Proceedings of Work-in-Progress Session of the 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris, France, July 2013. Work in Progress Session
- Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 3:3–3:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2727-5. doi: 10.1145/2659787.2659815
- Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. Semi-partitioned scheduling of fork-join tasks using work-stealing. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 25–34, Oct 2015. doi: 10.1109/EUC.2015.30
- Cláudio Maia, Luís Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. A closer look into the aer model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sept 2016. doi: 10.1109/ETFA.2016.7733567
- Cláudio Maia, Geoffrey Nelissen, Luís Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time*

Computing Systems and Applications (RTCSA), pages 1–10, Aug 2017a. doi: 10.1109/RTCSA.2017.8046313

Contents

Resumo i

Abstract iii

Acknowledgements v

List of Publications ix

1 Introduction 1

1.1 Parallelism . . . 2
1.1.1 Example of an OpenMP Task . . . 3
1.2 Parallelism and Real-Time Systems . . . 4
1.3 Resource Sharing . . . 5
1.4 Thesis Statement . . . 6
1.5 Contributions . . . 7
1.6 Thesis Structure . . . 8

2 Background and Related Work 9

2.1 Task Characterisation . . . 9
2.2 Platform Characterisation . . . 11
2.2.1 Processors . . . 12
2.2.2 Memory . . . 12
2.2.3 Memory Bus . . . 14
2.3 Multiprocessor Scheduling . . . 17
2.4 Parallel Real-Time Systems . . . 19
2.4.1 Parallel Task Models . . . 19
2.4.2 Earlier Parallel Models . . . 20
2.4.3 Recent Parallel Models . . . 21

3 Schedulability of Synchronous Parallel Tasks 25

3.1 Introduction . . . 25
3.2 System Model . . . 26
3.3 Critical Interference of Parallel Tasks . . . 27
3.4 Response-Time Analysis . . . 32
3.5 Sliding Window Technique . . . 33
3.6 Decomposing the Carry-out Job . . . 34
3.7 Workload of a Task Within a Window . . . 35
3.8 Schedulability Condition . . . 38
3.9 Complexity . . . 39

3.10 Evaluation 39
3.11 Summary 41

4 Applying Work-stealing to Real-time Systems **43**
4.1 Introduction 43
4.2 Randomised Work-stealing 44
4.3 Limitations of Randomized Work-stealing with Respect to Real-Time Systems . . 45
4.4 Literature on Real-Time Work-Stealing 46
4.5 A New Data Structure 48
4.6 Semi-partitioned Scheduling 49
4.7 System Model 50
4.7.1 Earliest Deadline First 50
4.7.2 Multiframe Task Model 51
4.8 Semi-partitioned Scheduling and Work-Stealing 52
4.8.1 Task Assignment Phase 53
4.8.2 Offline Scheduling Phase 54
4.8.3 Online Scheduling Phase 55
4.8.4 Example 56
4.9 Tasks with Density Greater Than 1 57
4.10 Schedulability Analysis 59
4.11 Simulation Results 63
4.11.1 Selected Heuristics 63
4.11.2 FFDO versus WFD 64
4.11.3 Overheads of the Approach 68
4.12 Summary 69

5 Schedulability of the 3-Phase Task Model **71**
5.1 Introduction 71
5.2 System Model 72
5.3 Runtime Execution Model 73
5.4 3-Phase vs. G-EDF in COTS Systems 73
5.4.1 Priority Assignment Policies 73
5.4.2 Simulator's Scheduling Behaviour 75
5.4.3 Experimental Settings 76
5.4.4 Experimental Results 77
5.5 Global Fixed-Priority Scheduling of the 3-Phase Task Model 82
5.5.1 Scheduling Policy 82
5.5.2 Background 84
5.5.3 A Different Perspective 87
5.5.4 Schedulability Analysis 89
5.5.5 Experimental Results 95
5.6 Summary 97

6 Conclusion **99**
6.1 Future Work 101

References **103**

List of Figures

1.1 Graphical representation of the code presented in Listing 1.1 4

2.1 Memory hierarchy in current COTS platforms 13
2.2 Example of a 3-phase task model schedule 16
2.3 Example of a fork/join task τ_i 21
2.4 Example of a synchronous parallel task τ_i 22

3.1 Task τ_i interfering on task τ_k 30
3.2 Densest possible packing of threads within the problem window 33
3.3 Densest possible packing of threads when a task skips some segment 35
3.4 Example of a decomposed job 35
3.5 Response-time analysis details 36
3.6 Number of schedulable task sets detected by the considered tests for m = 4 40
3.7 Number of schedulable task sets detected by the considered tests for m = 8 40

4.1 Work-stealing deque data structure 44
4.2 Priority inversion scenario 46
4.3 Fork-join task 50
4.4 Illustrative example of the proposed approach 56
4.5 Task decomposition with one task 58
4.6 Result of applying the proposed approach to a task set that contains a task with density greater than 1 59
4.7 Analysis proposed by Dorin et al. in [Dorin et al., 2010]. 60
4.8 Result after the offline analysis 61
4.9 Example of work-stealing and intermediate deadline computation 62
4.10 Possible cases for the admission control test 62
4.11 Percentage of unallocated tasks 64
4.12 Comparison between FFDO and WFD 65
4.13 Simulation results for FFDO and WFD 67

5.1 Simulation results for $m = 2$, E-phase smaller than A and R-phases 80
5.2 Simulation results for $m = 2$, E-phase larger than both A and R-phases 80
5.3 Simulation results for $m = 4$, E-phase smaller than A and R-phases 81
5.4 Simulation results for $m = 4$, E-phase larger than both A and R-phases 81
5.5 Comparison between $m = 2$ and $m = 4$, E-phase larger than both A and R-phases 82
5.6 Simulation results for $m = 8$ cores, slowdown = 1.5x 83
5.7 Problem Window 84
5.8 Computing the overlap lower-bound for $\rho = 2, m = 3$ in [Alhammad and Pellizzoni, 2014] 87

5.9 Pessimism of the analysis in [Alhammad and Pellizzoni, 2014] 87
5.10 Our schedulability analysis approach 88
5.11 Computing an upper-bound on bus holes 92
5.12 Bus holes 93
5.13 Schedulability ratio for $m = 4$ and as a function of the number of cores 95
5.14 Schedulability ratio for $m = 2$ and as a function of the memory ratio 96

Acronyms

BFD	Best-fit Decreasing
COTS	Commercial Off-the-shelf
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DBF	Demand Bound Function
DRAM	Dynamic Random Access Memory
EDF	Earliest Deadline First
FFD	First-fit Decreasing
FCFS	First Come, First Serve
FIFO	First-In, First-Out
GMF	Generalized Multiframe model
LIFO	Last-In, First-Out
MPI	Message Passing Interface
POSIX	Portable Operating System Interface
RAM	Random Access Memory
TDMA	Time-division Multiple Access
WCET	Worst-case Execution Time
WFD	Worst-fit Decreasing

Chapter 1

Introduction

The multiprocessor trend restarted recently (first multiprocessor systems appeared in the 60s/70s) and is moving at a fast pace. In 2001, Sun Microsystems and IBM (in a separated effort) manufactured the dual-core processors. Later on, in 2006, this type of processors became a mainstream technology powered by Intel and AMD. This shift in paradigm (moving from uniprocessor to multiprocessor systems) occurred due to the physical limitations of computer chips. Increasing the operating frequency and voltage of the chips leads to an exponential increase in power consumption and heat dissipation issues. In order to overcome such physical limitations, chip manufacturers increased the number of computing units operating in parallel *per* single chip, while maintaining a lower frequency of operation, *i.e.,* multicore systems. As a result of this paradigm shift, computing systems are gradually becoming multiprocessor, with each chip being composed of multiple core units. Nowadays, different platforms present a varying number of cores per chip, ranging from the tenths up to the hundreds. Some notable examples of such platforms are TILE64 from Tilera [Bell et al., 2008], the Epiphany processors designed by Adapteva [Adapteva, 2014] and the MPPA-256 Manycore Processor developed by Kalray [de Dinechin et al., 2013]. Future generations of processors are expected to integrate thousands of simple processors in a single chip [Asanovic et al., 2006].[1]

Perhaps the advantage of multicore systems that immediately stands out is the opportunity they offer to increase application performance by allowing each application to execute its code simultaneously and in parallel. However, while sequential programs execute faster if the clock speed of the processors is increased (under the assumption that concurrency is neglected), this is not the case for multicore platforms due to certain restrictions, such as workload distribution, synchronisation and coordination operations frequently occurring between cores, and the existence of shared resources. Hence, to obtain the best efficiency possible and take complete advantage of these platforms, sequential programs need to be rewritten and such restrictions be taken into account. This aspect is specially relevant in the real-time systems domain, the domain covered in this dissertation, where predictability is of utmost importance. In this domain and for efficiency

[1] In this dissertation, the terms multicore and multiprocessor are used interchangeably. Nevertheless, a clarification for the difference that exists between a core and a processor is given in the next chapter.

purposes, the move to a multiprocessor centred paradigm imposes new challenges as it requires moving out from traditional multiprocessor scheduling[2] algorithms that are focused on sequential tasks to scheduling algorithms that contemplate parallel tasks.

In this chapter we start by introducing relevant concepts related to parallelism and provide an example of how it can be exploited by applications, in Section 1.1. Then, we focus on the challenge of exploiting parallelism in real-time systems and the problem of resource sharing introduced by multiprocessor platforms, in Section 1.2 and Section 1.3, respectively. With both of these in mind (parallelism and resource sharing), we explicitly state which problems this dissertation intends to solve, in Section 1.4, and its contributions, in Section 1.5. Finally, the thesis structure is presented in Section 1.6.

1.1 Parallelism

Parallelism in computer programs can be exploited either explicitly, by using explicit parallel programming languages, or implicitly, by using implicit parallel programming languages [Freeh, 1996].

An explicit parallel programming language provides special constructs that allow the programmer to identify the opportunities for parallelism and break the program by its logical functionality. This division by functionality results in units of execution (commonly known as tasks or threads) that may be simultaneously executed in parallel in each of the platform's cores. The advantage of using such approach is that the programmer can write very efficient code at the cost of the time needed to produce it.

By using an implicit parallel programming language, the programmer relies on the compiler to automatically manage and extract parallelism at compile time in an implicit manner. The disadvantage of this approach is that it is compiler dependent and therefore not all parallelism may be discovered.

Besides the type of programming languages, as described above, there are other important aspects that require special attention from the programmer when developing parallel applications, as for instance the selection of programming models and frameworks/libraries.

Two well-known programming models [Diaz et al., 2012] exist for the development of parallel applications - shared memory and distributed memory models. In the shared memory model, all processors have access to the same random access memory (RAM) and tasks exchange data by accessing the shared memory. In the distributed memory model, each processor has access to its own private memory. If by any means a task needs data residing in another's processor memory, then both processors need to communicate by exchanging messages via a communication channel.

Several existing frameworks/libraries support the aforementioned models of computation (*e.g.,* OpenMP [OpenMP, 2011] or POSIX threads [Gallmeister, 1995] for shared memory, and Message Passing Interface (MPI) [MPI, 2014] for distributed memory, to name a few). The goal of

[2]The decision process that deals with the allocation of workload to system resources and its sequencing over a period of time is known as scheduling.

Listing 1.1: Simple OpenMP Example

```
some initialisation code

#pragma omp parallel num_threads(4)
{
        ... some computing intensive parallel code ...
}

clean up code
```

such frameworks/libraries is to enhance the programming languages and runtime environments with specific features that allow a programmer to focus on the functionality of a computer program, instead of focusing on the specific details of parallelisation. Some of these features include mechanisms for task creation and destruction, task synchronisation and scheduling, among others. Thus, a big advantage of such parallel frameworks/libraries is that they ease the programmer's effort by reducing the complexity of developing parallel programs.

1.1.1 Example of an OpenMP Task

Due to its importance and dominance in traditional multicore architectures, where all cores have access to the same memory address space, let us focus on the shared memory model and more specifically on OpenMP. Using OpenMP, a programmer has full control over the code parallelisation, thus making it an explicit parallel programming framework/library. Hence, programmers can annotate their programs to expose opportunities for parallelism and suggest a possible parallel decomposition to the framework's runtime. The annotations act as hints for parallelisation which may be considered by the runtime environment, as a function of the system load and with the objective of exploiting the maximum amount of parallelism possible.

Listing 1.1 presents a simple OpenMP example to show the reader of how such frameworks/libraries operate. In the example, the main thread of execution sequentially executes some initialisation code. After completing the initialisation phase, the main thread requires from the runtime the creation of 4 threads. Each of threads is responsible for the execution of the parallel block defined within the pragma *omp parallel*. After completing their execution, each of the parallel threads synchronise with the main thread so that the clean up code is executed. Then, the program ends its execution.

This model of execution is known as the *fork-join* model due to the fact that the main thread forks into several threads that join into a single main thread at the end of their execution. Figure 1.1 depicts a graphical representation of the code in Listing 1.1.

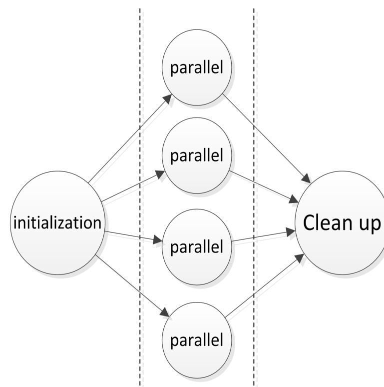


Figure 1.1: Graphical representation of the code presented in Listing 1.1

1.2 Parallelism and Real-Time Systems

A common definition found in the literature defines a real-time system as a system where its correct behaviour not only depends on the logical correctness of the system, but also on the time at which the operations are performed [Stankovic, 1988]. This type of systems is known for their predictability and stringent design requirements [Durrieu et al., 2014b; Leteinturier, 2007; Monot et al., 2010]. While in the past these systems were targeted at control applications, which are marked by their limited processing, a new set of applications (ranging from smart grids to autonomous driving) is demanding high processing in conjunction with real-time performance, and thus, powerful hardware is required to satisfy their needs [Pinho et al., 2015]. In fact, the required computing capacity they need can be obtained from state of the art multiprocessor platforms. In particular, real-time systems may take advantage of the platform's parallelism by distributing workload among the different cores for simultaneous execution, while using efficient scheduling techniques and consequently, better manage system resources.

However, bringing parallelism into real-time systems is not an easy task. Specially when the scheduling of applications is considered. Liu and Layland [Liu and Layland, 1973] observed the complexity of multiprocessor scheduling by stating the following:

> "... bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors."

While the uniprocessor scheduling problem reduces to deciding *when* to schedule each task, a new dimension adds to this one when shifting to multicores as it must also be decided *where* to execute each task.

Traditional multiprocessor scheduling deals with the scheduling of sequential tasks, that is, tasks which can only run in a single core. With this type of tasks, parallelism can only be exploited by increasing the number of tasks executing in the system and increasing the number of cores does not increase the execution speed for each task. This model of execution is known as

inter-task parallelism. Most results in real-time scheduling are devoted to the study of sequential tasks executing on multiple processors (see [Davis and Burns, 2011a] for a comprehensive and up-to-date survey). However, the shift from uniprocessors to multiprocessors revealed that scheduling real-time tasks is no longer a problem of scheduling sequential tasks. A real-time task may now exploit intra-task parallelism and be split into a set of sub-tasks that can be executed simultaneously in different processors at the same time instant (*i.e.,* potentially overlapping in time). Such a task is commonly denoted as a parallel real-time task or in short a parallel task.

On one hand, it is possible to take advantage of available cores to improve the execution of complex tasks with tighter timing constraints, whenever there is an opportunity in the system for the parallel execution of sub-tasks. On the other hand, such an approach requires efficiency from the system scheduler as now there is the need to map individual sub-tasks to each of the available cores. If the task's parallelism is rather regular, then it may be possible to find a mapping of tasks to cores at design time, such that the workload is balanced and the overall execution time of the task is reduced. But if the task's parallelism is irregular, then a static assignment of sub-tasks to cores may produce sub-optimal schedules where the workload is imbalanced (that is, some cores are excessively loaded while others are lightly loaded or even idle). Thus, the system scheduler must be efficient and capable of dynamically balance the workloads during runtime by taking into account the current system state and the dynamic nature of the tasks. The introduction of dynamic load balancing algorithms in the real-time systems domain is rather challenging due to the difficulty in guaranteeing the predictability of the system under analysis.

In the recent literature of real-time systems it is possible to find a few works that tackle parallel real-time tasks. Some of these works, namely [Lakshmanan et al., 2010] and [Saifullah et al., 2011], assume a model of execution similar to the fork-join model presented above in Figure 1.1. Nevertheless, analysing models that can leverage parallelism can be challenging from a schedulability viewpoint. For instance, usually there exists an execution dependency between different task segments [3] which imposes a partial order on execution.

1.3 Resource Sharing

In current multiprocessor architectures cores are not independent entities. They share physical resources, such as memory buses, memory controllers, last level caches, *etc.*, among themselves. While sharing resources may be beneficial for the threads of the same application, it may not be for threads of distinct applications as they compete for the resources, thereby introducing a problem of predictability for the real-time systems domain. In order to understand the implications of this problem, the reader needs to first understand how real-time systems are analysed with respect to their timing properties.

An important restriction that underpins the design of real-time systems is that (desirably) all timing properties should be met under all possible conditions. Consequently, these systems are

[3] A segment of a parallel real-time task is a region that is composed of an arbitrary number of sub-tasks where all sub-tasks can execute in parallel and independently from sub-tasks in other segments

analysed taking into account worst-case scenarios by the means of a *schedulability analysis*. The outcome of such analysis is a yes or no answer stating whether the system meets its deadlines or not, or in other words, whether it is schedulable or not.

An important parameter used in the definition of real-time applications and in the schedulability analysis is the *worst-case execution time* (WCET). The WCET is an upper-bound on the application's execution time considering the maximum time that it takes to execute in isolation in a given hardware platform[4]. WCET is also the parameter that is affected during runtime whenever there is contention in the system. Specifically, when two (or more) applications executing in two (or more) cores access shared resources simultaneously, the application's WCET may increase, which may jeopardize the results obtained offline from schedulability analysis.

Contention may occur due to several reasons, for instance a shared resource may only admit one access at a time (as it typically occurs in buses) or the state of a resource may be modified by one application accessing it in a way that it affects a concurrent application, causing a slowdown to the latter application (this behaviour is typically seen in shared caches) [Abel et al., 2013].

Several studies, as for instance [Zhuravlev et al., 2010], [Nowotsch and Paulitsch, 2012], [Radojković et al., 2012], show that due to shared resource contention, the execution times of applications may vary significantly. In particular, a common observed effect is the slowdown of applications due to *co-running applications*, *i.e.,* applications running on cores that share a resource. As an example on the amount of slowdown that can be observed due to shared resource contention, the authors in [Nowotsch and Paulitsch, 2012] observed a maximum slowdown of 5.1x in application execution, compared to execution in isolation, when multiple cores access network and memory concurrently. An even higher slowdown was observed by the authors in [Nélis et al., 2016]. In their research, they observed a slowdown of 8x due to co-running applications.

Solutions exist for the problem of contention in shared resources. For the cache contention problem, existing solutions apply cache partitioning strategies to eliminate interference between tasks from different cores, and consequently, bound the interference in the resulting non-partitioned shared caches. For the problem of bus contention, existing solutions use protocols to arbitrate the access to shared resources and analyse them accordingly in order to derive safe bounds. Such protocols can be time driven, *e.g.,* Time-division Multiple Access protocol (TDMA); event driven, *e.g.,* First Come, First Serve, Round Robin, *etc.,*; or a mix of both [Abel et al., 2013].

The conclusion that must be drawn from the above results is that special care must be taken when executing real-time applications in multicore platforms due to the existence of shared resources.

1.4 Thesis Statement

Moving from uniprocessor systems to multiprocessor systems is likely to fail if one does not take into account the problems that arise from such evolution, as the ones described above. Treating applications as if they are executing in a uniprocessor system while ignoring the parallel nature

[4] WCET and other parameters are formally described in the next chapter.

of the platform leads to an underutilization of system resources on one hand, and on the other hand, to an increase in interference due to co-running tasks. A clear aspect that still needs to be addressed by the real-time systems community is the lack of efficient models to handle the execution of parallel real-time applications and, ideally, that also cover the problem of resource sharing. Thus, the end goal of this dissertation is to have new models or enhance existing ones in order to derive the sound schedulability analysis that is needed by real-time applications running in multiprocessor systems.

This dissertation addresses the following two problems:

1. Problem of scheduling parallel real-time tasks in multiprocessor systems;
2. Problem of resource sharing in multiprocessor systems.

In particular, and related to each general problem above, we want to answer the following questions:

1. Is it possible to compute response-time upper bounds for parallel tasks when executing in multiprocessor systems?
2. Considering a scenario with co-running tasks and a shared resource, is it possible to compute upper-bounds on the interference imposed by co-running tasks in a multiprocessor system?

Motivated by the problems and questions above, the central proposition of this thesis is the following:

Real-time systems can be provided efficient schedulability tests that allows one to take advantage of multiprocessor systems. Supported models can consider intra-task parallelism or inter-task parallelism with shared resources. When dealing with intra-task parallelism, load-balancing is considered either naturally or via work-stealing. When dealing with shared resources, a model that decouples memory accesses from execution is effective when compared to other models that do not take shared resources into account.

1.5 Contributions

Considering the problems and questions above, this research work proposes the following contributions. For the first problem and first question, two contributions are proposed. The first contribution, presented in Chapter 3, considers a parallel task model that generalises the fork-join model presented above, known as the synchronous task model. Under this model, the worst-case scenario is derived in order to compute the worst-case response-time bounds for multiprocessor systems composed of identical processors. The second contribution, presented in Chapter 4, takes advantage of work-stealing [Blumofe and Leiserson, 1999] to reduce the average response-time of real-time tasks in order to create additional room in the schedule for less-critical tasks. The presented approach is a multi-stage approach that analyses the schedulability of the real-time tasks before and during runtime.

For the second problem and second question, the contribution proposed, presented in Chapter 5, uses a task model known as the 3-phase task model. In this model, memory accesses are decoupled from execution in order to circumvent the uncontrolled sources of interference, occurring due to co-running tasks in multiprocessor systems. An empirical analysis is carried first to compare the performance of different priority assignment policies against an implementation of global Earliest Deadline First (EDF) scheduling policy that considers inter-task interferences. Then, a schedulability test for the 3-phase task model is derived using a different analysis perspective. Instead of analysing the system following the standard's core's perspective, a bus perspective is used.

1.6 Thesis Structure

The remainder of this dissertation is structured as follows:

Chapter 2 details the most important properties about the platform and real-time tasks that are relevant for this dissertation. Moreover, it reviews the most important results found in the literature regarding the problem of scheduling parallel real-time tasks and the problem of memory bus contention.

Chapter 3 presents the schedulability analysis of fixed-priority synchronous parallel tasks executing in homogeneous multiprocessor systems.

Chapter 4 presents an approach that takes advantage of the work-stealing algorithm in a semi-partitioned scheduling setting for scheduling fork-join tasks. The proposed approach is a multi-stage approach that consists of an offline stage and an online stage. During the offline stage, tasks are mapped to cores so as to fill the capacity of the cores as much as possible. During the online stage, a variant of work-stealing is used among cores to balance the workload and consequently, to reduce, whenever possible, the response-time of the tasks that were accepted offline. The schedulability analysis for the approach is presented as well as the experimental results showing its viability.

Chapter 5 is focused on the problem of memory bus contention in real-time systems. It presents an empirical study that compares the performance of different priority assignment policies considering the 3-phase task model. Then, a schedulability test for the 3-phase task model is derived considering an approach that analyses the system from a bus perspective instead of following the common core's perspective. Results show that memory bus contention is a relevant problem in current multiprocessor platforms and that the 3-phase task model is a viable model to circumvent it.

Chapter 6 completes this dissertation by presenting some concluding remarks about the research work presented in this manuscript and outlining future work.

Chapter 2

Background and Related Work

A real-time system is designed (in its simpler form) to capture events from the environment using sensors and respond to those events in a timely manner through actuators (*i.e.,* in a generic sense, we denote the system that is being monitored and controlled as *controlled system*). In the context of real-time systems, timely manner means before a *deadline*, which is the maximum time within which a response must be produced. In these systems, the response time of the system, that is (in a non-formal manner) the time that it takes to fully respond to an input event or stimulus, is an important metric to consider as its performance may be affected depending on how long it takes to respond to the stimulus. In some cases, if the response time takes more time than the one expected at design time, catastrophic consequences may occur.

This chapter covers the main concepts related to the theory of real-time systems. It starts by characterising real-time tasks in Section 2.1, followed by the characterisation of the platform in Section 2.2. Concerning the platform, the main components are covered, namely the processors, the memory and the memory bus. Next, as scheduling is used throughout this dissertation, we devote our attention to multiprocessor scheduling theory in Section 2.3 to convey the concepts needed for the reader in order to understand the contributions of this dissertation. Finally, as this dissertation is also partly focused on the scheduling of parallel real-time tasks, we present in Section 2.4 the most relevant properties of parallel real-time systems.

2.1 Task Characterisation

A real-time application is modelled as a set of *tasks*, commonly denoted as τ. Each task τ_i in the set τ has functional and timing requirements (among other non-functional requirements) that must be guaranteed during runtime so that the result of its execution is deemed correct.

A real-time task can be classified as *hard*, *firm* or *soft*, according to the time instant at which its response should be completed. If a response to an event should always occur within a time period no greater than its deadline, then the task is classified as a *hard* real-time task. In this case, having a result after the deadline may cause catastrophic consequences for the controlled system. Hard real-time tasks should be guaranteed before the execution of the system, by using

offline schedulability analysis techniques. A task is classified as *firm* if producing a response to an event after the deadline is useless for the controlled system. In this case no serious consequences may result from the deadline miss. Finally, a task is classified as a *soft* real-time task if deadline misses are tolerated, as long as they are bounded. For this latter case, in case a deadline miss occurs, the output value still presents some utility for the system, however the controlled system suffers a performance degradation. Both firm and soft real-time tasks can be guaranteed by using schedulability analysis techniques that can be applied during the execution of the system.

Each new release of a task is denoted as a *job* meaning that a new instance of such task is being released into the system for execution. Tasks may have different release patterns, according to the frequency at which they are released into the system. Thus, tasks can be classified as *periodic*, *sporadic* or *aperiodic*.

A task that has periodic releases is denoted as a *periodic* task. Periodic tasks are characterized by a *period*, usually denoted as T_i, which indicates the frequency of release of each of its jobs. It is a common assumption found in the literature that during runtime a periodic task may release a potentially infinite sequence of jobs, where each job is released T_i time units apart from each other. In a *sporadic* task, the release of each of its jobs is separated by a *minimum inter-arrival interval*, also denoted as T_i. The interpretation for this parameter is that consecutive jobs of a sporadic task are *at least* separated T_i time units apart from each other. Nevertheless, at runtime the frequency of release may be larger than T_i. Similarly, as it happens to periodic tasks, the number of jobs released by sporadic tasks may potentially be infinite. Finally, in *aperiodic* tasks, task releases do not follow a well-known pattern and therefore do not have a period or a minimum inter-arrival time.

If all tasks in the set τ are released in the system at the same time instant, then the tasks' release is denoted as *synchronous*. On the other hand, if tasks are released at different time instants (*e.g.,* separated by some time offset), then the tasks' release is denoted as *asynchronous*.

Besides the period, there are two other important parameters that are used in the definition of real-time tasks, namely the worst-case execution time and deadline.

Several aspects influence the execution time of a task, as for instance its inputs, the scheduling algorithm, the platform, among others. The impact of these aspects during each task's execution can be observed in the duration of each of its jobs, that is, different jobs may have different execution times. Thus, in order to keep the system predictable throughout its execution, the *worst-case execution time* (WCET), denoted as C_i, is used. The task's WCET is an upper-bound on the time that it takes to execute the task in isolation in a given hardware platform.

The *deadline*, usually denoted as (D_i), represents the time instant at which the job of a task must complete its execution. Tasks can be further characterized according to the relation that exists between the deadline and the period. A task has a *constrained deadline* when its deadline is no greater than the period ($D_i \leq T_i$). A special case of constrained deadline tasks is known as *implicit* deadline tasks and occurs when the task's deadline equals its period ($D_i = T_i$). Finally, a task is

said to have an *arbitrary* deadline if there is no restriction on the value of the deadline D_i.[1]

Two other task properties can be defined, namely *utilization* and *density*. The *utilization* of task τ_i, denoted as U_i, is defined as $U_i = \frac{C_i}{T_i}$. The task's utilization represents the percentage of time the task is allocated to a given processor by executing C_i time units every T_i time units. The *density* of task τ_i, denoted as λ_i, is defined as $\lambda_i = \frac{C_i}{\min(D_i,T_i)}$. The task's density represents the percentage of time the task is allocated to a given processor by executing C_i time units every D_i time units. While for implicit deadline task sets the density equals the utilization of the task, for constrained deadline task sets the result is different. These two properties can be also defined for task sets. Thus, the *total utilization* of a set of n tasks τ is defined as $U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_i$. The total utilization of a task set represents the percentage of time the processor is allocated for the execution of the n tasks given that each task executes for C_i time units every T_i time units. Therefore, it indicates the minimum capacity that the platform must provide in order to execute the task set τ. The *total density* of a set of n tasks τ is defined as $\lambda_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \lambda_i$.

A task set τ is said to be *feasible* if there exists some scheduling algorithm that can schedule all possible job sequences released by the tasks belonging to τ, without missing any of the task's deadlines. A *feasibility test* determines if the task set τ executing on a given platform Π is feasible on that platform.

A task is said to be *schedulable*, with respect to a given scheduling algorithm, if it completes execution before its deadline when scheduled using that scheduling algorithm. In other words, the scheduling algorithm guarantees that the worst-case response time of the task is no greater than its deadline. The schedulability of task sets with respect to a scheduling algorithm and a platform can be evaluated through a *schedulability test*. That is, a *schedulability test* determines if a task set τ scheduled using a scheduling algorithm S in a given platform Π is schedulable using S on Π.

A schedulability test can be *sufficient*, *necessary* or *exact*. A *sufficient* test implies that if the test is passed, then the task set is schedulable, however if the test is not satisfied then the task set under evaluation may be schedulable or not. Nothing can be concluded from the test and another test shall be used. A *necessary* test entails that if the test is passed, then the task set may be schedulable but not necessarily. However, if the test is not passed, then the task set is certainly not schedulable. An *exact* test is both necessary and sufficient.

2.2 Platform Characterisation

In this section, the most important hardware components are described, *i.e.,* the ones that have influence in this dissertation's contributions and somehow, the ones that have the most impact in the execution of a real-time task. We start by introducing the notion of processor, core and multiprocessor system, the memory, and finally, we discuss the influence that the memory bus has in the execution of real-time systems.

[1] Without loss of generality, in this dissertation all time intervals and task parameters are assumed to be integer multiples of the system clock.

2.2.1 Processors

Several terms are used as synonyms for the central processing unit (CPU) in a computing system, as for instance processor or core, sometimes causing confusion. Specially when these terms are generalized to include multiple processing units, such as multiprocessors or multicores. Thus, in order to avoid confusion we clarify each of these terms in this section.

Originally, the CPU was a processor chip made of millions of transistors containing a single processing unit and a few other units to perform several operations (*e.g.,* arithmetic, logic, *etc.*). Consequently, the term processor is used as a synonym for CPU.

In a multiprocessor system, the platform contains several CPUs, each in a physical chip. However, with the advent of multicore systems, each chip started to include more than one processing unit *per* chip. Thus, in this configuration, each processing unit is denoted as a *core*. As it was explained in the introductory chapter, the reason for such a paradigm shift had to do with the physical limitations of chips. Hence, the industry opted to increase the parallelism provided in a single chip by the inclusion of several cores instead of increasing the processing speed of each processing unit (in a single chip).

In this dissertation, the term processor and core are used interchangeably and as a synonym for a single processing unit in the system.

Multiprocessor systems can be classified into three classes according to the characteristics of the processors present in the platform. If each processor in the platform presents the same computing capacity, *i.e.,* meaning that the frequency is equal in all processors, then the platform is said to have *identical* or *homogeneous* processors (as for instance [Bell et al., 2008] and [Adapteva, 2014]). In this class, all processors are interchangeable as a task takes the same amount of time to complete its WCET (in isolation) in every processor. If different processors in the platform have different computing capacities, the platform is said to be composed of *uniform* or *related* processors. In this case the rate of execution of a task depends on the frequency of the processor in which it executes. Finally, there is the *heterogeneous* or *unrelated* processor class. In this class, processors are different among themselves and consequently, the execution time of the tasks may differ between processors and some tasks may not be able to execute in all/some processors. An example of a platform belonging to this class is the MPPA-256 Manycore Processor developed by Kalray [de Dinechin et al., 2013].

In this dissertation we only consider identical multiprocessor platforms.

2.2.2 Memory

In the past years dynamic random access memory (DRAM) speed did not increase in the same proportion as CPU speed. In fact, the achieved improvement is much less than the one observed in the CPU. The difference in speed between both components leads to a speed gap that eventually causes memory accesses (even in those cases where a program is composed of only a few memory instructions) to dominate the total time spent executing a program. This phenomenon was predicted in [Wulf and McKee, 1995] and is known as the *memory wall*.

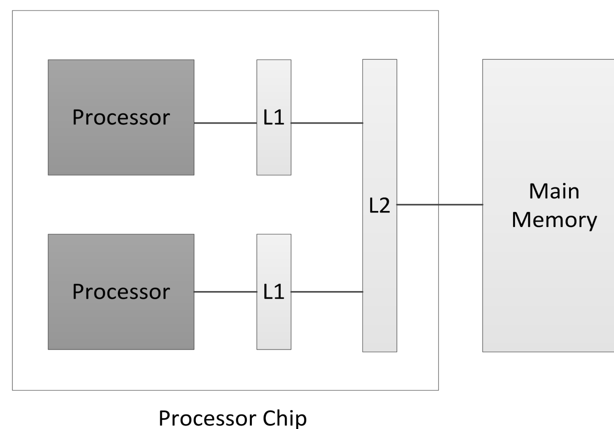


Figure 2.1: Memory hierarchy in current COTS platforms

In order to overcome the limitations of such a small increase in memory speed and at the same time attempt to decrease the memory access latencies, a hierarchical approach is used in modern computing systems.

The memory hierarchy is organized in several levels where each level is smaller in size and faster than the subsequent level in the hierarchy (when moving away from the processor chip). The faster memories that are closer to the processor are denoted as *caches* and, usually, are located within the processor chip. Caches act as a buffer for the data residing in main memory[2] and work according to the principle of locality which states that programs are likely to reuse data and instructions they have used recently.

Let us use Figure 2.1 to explain the reader how data traverses the memory hierarchy. The hierarchy in the figure has 3 levels, cache L1 (private to the core connected to it), cache L2 (shared between the cores connected to it) and finally, main memory (shared by all cores in the system). In our example, we assume a task is running in a single processor and it does not migrate to the other processor. When executing a task, the processor looks first for program data/instructions in cache L1. If data/instructions are found in that cache then there is a *cache hit* and the processor can use them without requesting data/instructions from the other levels. Otherwise, a *cache miss* occurs (data/instructions are not in the cache) and a request is made to the subsequent level. In our example, that level is L2. Then, the process is repeated. A hit or a miss may occur. If a hit occurs data is moved to level L1, otherwise a data request is made to main memory in order to be retrieved and stored temporarily in the cache.

The memory hierarchy is developed to improve the average memory access time and consequently, the average execution time of programs. However, such design approach, which is typically found in Commercial Off-the-shelf (COTS) platforms, is a source of unpredictability in the context of real-time systems. Cache properties like data replacement strategy, size, organization and access order influence the cache hit/miss ratio and make it difficult to predict cache

[2]The main memory is usually found outside the processor chip.

behaviour. Besides these properties, one needs also to consider the interference due to task *pre-emptions* (when one task is interrupted during execution to allow another, usually more urgent, task to execute) and *migrations* (when one task resumes its execution in a different core than the one where it was executing before being interrupted) which incur additional cache misses.

In this dissertation, we assume that the platform follows an hierarchy equal to the one depicted in Figure 2.1. However, caches and main memory are treated as black boxes. In fact, in the literature it is possible to find several works that are devoted to the topic of adding predictability to caches. The interested reader may look for the following work as a starting point [Gracioli et al., 2015].

2.2.3 Memory Bus

Cache unpredictability is not the only problem that is found when using a memory hierarchy similar to the one depicted in Figure 2.1. There is another important problem that arises from the way that memory is designed. If one looks carefully to the memory hierarchy depicted in Figure 2.1, one may easily see that with the exception of private memory levels, multicore processors share paths to the different levels in the memory hierarchy. These paths are part of the system memory bus and because they are shared, they may lead to contention when more than one core simultaneously perform a memory request to the same shared level of memory. For instance, when a memory request is made to main memory or even level L2.

In order to avoid the undesired effects of bus contention, COTS manufacturers add arbitration mechanisms to the system memory bus. Nevertheless, the arbiters employed in general-purpose systems are: (1) often undocumented and their implementation is hidden; (2) not controlled by the operating system and consequently, the exact time instants at which the memory requests are made are unknown as they are a result of cache misses; and (3) unfair and consequently may re-order memory requests (subject to the arbiter's own rules) and neglect task priorities (which are defined at the operating system level) in order to optimize, for instance, memory bandwidth [Dasari et al., 2013]. Thus, these arbitration mechanisms have a direct impact on system performance and the response time of tasks. In fact, if their behaviour is not accounted for in the WCET analysis of the tasks composing a system, the actual worst-case time observed at runtime may drastically deviate from the predictions made at design time.

The memory bus contention problem is a well-known problem in the real-time systems community and several authors have already devoted their efforts to it. In the following paragraphs, we cover the most relevant work that has been done in the research of the memory bus contention problem.

Deterministic architectures (such as MERASA [Ungerer et al., 2010], PRET [Lickly et al., 2008]) consisting of mechanisms to control interference at the hardware level have already been proposed in the past. Nevertheless, this type of solutions is very specific, leading the stakeholders (usually due to the costs involved in the development of specific hardware platforms) to adopt general-purpose platforms to implement their products.

2.2.3.1 Time-Driven vs. Event-Driven Approaches

Arbitration approaches can be classified into two distinct classes [Abel et al., 2013]: time-driven and event-driven.

Time-driven approaches, such as the ones proposed in [Kelter et al., 2011], [Chattopadhyay et al., 2010], [Schranzhofer et al., 2010], employ Time-division Multiple Access (TDMA) as the bus arbitration policy. The idea behind these approaches is to time-partition the access to the bus into time slots and generate a bus schedule. At runtime, the arbiter uses the generated bus schedule to grant permission to a given core to access the bus. A core is allowed to access the bus if the current time slot is assigned to that core, otherwise the core has to wait until the next available time slot that is assigned to it. TDMA-based approaches provide temporal isolation between cores and thus have the advantage of allowing each core to be analysed in an independent manner. As each core may only perform memory requests in its assigned time slot, cores cannot interfere with each other. However, for memory operations to be efficient there must be an alignment between memory requests and each core's assigned slots, otherwise many slots may be wasted.

Event-based approaches provide bounds on the interference that a resource may suffer in a worst-case scenario by knowing the maximum number of memory accesses that a task may request and the arbitration policy of a given resource.[3] Some works in the literature ([Pellizzoni et al., 2010], [Schliecker et al., 2010]) use the concept of arrival curves [Thiele et al., 2000]. For instance, in [Pellizzoni et al., 2010] arrival curves are used to model the maximum amount of memory traffic produced by all tasks executing in a given core in a given time interval. Then, the derived curves are used to compute bounds on the delay incurred by a given task considering the arrival curves derived for the cores not executing the analysed task and peripheral buses. In [Schliecker et al., 2010], arrival curves are used to model the load of each processor in the system. Other works, as for instance [Ivers et al., 2006], estimate the maximum delay a task may suffer due to memory interference when executing in a system where resources are shared.

2.2.3.2 Co-Scheduling Approaches

Co-scheduling approaches have also been proposed to circumvent the memory bus contention problem. The idea behind such approaches is to decouple memory requests from the actual task's execution such that all the code and data needed during execution are loaded in a core's local memory before beginning the task's execution. By pre-loading the task's code and data in the core's local memory, a core can execute the task without suffering any kind of interference.

Co-scheduling was the target of research in [Schranzhofer et al., 2010] where the authors analyse different resource access models. In particular, depending on the studied model, accesses to shared resources either occur in specific phases (as in a pure co-scheduling approach) or occur, without any restriction, throughout the task's execution. The objective of that study was to evaluate which model performs better, considering the interaction with shared resources, in terms of worst-case response times and schedulability. The conclusion is that a model with 3 phases is the one that

[3] In this context, a worst-case scenario is a scenario that maximizes the effects of interference on memory requests.

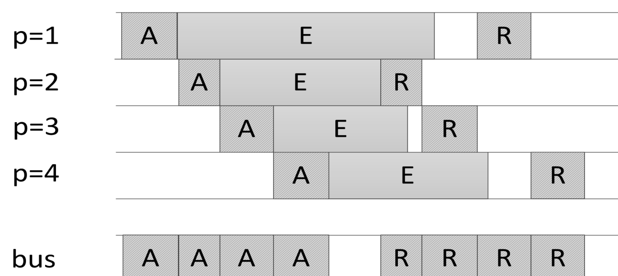


Figure 2.2: Example of a 3-phase task model schedule

performs best when compared to models where no restriction is posed on the accesses to shared resources.

The 3-phase task model is a generalization of the PRedictable Execution Model (PREM) [Pellizzoni et al., 2011]. In PREM, tasks consist of only two phases (known as the predictable intervals): a memory phase and an execution phase. In the memory phase, tasks fetch data and instructions from main memory into the core's local memory while in the execution phase, tasks execute without requiring any access to the shared memory and thereby minimizing any possible interference during their execution. The 3-phase task model generalizes PREM by adding a third phase in which the modified data is pushed back from the core's local memory into the main memory. Moreover, tasks that follow this execution model never access the bus during their execution phase, instead, all the bus accesses are performed during the first and third phases. Figure 2.2 depicts a schedule where four tasks execute in a multiprocessor platform, each executing in a processor. The memory phases (A and R, where A stands for *Acquisition* and R stands for *Restitution*) require the use of the bus while the execution phases (in the figure represented by the letter E) do not require any access to the memory bus.

The 3-phase task model has been subject to experiments carried out to evaluate the applicability of the model in different domains. In [Durrieu et al., 2014a], the authors use the 3-phase task model to model periodic tasks in a flight management system. Moreover, in [Girbal et al., 2015], the authors show that executing tasks in a multicore system leads to increases in the WCET measured in isolation of up to 3x the value in isolation, and that by using the 3-phase task model it is possible to obtain an interference-free execution in a multicore system. A similar observation was made in [Nowotsch and Paulitsch, 2012] where the authors evaluate the effects of having multiple applications of different criticality levels executing in a multicore platform. More precisely, the authors observed a maximum slowdown of 5.1x in application execution when multiple cores access network and memory concurrently. Both of these results show that special care must be taken when executing safety-critical applications in multicore platforms due to the increase in WCET as a result of interference related to concurrent accesses to shared resources. A similar result to both of these works is presented in Chapter 5.

In [Becker et al., 2016], the 3-phase task model is applied to AUTOSAR applications in order to obtain a contention free execution in a many-core architecture. In [Tabish et al., 2016], the authors integrate the 3-phase task model with TDMA managed accesses to a system bus, as a

way to serialize memory phases in multicore operating systems for embedded scratchpad-based multicore architectures.

None of the above mentioned works tackle the problem of how to globally schedule 3-phase tasks in a multicore system. To the best of our knowledge, the only work that provides a solution to this problem is the work presented by Alhammad and Pellizzoni [Alhammad, 2016; Alhammad and Pellizzoni, 2014]. In this dissertation, we fill that gap in the literature and propose in Chapter 5 a solution to this problem that improves the work proposed by [Alhammad, 2016; Alhammad and Pellizzoni, 2014].

2.3 Multiprocessor Scheduling

Before presenting the most important concepts of real-time multiprocessor scheduling, let us present some properties of scheduling algorithms that are useful for understanding some concepts proposed in this dissertation.

A scheduling algorithm is said to be *preemptive* if it is capable of suspending a job during execution and later resume it from the point where it was suspended. Usually, preemption operations occur due to the arrival of higher priority tasks into the system. A *non-preemptive* scheduling algorithm does not suspend tasks. Once the tasks are allocated into the processor, tasks execute continuously until completion, time instant at which another task is selected for execution. A scheduling algorithm is said to be *work-conserving* if it never idles a processor when there is a ready task[4] waiting to be executed. A scheduling algorithm is said to be *optimal* if it schedules all the task sets that are feasible and that abide by the task model.

Two important metrics are used to quantitatively compare different scheduling algorithms: the utilization bound and resource augmentation bound ([Kalyanasundaram and Pruhs, 1995] and [Phillips et al., 1997]).

The *utilization bound* of a given scheduling algorithm A on a platform Π, consisting of m unit-speed processors, is defined as the largest utilization U_b such that all implicit-deadline task sets composed of sequential tasks with utilization $U \leq U_b$ are deemed schedulable by A when executed in platform Π.

While the utilization bound is based on the utilization factor and therefore on the properties of the task set, a resource augmentation bound quantifies the processor speed-up factor with respect to an optimal scheduling algorithm. That is, it quantifies how much one has to increase the processor speed in order to guarantee the schedulability of a task set using a given scheduling algorithm A instead of an optimal one.

Formally, a scheduling algorithm A has a *resource augmentation bound* b on a given platform Π, consisting of m unit-speed processors, if it successfully schedules all the feasible task sets, which are schedulable by an optimal algorithm on Π, on a platform where the processors are b times as fast than the ones in Π.

Real-time multiprocessor scheduling theory deals with two problems [Davis and Burns, 2011a]:

[4] A ready task is a task that is waiting for access to the processor.

- the *allocation problem* - the decision problem of how a set of n tasks should be allocated on a set of m processors;
- the *priority problem* - the decision problem of choosing the order a set of tasks should follow so that each task's deadline is met.

Concerning the allocation problem, tasks can be classified according to the type of migration that their jobs are allowed to perform. In the most restrictive type, tasks are pinned to processors and no migration is allowed to occur during execution, meaning that all the task's jobs must execute in the processor where they were assigned initially. *Task-level migration* allows a task to migrate and execute on multiple processors but migrations may only occur at job-boundary. Finally, *job-level migration* allows jobs to migrate to other processors during execution, but it is forbidden for a job to execute simultaneously on different processors.

Regarding the priority problem, tasks have *fixed task priority* if each task is assigned a priority and the same priority is applied to all its jobs; *fixed job priority* when each job of a task may have a different priority, but the priority of a job does not change until the job finishes its execution; and *dynamic priority* when the priority of the jobs may change during execution.

Two paradigms are usually used to distinguish real-time multiprocessor scheduling algorithms [Carpenter et al., 2004]: *partitioned* and *global* scheduling.

In *partitioned scheduling*, each task is assigned to a processor and is not allowed to migrate[5] among cores. Each processor has its own subset of tasks to execute and is treated independently with respect to task scheduling. Thus, each processor uses a uniprocessor scheduling algorithm to schedule the tasks assigned to it. Consequently, different algorithms may be in use during system execution, one *per* core. Due to the restrictions of no migration, partitioned approaches are not work-conserving.

In *global scheduling*, a global scheduler selects the next task from a global queue of ready tasks, that is shared by all processors, and assigns it to an idle core. At any given time instant t, the m higher priority tasks are assigned to m cores. Tasks are allowed to migrate among the cores.

Partitioned scheduling has the advantage of reducing the problem of real-time multiprocessor scheduling to a set of uniprocessor problems by treating each processor independently. This is advantageous as it reduces the number of migrations occurring in the system and consequently, runtime overheads are also reduced. The biggest disadvantage of partitioned approaches is that they require the use of bin packing in order to optimally assign tasks to processors, a problem that is known to have NP-hard complexity.

Global scheduling offers the advantage of using the system resources in a more effective manner but incur higher runtime overheads due to task migration and contention in the global runtime queue.

[5] Task migration causes overhead due to the need of reloading the task's instructions and data into the core's local memory where the task migrated to. In the worst-case it means that task's instructions and data have to be fetched from the main memory. Even though the migration overhead may be included in the WCET of the migrating task, in practice this may affect the performance of the system [Bastoni et al., 2011].

Concerning schedulability, in the worst case, partitioned scheduling algorithms (even for an optimal algorithm) cannot guarantee that task sets with utilization greater than $\frac{m+1}{2}$ are schedulable on a platform with m cores [Carpenter et al., 2004]. This result means that nearly fifty percent of the platform may be unused. This utilization bound is also known as the maximum utilization bound for global fixed-task, fixed-job priority scheduling algorithms for implicit-deadline task sets executing on a platform with m homogeneous cores [Andersson et al., 2001]. Moreover, Leung and Whitehead [Leung and Whitehead, 1982] prove that partitioned and global approaches for fixed-priority task scheduling are incomparable. Thus, there are task sets that are feasible under partitioned scheduling that are not under global scheduling and vice-versa.[6]

Both global and partitioned approaches suffer from scheduling anomalies where favourably changing some parameters of the tasks, as for instance the computation times or periods, may cause problems in terms of schedulability of previously feasible task sets. For instance, in [Andersson and Jonsson, 2000] the authors provide examples for preemptive global scheduling. In a particular example, they show that changing the task period (and therefore varying the processor load) of higher priority tasks leads to an increase in the interference suffered by a lower priority task, or even to the unschedulability of the system. Many examples of such favourable modifications that lead to anomalous behaviour exists.

An alternative approach to multiprocessor scheduling combines features of both partitioned and global scheduling as a way to improve the utilization bounds of partitioned scheduling algorithms. This approach is denoted as *semi-partitioned* scheduling ([Anderson et al., 2005], [Andersson and Tovar, 2006], [Kato et al., 2009]). In this approach, there is an offline allocation of tasks to processors as in partitioned scheduling, however some tasks (those that cannot be assigned to a single processor due to use of bin packing) are allowed to migrate between different processors, thus having a global scheduling behaviour.

2.4 Parallel Real-Time Systems

This dissertation also considers the scheduling of parallel real-time tasks. Thus, in order to provide the reader with the background needed to understand the contributions being proposed, this section and its subsections detail the most relevant properties of parallel real-time systems.

2.4.1 Parallel Task Models

As stated in the introductory chapter, the major property that parallel task models for real-time systems try to take advantage of is intra-task parallelism. Opposed to inter-task parallelism[7], intra-task parallelism allows simultaneous execution of tasks by dividing a task in a set of sub-tasks on several cores in parallel. In order to keep up with the pace, the real-time systems community had to

[6] For more information about schedulability results for different algorithms used for partitioned and global scheduling the interested reader may consult the following survey [Davis and Burns, 2011a].

[7] Recall that with inter-task parallelism, parallelism can only be exploited by increasing the number of tasks executing in the system and increasing the number of cores does not increase the execution speed for each task.

adapt to the new hardware trends and was forced to develop new models to cope with parallelism in real-time systems for the sake of efficiency. Consequently, it is already possible to find a few models and results concerning the multiprocessor scheduling of parallel real-time tasks.

Several models exist: the fork-join model, the synchronous task model and directed acyclic graph (DAG) model (a general model of parallel tasks)[8]. For each of these models, the following two necessary conditions hold: (i) the utilisation of an individual task can be greater than 1 but it has to be no greater than the number of processors available in the platform; (ii) the critical path length of the parallel task should always be no greater than the task's deadline, for all tasks in the task set under the penalty that a task does not complete its execution within the deadline.

2.4.2 Earlier Parallel Models

In this section initial results (yet applicable) to tackle parallelism in real-time systems are presented, while more recent models are presented in the subsequent subsections.

Drozdowski [Drozdowski, 1996] considers the problem of scheduling parallel tasks with the objective of minimising the makespan. Han and Lin [Han and Lin, 1989] prove that the problem of scheduling parallelisable jobs with a fixed priority is NP-Hard.

Goossens and Berten [Goossens and Berten, 2010] redefined a classification from the parallel literature. Following this classification, a job may be classified as *rigid*, *moldable* or *malleable*. A job is *rigid* if the number of processors assigned to it is determined *a priori*, and this number does not change throughout execution. A job is said to be *moldable* if the number of processors assigned to it is determined by the scheduler, and it does not change throughout execution. Finally, a job is said to be *malleable* if the number of processors assigned to it is determined by the scheduler at runtime. Taking into account this classification, a task is said to be rigid if all of its jobs are rigid; moldable if all of its jobs are moldable; and malleable if all of its jobs are malleable.

Considering works on moldable tasks, Manimaran et al. [Manimaran et al., 1998] proposed a variant of non-preemptive Earliest Deadline First (EDF) that considers parallel real-time tasks. Kato and Ishikawa [Kato and Ishikawa, 2009] proposed the Gang EDF algorithm, which applies EDF to the traditional gang scheduling scheme.

Concerning rigid tasks, Goossens and Berten [Goossens and Berten, 2010] not only provided the above-mentioned classification for parallel real-time tasks but also proposed a scheduling algorithm for parallel rigid real-time tasks based on gang scheduling.

Malleable tasks were covered by Jansen [Jansen, 2002], Collette et al. [Collette et al., 2008], and Korsgaard and Hendseth [Korsgaard and Hendseth, 2011]. Jansen [Jansen, 2002] focused on minimising the makespan but without considering real-time constraints. Collette et al. [Collette et al., 2008] studied the problem of global scheduling of sporadic task systems on multiprocessors

[8] Several works, as for instance [Bonifaci et al., 2013; Liu and Anderson, 2010; Saifullah et al., 2014] are addressing the directed acyclic graph model. The study of tasks' schedulability under this model is out of scope of this dissertation. The interested reader is redirected to the mentioned works.

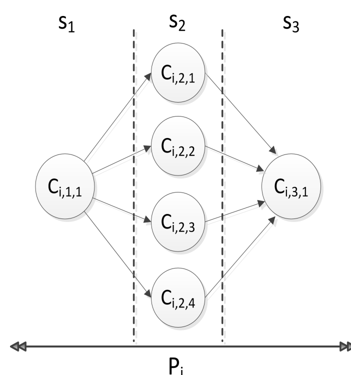


Figure 2.3: Example of a fork/join task τ_i. This task has two sequential segments (s_1 and s_3) with one thread each, and one parallel segment s_2 composed of 4 threads.[10]

considering job-level parallelism. Korsgaard and Hendseth [Korsgaard and Hendseth, 2011] proposed a sustainable schedulability test for malleable tasks scheduled with global Earliest Deadline First (EDF)[9].

2.4.3 Recent Parallel Models

In this section, we devote our attention to the works proposed in the literature that are strictly related to the contributions of this dissertation. Thus, we present the most important works that focus on the fork-join task model and synchronous task model.

2.4.3.1 Fork-Join Parallel Tasks

The fork-join task model, depicted in Figure 2.3, is a model used by some frameworks (as for instance [OpenMP, 2011], [Oracle, 2011], [Frigo et al., 1998]). In its basic form, the job of a task has two sequential segments and a parallel segment. But, in fact and generally speaking, the fork-join model imposes a restriction in which each parallel segment should always be preceded by a sequential segment and succeeded by another sequential segment. Sequential segments have a single unit of execution and the parallel segments are composed of several independent threads that are allowed to execute in parallel if the platform allows.

Lakshmanan et al. [Lakshmanan et al., 2010] study the scheduling of periodic fork-join real-time tasks on multiprocessor platforms. In their model, each task is divided into sequential and parallel segments. Parallel segments must be preceded and followed by a sequential segment. All parallel segments must have the same number of threads, and the number of threads cannot be greater than the number of processors in the platform. In order to schedule such tasks in a multiprocessor platform, the authors propose the decomposition of fork-join tasks using the *task*

[9]Global EDF is the extension of the Earliest Deadline First algorithm to homogeneous multiprocessor systems. The EDF algorithm [Liu and Layland, 1973] for single core assigns the highest priority to the job that has the earliest deadline among all the jobs ready to execute. The global version of the algorithm considers for execution, at any time t, the m ready jobs with earliest deadline on a platform with m cores.

[10]P_i represents the critical path length of the task. Its definition can be found in the next chapter.

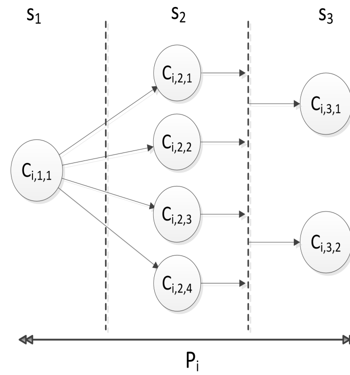


Figure 2.4: Example of a synchronous parallel task τ_i. This task has a sequential segment (s_1) with one thread, and two parallel segments s_2 and s_3 composed of 4 threads and two threads respectively.

stretch transform algorithm. Then, for the decomposed task set, a resource augmentation bound of 3.42 is obtained when the task set is scheduled using partitioned Deadline Monotonic[11].

In [Wang and Parmer, 2014], the Fork-Join OS (FJOS) is presented. FJOS is an operating system based on Composite OS, and its behaviour is compared with the GOMP [FSF, 2014] implementation on Linux. Moreover, the schedulability analysis technique proposed in [Axer et al., 2013] is adapted to include overheads based on real measurements in FJOS. As in [Axer et al., 2013], such an approach is also based on partitioned fixed-priority scheduling for real-time systems.

2.4.3.2 Synchronous Parallel Tasks

Saifullah et al. [Saifullah et al., 2011] generalise the fork-join model presented in [Lakshmanan et al., 2010], denoted as synchronous task model. In the synchronous parallel task model, tasks are composed of several segments, each containing one or more independent threads. Segments have precedence constraints among themselves, and within a segment all threads are released simultaneously and may execute in parallel. Moreover, the threads belonging to a segment can only start their execution after the threads in the previous segment finish theirs, thus creating a synchronization point. In this model (depicted in Figure 2.4), there is no restriction on the number of segments per task, and on the number of threads per segment. To analyse the schedulability of the model, the authors in [Saifullah et al., 2011] propose an algorithm to decompose implicit-deadline parallel tasks into constrained-deadline sequential tasks. For the decomposed task sets they derive resource augmentation bounds of 4 and 5 for the global Earliest Deadline First (EDF) scheduling algorithm and partitioned deadline monotonic, respectively.

The authors in [Chwa et al., 2013] analyse the behaviour of synchronous parallel real-time tasks under global EDF. In particular, they derive a schedulability condition by extending the

[11] Deadline Monotonic is a scheduling algorithm that assigns a fixed priority to each task which is inversely proportional to its relative deadline D_i. At any time instant, the task that has the shortest relative deadline is the one selected for execution.

traditional interference-based analysis to accommodate the parallel behaviour of the tasks. The concept of critical interference is introduced in order to capture the interference of parallel threads within the segments.

In Chapter 3, we borrow the concept of critical interference from [Chwa et al., 2013] to propose tighter schedulability conditions for the fixed-priority scheduling of synchronous parallel tasks.

Chapter 3

Schedulability of Synchronous Parallel Tasks

3.1 Introduction

Having parallelism at the platform level allows real-time systems developers to support applications with higher complexity. However, as it was explained in Chapter 1, higher complexity may require that applications take advantage of intra-task parallelism in order to improve their execution times, possibly with tighter timing constraints.

Intra-task parallelism can be harnessed by splitting a task into a set of sub-tasks that can be executed simultaneously in different processors at the same time instant (*i.e.,* potentially overlapping in time). Models such as the fork-join model or its generalization, the synchronous task model, are good candidates for harnessing intra-task parallelism in real-time systems.

In this chapter we focus on the schedulability analysis of fixed-priority synchronous parallel tasks executing in homogeneous multiprocessor systems. Tighter upper-bounds on the workload within a window of interest are derived which allows one to compute response-time upper bounds of the interfering jobs, similarly to the technique proposed in [Bertogna and Cirinei, 2007] for sequential task sets. The presented approach improves over the work reported in [Chwa et al., 2013], providing tighter schedulability conditions and extending the analysis to fixed-priority task systems.

The chapter starts by detailing the model and the assumptions used throughout the chapter, in Section 3.2. After presenting the system model, we introduce the notion of critical interference in parallel real-time tasks in Section 3.3. Using this notion, we proceed to the response-time analysis in Section 3.4 by introducing two techniques, the sliding window technique (Section 3.5) and the decomposition of the carry-out (Section 3.6), which allows one to find the densest possible packing of jobs of a parallel task in an interval of time. All the workload terms needed for the schedulability condition are described in Section 3.7 and the condition itself in Section 3.8. Finally, the results are presented in Section 3.10.

3.2 System Model

Let $\tau = \{\tau_1, ..., \tau_n\}$ denote a set of n synchronous parallel sporadic tasks. Each task τ_i in τ releases an infinite sequence of jobs that are allowed to execute in more than one core at the same time instant and are separated by at least T_i time units. Each task has a deadline $D_i \leq T_i$ (*i.e.,* commonly referred to as constrained deadline model), meaning that each of its jobs needs to complete its execution at most D_i time units after its release.

In addition, each task τ_i is characterised by a sequence of segments $s_i = \{\sigma_{i,1}, ..., \sigma_{i,s_i}\}$, where each segment $\sigma_{i,j}$ is composed of a set of $m_{i,j}$ parallel jobs, $\{J_{i,j,1}, ..., J_{i,j,m_{i,j}}\}$, each one having the same priority as the task that spawns it.

Parallel jobs, or in short p-jobs, are independent sequential threads that may be executed in parallel, *i.e.,* in different processors at the same time instant. Before a segment starts executing any of its p-jobs, all the p-jobs of the preceding segment (if any) must have been completed. That is, for all $\sigma_{i,\ell}, \sigma_{i,r} \in s_i$ such that $\ell < r$, the sub-tasks belonging to $\sigma_{i,r}$ cannot start executing unless those of $\sigma_{i,\ell}$ have completed. Other than the processing units and segment precedence constraints we assume no other shared resources exist in our system.[1]

As mentioned in the previous chapter, our platform $\pi \stackrel{\text{def}}{=} \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprises m homogeneous cores, *i.e.,* all the cores have the same computing capabilities and are interchangeable.

In this work, similarly to the work proposed by Saifullah et al. in [Saifullah et al., 2011], we allow the number of p-jobs of a segment to be greater than the number of cores. That is, $m_{i,j}$ may be greater than m for some segment $\sigma_{i,j}$. We denote the maximum degree of parallelism of a task as m_i and define it as $m_i = \max_j\{m_{i,j}\}$.

Each p-job instance $J_{i,j,k}$ is characterized by a worst-case execution time $C_{i,j,k}$. The worst-case execution time $C_{i,j}$ of each segment $\sigma_{i,j}$ is given by:

$$C_{i,j} = \sum_{k=1}^{m_{i,j}} C_{i,j,k}. \tag{3.1}$$

Then, the overall worst-case execution time C_i of a task τ_i is defined as:

$$C_i = \sum_{j=1}^{s_i} C_{i,j}. \tag{3.2}$$

Both equations above represent the time it takes to execute a segment (Equation 3.1) or a task (Equation 3.2) in a dedicated single processor platform, *i.e.,* without any parallelism at all.

The *minimum worst-case execution time* P_i of a task τ_i is the time τ_i takes to execute when the number of processing units m is infinite, *i.e.,* the critical path length of task τ_i. Formally, P_i is defined as:

$$P_i = \sum_{j=1}^{s_i} P_{i,j}, \tag{3.3}$$

[1] A task which consists of a single sub-task in each of its segments is considered a sequential task.

where $P_{i,j}$ represents the worst-case execution time of the largest p-job(s) of segment $\sigma_{i,j}$. Formally,

$$P_{i,j} = \max_{k=1}^{m_{i,j}} \{C_{i,j,k}\}. \quad (3.4)$$

The *utilisation* U_i of task τ_i is the ratio between the task's overall worst-case execution time and period, $U_i = \frac{C_i}{T_i}$. For the task set τ, the *total utilisation* is defined as $U(\tau) = \sum_{i=1}^{n} U_i$.

The worst-case response-time of τ_i, denoted as R_i, is given by the maximum amount of time that elapses between the release time (r_i) of any job of τ_i and its completion time.

When dealing with parallel tasks several factors influence the computation of R_i, namely the inter-task and intra-task interferences (detailed further in the next sections); the precedence constraints between the segments of a parallel task; the degree of parallelism[2] of each segment; and the number of cores provided by the hardware platform. As it may be extremely difficult to derive the exact worst-case response time of a task considering all the above factors, a typical approach found in the real-time systems literature is to compute an upper bound R_i^{ub} on the response-time of task τ_i.

A fully preemptive system is assumed where any executing p-job may be preempted and resumed later without any cost. At any given instant, the m ready p-jobs with the highest priority are the ones executing in the cores. Ties are broken arbitrarily. Moreover, as we are dealing with fixed-priority task systems, we assume that tasks are indexed in priority order, with task τ_1 being the highest priority one.

Regarding task set feasibility, there are two necessary conditions for the feasibility of fork-join and synchronous parallel task models: (1) $U(\tau) \leq m$, which states that the total utilisation of the task set should not be greater than the number of cores in the system (m); and (2) $P_i \leq D_i$, which states that the critical path length of a task should not be greater than its deadline. Moreover, it is not guaranteed that parallel task sets with $U(\tau) \leq m$ are schedulable in a system with m cores as there exist task sets with a total utilization greater than and arbitrarily closer to 1 ($U \approx 1$) that are unschedulable in a system with m processor cores, as shown in [Lakshmanan et al., 2010].

As a final remark, it is important to note that with the synchronous parallel task model there may be feasible task sets in which some task has a utilisation larger than 1. With such tasks serialisation techniques are not possible as the derived sequential task would be clearly unschedulable.

Table 3.1 presents a summary of the important notation defined and used throughout this and the following chapter for quick reference.

3.3 Critical Interference of Parallel Tasks

Interference is an important concept widely used in real-time systems. For traditional sequential task sets, the interference a task τ_k suffers over an interval of length L, denoted as $I_k(L)$, is defined as the sum of all intervals of time in which τ_k is ready to execute but it cannot execute due to the

[2] Degree of parallelism is a metric that indicates the number of cores in a multiprocessor system actually executing a particular task in a given time period.

Table 3.1: Summary of notation

Symbol	Description
m	Number of processors in the platform
n	Number of tasks in the task set
τ	Set of periodic or sporadic tasks
U_i	Utilisation of task τ_i, *i.e.*, $\frac{C_i}{T_i}$
$U(\tau)$	Total utilisation of the task set τ
T_i	Period of task τ_i
D_i	Relative Deadline of task τ_i
C_i	Overall worst-case execution time requirement of τ_i
P_i	Minimum worst-case execution time of task τ_i
s_i	Number of segments in task τ_i
m_i	Maximum degree of parallelism of task τ_i
$C_{i,j}$	Overall worst-case execution time of segment $\sigma_{i,j}$
$P_{i,j}$	Minimum worst-case execution time of segment $\sigma_{i,j}$
$m_{i,j}$	Number of p-jobs within segment $\sigma_{i,j}$
$C_{i,j,k}$	Worst-case execution time of p-job $J_{i,j,k}$
r_i	Release time of a job of task τ_i
d_i	Absolute deadline of a job of task τ_i
R_i	Worst-case response time of task τ_i
R_i^{ub}	Upper-bound of R_i
L	Generic interval $[r_k, r_k + R_k^{ub}]$
$I_k(L)$	Critical interference on task τ_k in any interval L
$I_{i,k}(L)$	Critical interference of task τ_i on task τ_k in any interval L
$I_{i,k}^p(L)$	Critical interference of task τ_i on task τ_k with depth at least p in any interval L
$W_i^p(L)$	Workload of task τ_i of at least p p-jobs in any interval L

execution of other higher priority tasks in the system. In particular, the interference of a higher priority task τ_i over task τ_k over an interval of length L is denoted as $I_{i,k}(L)$, and is defined as the sum of all intervals of time in which τ_i is executing but τ_k is not, even though it is ready to execute. Intuitively, the interference that a task suffers cannot be greater than the total workload of the higher priority jobs.

Two types of interference need to be considered when dealing with synchronous parallel tasks, namely *inter-task* and *intra-task* interferences. Inter-task interference is the interference caused on a given job by other higher priority jobs executing in the system in a given time interval. This is the same as the standard interference widely used in traditional sequential models (we formally define it in Section 3.3). Intra-task interference is only related to parallel task models, and can be defined as the self-interference caused by the execution of parallel jobs of the same task instance.

In order to compute the interference of a parallel task, we adopt the concept of *critical thread*[3], as previously defined in [Chwa et al., 2013].

Definition 1. *A thread is* critical *if it is the last one to complete among the threads belonging to the same segment.*

For deriving the worst-case response time of a task, it is then sufficient to characterize the interference imposed to its critical threads, as they are the ones suffering the largest interference.

Definition 2. *The* critical interference $I_k(L)$ *on task* τ_k *in any interval of length* L *is defined as the cumulative time in which a* critical thread *of task* τ_k *is ready to execute but it cannot due to the execution of other parallel jobs.*

Given the above definitions, the following theorem simply follows.

Theorem 1. *Given a set of synchronous parallel tasks* τ *scheduled by any work-conserving*[4] *algorithm on* m *identical cores, the worst-case response-time of each task* τ_k *can be upper bounded by* R_k^{ub} *if*

$$P_k + I_k(R_k^{ub}) \leq R_k^{ub}. \tag{3.5}$$

Proof. Consider the job of τ_k that leads to the worst-case response time R_k. Let r_k be its release time. Within a scheduling window $[r_k, r_k + R_k^{ub}]$, Equation (3.5) guarantees that all s_k critical threads have sufficient time to execute P_k time-units, while accommodating the interference suffered from other threads, accounted for in $I_k(R_k^{ub})$. Since the execution requirement of each critical thread cannot exceed the minimum worst-case execution time of the corresponding segment, Equation (3.3) guarantees that all critical threads complete their execution within the considered interval, proving the theorem. □

The problem of the above theorem is that computing the exact interference imposed on the considered task is difficult (due to the different possible interleavings that tasks may have when executing in the system). To sidestep this problem, a common approach is to express the total interference as a function of individual task interfering contributions, and upper bound such contributions with the worst-case workload executed by each task in the considered window.

Definition 3. *The critical interference* $I_{i,k}(L)$ *imposed by task* τ_i *on task* τ_k *in any interval of length* L *is defined as the cumulative workload executed by p-jobs of task* τ_i *while a critical thread of* τ_k *is ready to execute but is not executing.*

Differently from the sequential case, each task τ_i may contribute with different p-jobs at the same time to the individual interference on a task τ_k. In the particular case when $i = k$, the critical interference $I_{k,k}(L)$ may include the interfering contributions of (non critical) p-jobs of task τ_k on itself, *i.e.,* the intra-task interference.

[3] While we prefer using the term parallel job instead of thread, we decided here to keep the name "thread" for homogeneity with the original definition. However, both terms are interchangeably used in this chapter.

[4] As it was mentioned in Chapter 2, a scheduling algorithm is said to be *work-conserving* if it never idles a core when there is a ready task waiting to be executed.

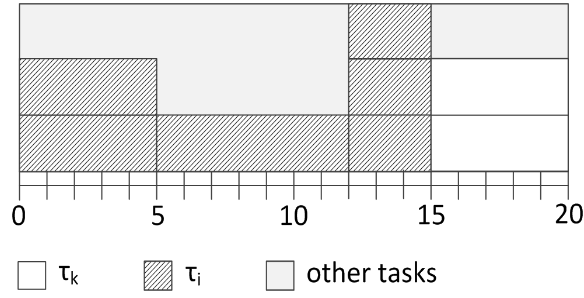


Figure 3.1: Task τ_i interfering on task τ_k

The next lemma allows expressing the total interference as a function of single task interferences.

Lemma 1. *For any work-conserving algorithm, the following relation holds:*

$$I_k(L) = \frac{1}{m} \sum_{\forall \tau_i} I_{i,k}(L). \tag{3.6}$$

Proof. From the work-conserving property of the considered scheduler, it follows that whenever a critical thread of τ_k is interfered, all m cores are busy executing other p-jobs. Therefore, the total amount of workload executed by p-jobs interfering with critical threads of τ_k within the considered window is $mI_k(L)$, giving the following relation:

$$\sum_{\forall \tau_i} I_{i,k}(L) = mI_k(L).$$

The lemma simply follows by rephrasing the terms. □

As previously mentioned, the individual interference $I_{i,k}(L)$ accounts for all p-jobs of τ_i interfering with τ_k, including p-jobs that are executing at the same time. In order to capture how many parallel jobs of τ_i may simultaneously interfere with task τ_k, we will borrow from [Chwa et al., 2013] the concept of *at least p-depth critical interference*[5].

Definition 4. *The* at least p-depth critical interference *of τ_i on τ_k in any interval of length L, denoted as $I^p_{i,k}(L)$, is defined as the total amount of time in which a critical thread of τ_k is ready to execute but cannot execute while there are at least p threads of task τ_i simultaneously executing in the system.*

To better understand the meaning of $I^p_{i,k}(L)$, consider the example in Figure 3.1, where task τ_i interferes with τ_k's execution with two threads for five time-units, one thread for seven time-units, and three threads for three time-units. In this case, $I^1_{i,k}(L) = 15$, $I^2_{i,k}(L) = 8$, and $I^3_{i,k}(L) = 3$.

[5]Note that we are simplifying the analysis and notations with respect to [Chwa et al., 2013], without making use of the "exact" p-depth interference, which, to our belief, is not needed for the purposes of this paper. Also the theorems presented in this section have therefore subtle differences from the corresponding ones in [Chwa et al., 2013]. This is for instance the case of Lemma 2, which differs from a similar result proved in [Chwa et al., 2013] in that the notion of "at least p-depth critical interference" is used instead of the "exact p-depth critical interference".

The following lemma allows establishing a relation between the overall critical interference on a task τ_k and the at least p-depth critical interference of each task τ_i on τ_k.

Lemma 2. *For any work-conserving algorithm, the following relation holds:*

$$I_k(L) = \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I_{i,k}^p(L). \tag{3.7}$$

Proof. Considering each single interfering task τ_i, the amount of execution by all p-jobs of τ_i interfering with τ_k within the considered window equals $\sum_{p=1}^{m} I_{i,k}^p(L)$. The Lemma follows from Lemma 1. □

We will now extend to the parallel task model considered in this paper two results proved in [Bertogna et al., 2005] and [Bertogna and Cirinei, 2007] for sequential tasks.

Lemma 3.

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^p(L), x\right) \geq mx \;\Leftrightarrow\; I_k(L) \geq x.$$

Proof. *If.* We would like to prove that if $I_k(L) \geq x$, then $\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^p(L), x\right) \geq mx$.

For a given length L, let ξ be the number of at least p-depth critical interferences $I_{i,k}^p(L) \geq x$, namely:

$$\xi = \left| \left\{ I_{i,k}^p(L) \geq x \right\}_{\forall i,p} \right|.$$

If $\xi > m$, then $\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^p(L), x\right) \geq \xi x > mx$. Otherwise, $(m - \xi) \geq 0$, and, using Lemma 2,

$$\begin{aligned}
\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^p(L), x\right) &= \xi x + \sum_{\forall \tau_i} \sum_{p: I_{i,k}^p < x} I_{i,k}^p(L) \\
&= \xi x + m I_k(L) - \sum_{\forall \tau_i} \sum_{p: I_{i,k}^p \geq x} I_{i,k}^p(L) && \text{[Lemma 2]} \\
&\geq \xi x + m I_k(a,b) - \xi I_k(a,b) \\
&= \xi x + (m - \xi) I_k(a,b) \\
&\geq \xi x + (m - \xi) x = mx. && \text{[using } I_k(L) \geq x\text{]}
\end{aligned}$$

Only if. From Lemma 2, we have

$$\begin{aligned}
I_k(L) &= \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I_{i,k}^p(L) \\
&\geq \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^p(L), x\right) \geq \frac{1}{m} mx = x.
\end{aligned}$$

□

Theorem 2. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by R_k^{ub} if*

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(I_{i,k}^{p}(R_k^{ub}), R_k^{ub} - P_k + 1\right) < m(R_k^{ub} - P_k + 1)$$

Proof. If the inequality holds, Lemma 3 gives

$$I_k(R_k^{ub}) < R_k^{ub} - P_k + 1.$$

Since a discrete time model is used, we have

$$I_k(R_k^{ub}) \leq R_k^{ub} - P_k.$$

The theorem then follows from Theorem 1. □

In the following section, the above theorem is used to derive a sufficient schedulability test for synchronous parallel task systems scheduled with a global fixed priority algorithm.

3.4 Response-Time Analysis

In order to exploit the theorem proved in the previous section to analyse the schedulability of parallel task systems, it is necessary to compute the critical interference terms. Since finding such terms is known to be a difficult problem for multiprocessor systems, a common approach is to use upper bounds that are easier to compute. An upper bound on the interference of a task τ_i in a window of length L is given by the maximum workload that τ_i can execute within the considered window. However, computing the maximum workload that can be executed by τ_i in a generic window is also a difficult task. To sidestep this problem, a typical technique is to consider pessimistic scenarios in which the workload in a given window cannot be smaller than in the worst-case situation. We hereafter describe the pessimistic scenario considered in this paper.

Consider a window of length L that spans the interval $[r_k, r_k + L]$ of a given (interfered) task τ_k. We call this interval of time the *problem window*. Within this window, we provide an upper bound on the execution of an interfering task τ_i. As commonly adopted in the literature, we will call *carry-in job* the first instance of τ_i executing in the problem window, having a release time before and deadline inside the window. By contrast, the *carry-out job* has its release time within (or before) and deadline after the window. Note that in this chapter, we consider that a job that has both release time and deadline outside the window is considered to be a carry-out job. All τ_i's instances whose release time and deadline are entirely contained within the considered window will be denoted as *body jobs*.

As shown in [Bertogna and Cirinei, 2007], the densest possible packing of sequential jobs of a task τ_i is found when:

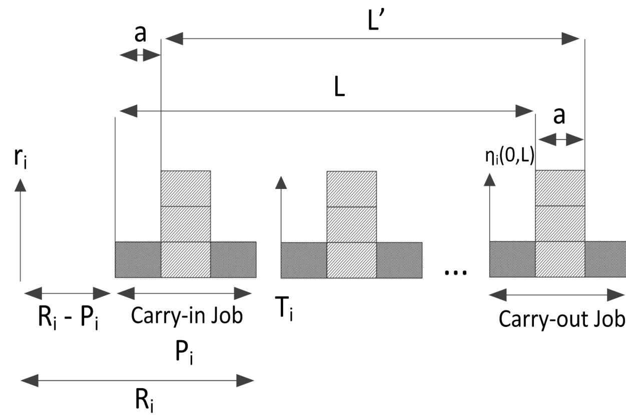


Figure 3.2: Densest possible packing of threads within the problem window

1. A job starts executing at the beginning of the problem window, and completes as close as possible to its response time. In other words, the job starts executing $R_i - P_i$ time-units after its release time, in correspondence to the beginning of the problem window.

2. All subsequent jobs of τ_i are executed as soon as possible after being released, *i.e.,* respecting the period T_i.

Such a situation is depicted in Figure 3.2 for a parallel task τ_i in the problem window.

3.5 Sliding Window Technique

An important observation to make is that the scenario described above may not represent the worst-case workload in the synchronous parallel task model considered in this chapter. This happens because the parallel task structure is characterized by precedence constraints that may affect the densest possible packing of p-jobs. Consider the example in Figure 3.2, where a task composed of three segments is considered in the above scenario. The carry-in job is fully contained inside the problem window L, while the carry-out is only partially contained. Now, if the window is shifted right by one segment (as represented by the window L' in the figure), the carry-in contribution decreases by one p-job, while the carry-out contribution increases by three p-jobs, leading to a larger task workload within the considered window.

In order to properly consider the worst-case workload contribution of each task in the problem window, we check all different meaningful alignments of the problem window with respect to the task structure. Note that shifting right the window of interest, the workload contribution has a discontinuity whenever one of the extreme points of the window coincides with a segment boundary. Therefore, we can check all possible scenarios in which the window of interest is shifted to the right from the original configuration, such that either (i) the window starts at the beginning of a segment of the carry-in job, or (ii) the window ends at the end of a segment of the carry-out job.

Formally, we consider the worst-case workload of a task τ_i in a window of length L, taking the maximum workload of the considered task, over all possible configurations in which the window

is shifted right from the original configuration by $a \in \Gamma_1 \cup \Gamma_2$, where Γ_1 and Γ_2 are the sets of significant offsets to check corresponding to scenario (i) and (ii), respectively (see Figure 3.5).

Before deriving the formal offset values to check, let $\eta_i(a,L)$ be the carry-out length for task τ_i in a window of length L and offset a. Then,

$$\eta_i(a,L) = \min\left(L, (L+R_i-P_i+a) \mod T_i\right).$$

We note that the meaningful offsets to consider in scenario (i) correspond to the best-case starting times of each segment $\sigma_{i,j}$ of τ_i, *i.e.,* $\sum_{x=1}^{j} P_{i,x}, \forall j \in [1,s_i]$. Moreover, all offsets greater than $P_i - \eta_i(0,L)$ can be ignored, since they would cause the end of the window to fall beyond the end of the carry-out job, resulting in a smaller workload. Therefore,

$$\Gamma_1 \doteq \left\{ \sum_{x=1}^{j} P_{i,x} \le P_i - \eta_i(0,L), \forall j \in [1,s_i] \right\}.$$

The offsets to consider in scenario (ii) correspond to the difference (when positive) between the best-case starting times of each segment $\sigma_{i,j}$ and the original carry-out length $\eta_i(0,L)$, *i.e.,*

$$\Gamma_2 \doteq \left\{ \max\left(0, \sum_{x=1}^{j} P_{i,x} - \eta_i(0,L)\right), \forall j \in [1,s_i] \right\}.$$

3.6 Decomposing the Carry-out Job

One last observation concerns *predictability*, as defined in [Ha and Liu, 1994][6]. A schedulability test needs to be predictable, in that it should consider all possible execution times of a task system, as long as they do not exceed the given worst-case execution time. In other words, we would like the response-time provided by our analysis to be sufficiently robust to consider all possible execution requirements of the given tasks, including when some segment $\sigma_{i,j}$ requires less than $C_{i,j}$ time-units, or when a task may skip some of the segments. A schedulability test that does not properly consider situations when execution requirements are reduced is by no means sufficiently robust for critical applications.

The problem with the above approach is that a larger workload may fit the considered window if the carry-out skips some segment. Consider the example in Figure 3.3. In the upper scenario, the original situation is depicted, with the carry-out job contributing to the workload in the window of interest with its first two segments. However, when the second segment of the carry-out job is skipped, a worse situation is found, as shown in the lower part of the figure, since a segment with a higher parallelism may enter the window, resulting in a larger workload.

Considering all possible combinations of execution times appears overly complicated as it requires a combinatorial exploration of the possible segment instances of each task. To solve this problem and therefore allowing our analysis to be sufficiently robust, we will consider a

[6] In [Baruah and Burns, 2006], a broader concept is defined, *i.e.,* "sustainability", which generalizes the notion of predictability.

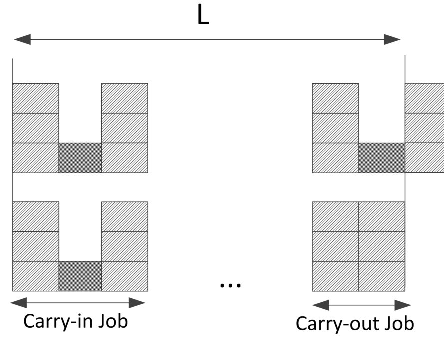


Figure 3.3: Densest possible packing of threads when a task skips some segment

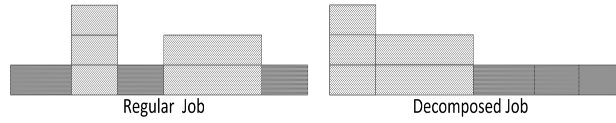


Figure 3.4: Example of a decomposed job

pessimistic situation in which the carry-out job is decomposed, re-aligning the parallel segments such that the segments with higher parallelism are shifted to the beginning of the job's execution. Thus, segments are ordered by their number of p-jobs following a non-increasing pattern where segments with a higher number of p-jobs execute first, as depicted in Figure 3.4.

Replacing the original carry-out job by a decomposed job results in placing the parallel segments with higher parallelism within the window of interest, which allows us to obtain a sound upper bound on the workload of the carry-out job.

We are now ready to derive an upper bound of the workload that each task may impose on a window of length L.

3.7 Workload of a Task Within a Window

Before presenting the analytical derivation of the workload components, we introduce the notion of "at least p-depth workload".

Definition 5. *The* at least p-depth workload *of a task τ_i in a window of length L, denoted as $W_i^p(L)$, is the sum of all intervals in which at least p threads of τ_i execute simultaneously in parallel.*

Note that the following relation holds by the definition of $I_{i,k}^p(L)$:

$$I_{i,k}^p(L) \leq W_i^p(L).$$

The above relation, together with Theorem 2, gives the following lemma.

Lemma 4. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by*

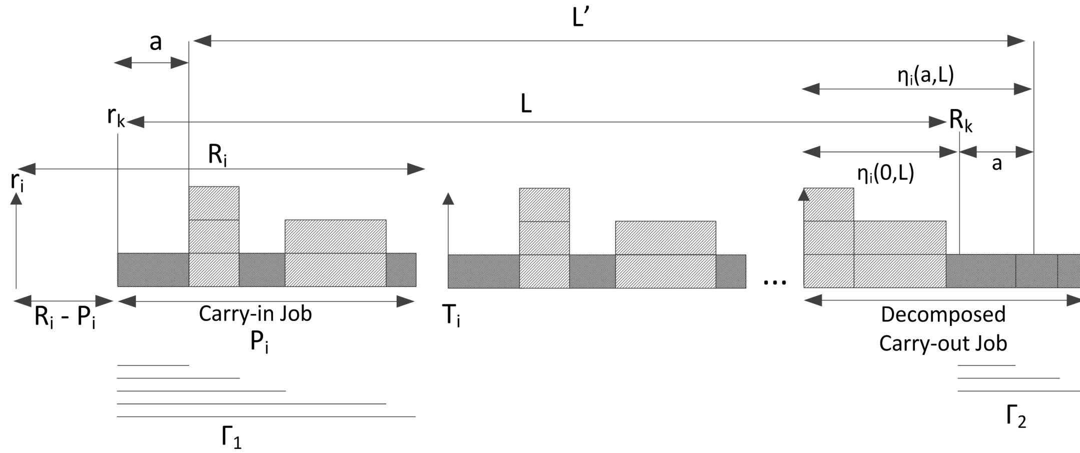


Figure 3.5: Response-time analysis details

R_k^{ub} *if*

$$\sum_{\forall \tau_i} \sum_{p=1}^{m} \min\left(W_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) < m(R_k^{ub} - P_k + 1)$$

It now only remains to derive an upper bound on $W_i^p(L)$. We will compute such an upper bound by considering the at least p-depth contributions of carry-in, body and decomposed carry-out of each task τ_i in the worst-case scenario summarized in Figure 3.5, for all significant offsets $a \in \Gamma_1 \cup \Gamma_2$.

To compute the at least p-depth workload of the decomposed carry-out job, it is necessary to consider the first $\eta_i(a,L)$ units of the decomposed carry-out job. The following function computes the at least p-depth workload executed within the first x units of a generic job of τ_i.

$$g_i^p(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ \sum_{j=1:m_{i,j} \geq p}^{z} P_{i,j} + (x - \sum_{j=1}^{z} P_{i,j}), & \text{if } 0 < x \leq P_i \\ & \text{and } m_{i,z+1} \geq p \\ \sum_{j=1:m_{i,j} \geq p}^{z} P_{i,j}, & \text{if } 0 < x \leq P_i \\ & \text{and } m_{i,z+1} < p \\ \sum_{\forall j:m_{i,j} \geq p} P_{i,j}, & \text{otherwise,} \end{cases} \tag{3.8}$$

where z represents the index of the last segment that is fully included in the interval, so that $(z+1)$ is the index of the segment that may execute partially within the carry-out interval.

The number of body jobs of τ_i executing in L is given by

$$\beta_i(L) = \left\lfloor \frac{L + R_i - P_i}{T_i} \right\rfloor - 1. \tag{3.9}$$

Note that $\beta_i(L)$ does not depend on a because the range in which a is varied never influences the

number of body jobs. The at least p-depth workload of the body jobs of τ_i executing in L is then given by

$$b_i^p(L) = \beta_i(L) \sum_{\forall j: m_{i,j} \geq p} P_{i,j}. \tag{3.10}$$

The carry-in length $\alpha_i(a,L)$ can be derived as[7]

$$\alpha_i(a,L) = L - \eta_i(a,L) - \beta_i(L)T_i.$$

The at least p-depth carry-in contribution can then be derived by computing the workload executed within the last $\alpha_i(a,L)$ units of the carry-in job. The following function (from [Chwa et al., 2013]) computes the at least p-depth workload executed within the last x units of a job of τ_i.

$$f_i^p(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ \sum_{j=h:m_{i,j} \geq p}^{s_i} P_{i,j} + (x - \sum_{j=h}^{s_i} P_{i,j}), & \text{if } 0 < x \leq P_i \\ & \text{and } m_{i,h-1} \geq p \\ \sum_{j=h:m_{i,j} \geq p}^{s_i} P_{i,j}, & \text{if } 0 < x \leq P_i \\ & \text{and } m_{i,h-1} < p \\ \sum_{\forall j:m_{i,j} \geq p} P_{i,j}, & \text{otherwise,} \end{cases} \tag{3.11}$$

where h represents the index of the earliest segment that is fully included in the interval, so that $(h-1)$ is the index of the segment that may execute partially within the carry-in interval.

Considering Equation 3.8, Equation 3.10 and Equation 3.11, an upper bound on the at least p-workload of a task τ_i in a window of length L and offset a is given by:

$$\widehat{W}_i^p(L,a) = f_i^p(\alpha_i(a,L)) + b_i^p(L) + \tilde{g}_i^p(\eta_i(a,L)), \tag{3.12}$$

where $\tilde{g}$ denotes that the function g is applied to the decomposed job. An upper bound on the worst-case workload of τ_i with depth at least p in a window of length L is then derived as

$$\widehat{W}_i^p(L) = \max_{a \in \Gamma_1 \cup \tilde{\Gamma}_2} \left\{ \widehat{W}_i^p(L,a) \right\}, \tag{3.13}$$

where $\tilde{\Gamma}_2$ denotes that the offsets in this set are computed, again, considering the decomposed job.

Note that the above expression can be used to bound the inter-task workload from interfering tasks.

Before applying Lemma 4, a bound should also be provided to the intra-task interference, accounting for the workload of p-jobs from the same task. An upper bound on the intra-task

[7] When $R_i = P_i$ and $L \geq T_i$, the first job of τ_i executing in the window of interest is accounted for in the carry-in and not in the body contribution despite it has both release time and deadline within the window.

workload of task τ_k with depth at least p can be given by:

$$\widehat{W}_k^p = \sum_{\forall j: m_{k,j} \geq p+1} P_{k,j}, \tag{3.14}$$

where the sum is extended over all segments with parallelism at least $p+1$ instead of p since the p-jobs of the critical threads do not contribute to the critical interference.

3.8 Schedulability Condition

Given the worst-case inter-task and intra-task workloads presented in the previous sections, we are now in a position for deriving an upper bound on the worst-case response time of a parallel task.

Lemma 5. *Given a set of synchronous parallel tasks τ scheduled by any work-conserving algorithm on m identical cores, the worst-case response-time of each task τ_k can be upper bounded by R_k^{ub} if*

$$\begin{aligned}\sum_{\forall \tau_{i \neq k}} \sum_{p=1}^{m_i} \min\left(\widehat{W}_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) \\ + \sum_{p=1}^{m_k} \min\left(\widehat{W}_k^p, R_k^{ub} - P_k + 1\right) \\ < m(R_k^{ub} - P_k + 1).\end{aligned}$$

Proof. The proof simply follows from Lemma 4, using the derived upper bounds instead of the real p-depth workload, and extending the p-indexed sum over the maximum number of p-jobs of each task[8]. □

For the special case of global fixed-priority scheduling, the interfering workload may be limited to the set of tasks having higher priority than τ_k. The following theorem can then be used to derive R_k^{ub} in a fixed priority setting.

Theorem 3. *Given a set of synchronous parallel tasks τ scheduled by global fixed-priority on m identical cores, an upper bound R_k^{ub} on the worst-case response-time of a task τ_k can be derived by the fixed-point iteration of the following expression, starting with $R_k^{ub} = P_k$:*

$$R_k^{ub} \leftarrow P_k + \left\lfloor \frac{1}{m}\left(\sum_{\forall i<k} \sum_{p=1}^{m_i} \min\left(\widehat{W}_i^p(R_k^{ub}), R_k^{ub} - P_k + 1\right) + \sum_{p=1}^{m_k} \min \left(\widehat{W}_k^p, R_k^{ub} - P_k + 1\right)\right)\right\rfloor.$$

Proof. If the iteration ends before R_k^{ub} reaches D_k, it is easy to see that the condition of Lemma 5 is satisfied, proving the theorem. □

[8]As in [Chwa et al., 2013], we are not taking advantage of the fact that carry-in and carry-out contributions may be less dense than in the considered scenario when there is some segment $\sigma_{i,j}$ with a parallelism $m_{i,j}$ greater than the number of processors m.

A schedulability test for systems scheduled with global fixed-priority is easily derived by computing R_k^{ub} for each task τ_k *in priority order*, starting from the highest priority one, and checking whether $R_k^{ub} \leq D_k$ for all tasks. If not, the test is not able to guarantee the schedulability of the system. Note that, updating response time upper bounds in priority order allows one to optimally exploit Theorem 3, since every task can use the most updated response times of the higher priority tasks, leading to smaller inter-task interferences.

3.9 Complexity

The complexity of the proposed response-time analysis is pseudo-polynomial in the task parameters, as is the original response-time analysis for sequential task sets presented in [Bertogna and Cirinei, 2007]. However, with respect to the sequential analysis, an additional s_i term has to be considered to account for the sliding window technique that repeats the workload computation for all segment starting times of the carry-in and carry-out jobs.

To obtain a faster analysis, a simple method is to consider the complete execution of the carry-in and carry-out job instances. To do that, it is sufficient to replace $\widehat{W}_i^p(L)$ in Theorem 3 with the following term:

$$\widehat{W}_i^p(L) = \left(\left\lfloor \frac{L + R_i - P_i}{T_i} \right\rfloor + 1 \right) \sum_{\forall j : m_{i,j} \geq p} P_{i,j}. \tag{3.15}$$

As we will show in the experimental section, this method allows obtaining a faster worst-case response time computation without significant schedulability losses.

3.10 Evaluation

This section presents the simulation results to evaluate the behaviour of our schedulability analysis, comparing it to the approach proposed by [Chwa et al., 2013]. We only show the results for the implicit deadline case, which are however representative of the general behaviour. Concerning the simulation environment, we use a similar setting as in [Chwa et al., 2013]. We start by generating a task set with m tasks, creating new task sets by adding a new task to the previous one until the task set utilization exceeds the number of processors. The above procedure is repeated until 40,000 task sets are generated.

The percentage of parallel tasks in the task set is controlled by a parameter that generates a random percentage value in the interval $[0, 100]$. The periods of sequential tasks are uniformly generated in $[100, 1000]$, with C_i uniformly chosen from $[1, T_i]$. For parallel tasks, the number of segments s_i is uniformly generated in $[1, 5]$; the number of threads per segment $m_{i,j}$ is uniformly generated in the interval $[1, 3m/2]$; the worst-case execution times of the threads in each of the segments is uniformly chosen in the interval $[1, T_i/s_i]$; periods are uniformly generated in $[100, 10000]$.

For the generated task sets, we compare the number of schedulable task sets detected by our analysis (PAR-RTA) with the approach proposed in [Chwa et al., 2013], denoted as PAR-EDF.

As the authors in [Chwa et al., 2013] show that PAR-EDF outperforms approaches that use decomposition techniques to schedule parallel tasks, we do not perform this comparison ourselves. In our results, we also show the performance of the faster method (PAR-RTA-UP) presented in Section 3.9 that uses the workload upper bound given by Equation (3.15).

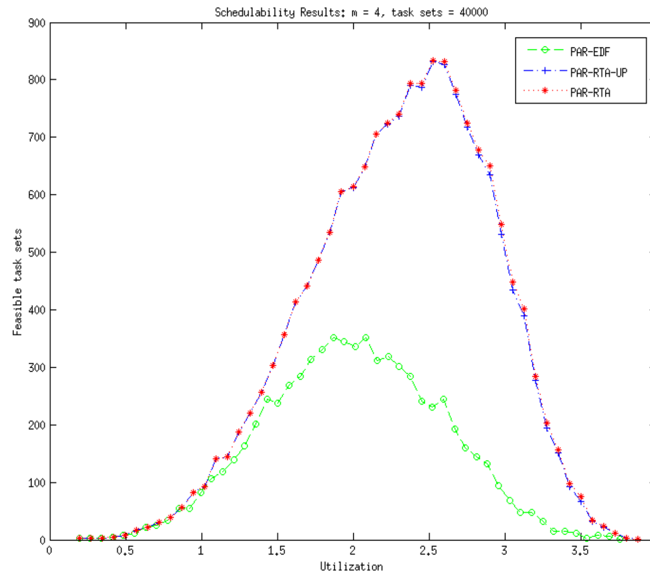


Figure 3.6: Number of schedulable task sets detected by the considered tests for m = 4

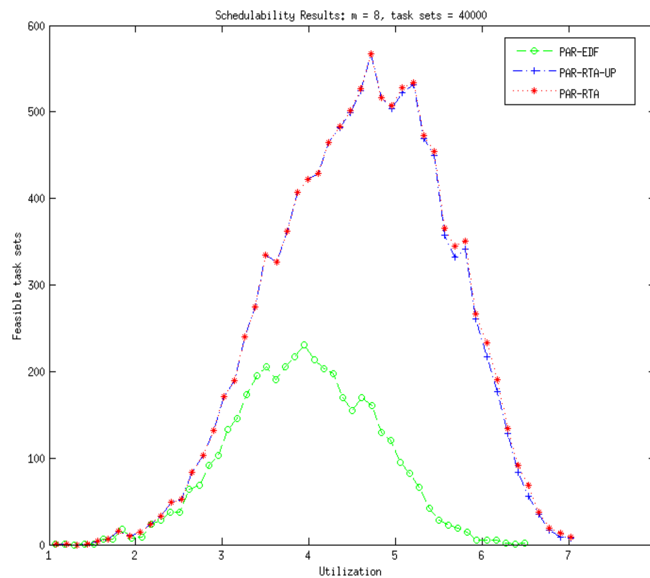


Figure 3.7: Number of schedulable task sets detected by the considered tests for m = 8

Figure 3.6 shows the results for $m = 4$. Both our approaches clearly outperform PAR-EDF, detecting 230% more schedulable task sets. Interestingly, the faster method using the simplified upper bound has a performance very similar to the complete method (within 1%)[9]. Increasing the number of processors, the situation is similar. Figure 3.7 shows the case with $m = 8$. While the

[9]We found a similar result for sequential task sets, comparing the test in [Bertogna and Cirinei, 2007] with a pessimistic version that accounts for a complete carry-out contribution.

number of schedulable task sets detected by all tests decreases, the relative performances remain the same.

3.11 Summary

In this chapter the problem of scheduling parallel real-time tasks in a multiprocessor system composed of m homogeneous cores was addressed. Considering the synchronous task model, we derived upper-bounds on the workload within a window of interest in order to compute response-time upper bounds of the interfering jobs. Two techniques are used for the derivation of the contribution of the worst-case workload in a window of interest. The first technique introduced is the sliding window technique which allows one to check all the different meaningful alignments that may occur in the problem window with respect to the task structure. The second technique considers the decomposition of the carry-out job in order to make the proposed approach sustainable. That is, the decomposition of the carry-out job requires that parallel segments of the carry-out job are re-aligned such that the segments with higher parallelism are shifted to the beginning of the job's execution. Consequently, more workload is moved inside the window of interest. Both techniques make our analysis sufficiently robust to be considered in more critical scenarios.

Regarding the obtained results, the presented approach improves over the state of the art reported in [Chwa et al., 2013], by providing tighter schedulability conditions and extending the analysis to fixed-priority task systems. In addition, we clearly outperform the work in [Chwa et al., 2013], in the number of schedulable task sets.

Future work includes the application of the improvement proposed by Nan Guan et al. in [Guan et al., 2009] for synchronous parallel real-time tasks. Precisely, we can simplify the assumption that every higher priority task has carry-in and make our analysis less pessimistic by considering that only $(m-1)$ tasks contribute for the carry-in term as presented in [Guan et al., 2009].

In the subsequent chapter, we continue to explore the parallelism provided by current multicore systems. Instead of considering a purely global scheduling approach, we introduce a novel approach that combines a semi-partitioned scheduling with a variant of work-stealing.

Chapter 4

Applying Work-stealing to Real-time Systems

4.1 Introduction

Work-stealing is a load-balancing algorithm that allows an idle core to randomly steal workload from a busy core (usually referred to as the *victim*) with the objective of reducing the average response time of parallel tasks. Several properties make it a viable algorithm to be used in multiprocessor scheduling. Namely, it is capable of load balancing workloads, provide good data locality and, due to its random stealing behaviour, contention can also be reduced.

While randomness in the selection of a victim is traditionally acceptable in several computing domains, no guarantee can actually be provided regarding the timing behaviour of tasks due to the possibility of priority inversion. Hence, if one wants to use work-stealing in real-time systems, it has to modify the original algorithm to circumvent this issue.

This chapter starts by detailing the behaviour of work-stealing and presenting its limitations with respect to real-time systems (Section 4.2 and Section 4.3). Then, Section 4.4 presents works found in the literature that apply work-stealing in real-time systems and Section 4.5 introduces a new data structure that may be used in the context of these systems. In Section 4.6 semi-partitioned scheduling is introduced and in Section 4.7 the system model used throughout this chapter is detailed. An approach that combines a variant of work-stealing with semi-partitioned scheduling is presented, in Section 4.8. The approach consists of an offline stage and an online stage. During the offline stage, a multi-frame task model is adopted to perform the fork-join task-to-core mapping so as to improve the schedulability and the performance of the system. During the online stage, the variant of work-stealing is used among cores to improve the system responsiveness as well as to balance the execution workload. The end goal of this approach is to reduce the average response time of tasks and create additional room in the schedule for less-critical tasks (*e.g.,* aperiodic and best-effort tasks).[1].

[1]Note that the balance of the platform workload at runtime also allows for a better control of the platform energy consumption [Aydin and Yang, 2003; Kang and Waddington, 2012].

Finally, the schedulability analysis for the approach is presented in Section 4.10 and different experiments considering different allocation heuristics are also performed. Results are presented in Section 4.11.

4.2 Randomised Work-stealing

Blumofe et al. ([Blumofe and Leiserson, 1999]) proposed a randomised work-stealing scheduler with provable time and space bounds for parallel applications with fully-strict computations (in a fully-strict computation a task only synchronises with its parent). The randomised work-stealing scheduler consists of a pool of worker threads (usually there is a one-to-one mapping of worker threads to cores) where each worker thread maintains a local double-ended queue. A double-ended queue, depicted in Figure 4.1, is a concurrent data structure that operates as a queue and stack by allowing push and pop operations at both ends [Knuth, 1997] (in short *deque*).

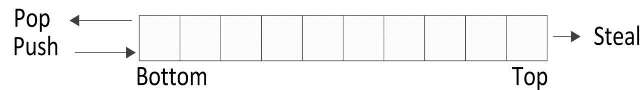


Figure 4.1: Work-stealing deque data structure

When a task spawns a new sub-task, the parent task is suspended and pushed into the deque and the just spawned child task starts executing. When the child completes its execution, the parent task is popped from the deque and resumes its execution. Worker threads access their local deques as a stack by pushing and popping tasks from the bottom of the deque, in a Last-in, First-Out (LIFO) order. However, whenever a worker thread becomes idle, as a result of the local deque becoming empty, it turns into a *thief* and it may steal work from other randomly selected busy worker threads, known as the *victims*. When stealing work from a victim, thieves treat the victim's deque as a queue and steal work that was enqueued first (the topmost task), by following a First-In, First-Out (FIFO) order. If the chosen victim has a task in its deque, this task is stolen and it is executed by the thief. Otherwise, if no task is found in the deque of the selected victim, a new random victim is selected. The selection process continues until either a new task is found in a victim's deque or no task is found and the thief suspends its execution.

The main benefits of randomised work-stealing are threefold: (i) the reduction of contention; (ii) the load balancing of the workloads; and (iii) providing good data locality [Blumofe and Leiserson, 1999].

Contention is reduced by design as instead of a single concurrent work queue shared among worker threads, worker threads own their local copy of a deque. Moreover, the way worker threads operate on the deques also contributes to a reduction in contention. Indeed, worker threads execute their own work from one end of the deque and steal work from the other end of a victim's deque.

Two aspects contribute to a good load balancing strategy. The first aspect relies on the fact that early executed tasks may generate more work than later generated tasks. That is, due to the way that tasks are pushed into the deque, a parent is pushed to the deque while the child is

executing. Consequently, parents become candidates to be stolen thereby increasing the chance of further parallel decompositions. The second aspect relates to the fact that idle worker threads have the initiative and look for work in other worker threads' deques while busy threads execute their work. Not only this contributes to load balancing, but also to a reduction of overhead, as it is the responsibility of the idle worker to perform all the migration operations (instead of a busy worker).

Finally, randomised work-stealing offers good data locality as long as tasks that are close together in the computation graph (as for instance, nearest neighbour tasks) are scheduled in the same processor. Nevertheless, further work has been done in order to improve the data locality of randomised work stealing for parallel workloads as for instance [Acar et al., 2000] and [Narlikar, 2002].

4.3 Limitations of Randomized Work-stealing with Respect to Real-Time Systems

While randomness in the selection of a victim is traditionally acceptable in several computing domains, no guarantees can actually be provided regarding the timing behaviour of the tasks. Thus, the general purpose randomised version of work stealing does not present a deterministic and predictable behaviour that allows it to be used in real-time systems as is. There are two main reasons that need to be considered. The first reason is that in randomised work-stealing the tasks' response times are unbounded, *i.e.,* as new tasks are spawned these are pushed into the bottom of a worker's deque making a task at the top to wait unboundedly if all workers are busy. The second reason is that using one deque per core is a source of priority inversion if different task priorities are considered. In the randomised approach, worker threads steal tasks from randomly selected deques of other threads. If different task priorities are considered (as it is typically the case in real-time systems), high priority tasks may eventually be pushed to the thread's deque after lower priority tasks. This behaviour may lead to priority inversion and consequently deadline misses as in the end of the deque there may be lower priority tasks to be stolen by thief threads. A motivational example to this problem is presented next.

Example 4.3.1. *Let us assume a system with two cores and two worker threads, WT_1 and WT_2. In core 1, WT_1 is executing a low priority task (τ_l), and in core 2, WT_2 is executing a high priority task (τ_h^1). Now let us further assume that τ_l spawns low priority subtasks which are pushed into core 1's deque. If at this particular time instant a new high priority task becomes ready to execute (let us denote it τ_h^2), τ_l is preempted. If during its execution τ_h^2 also spawns new subtasks, these subtasks are enqueued into core 1's deque, pushing older subtasks (the ones with low priority) to the "end" of the queue, according to the rules of randomised work-stealing. If at this time instant, core 2 becomes idle, its worker thread is allowed to steal work from core 1's deque. Since randomised work-stealing works by stealing older subtasks from the deque, it will steal and execute those of low priority (the subtasks of τ_l).*

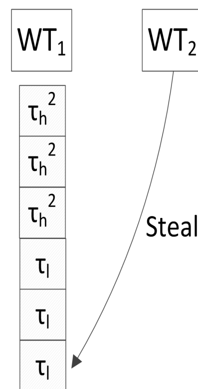


Figure 4.2: Priority inversion scenario. State of the system when WT_2 steals work from WT_1.

Clearly, Example 4.3.1 describes a priority inversion scenario (the state of the system when the priority inversion occurs is depicted in Figure 4.2). In order to have a correct behaviour, from a real-time systems perspective, both cores should have been executing the subtasks spawned by τ_h^2 in parallel[2]. However, if stealing was not allowed, core 2 would have been idle and the system would have been wasting resources.

The example above shows that if one wants to have deterministic and predictable work-stealing, the randomised work-stealing algorithm needs to be modified. In particular, all the introduced modifications should be handled carefully not only to assure the timeliness of real-time tasks, but also to take into consideration the impact of task migration and its effects on the predictability of the system.

4.4 Literature on Real-Time Work-Stealing

In the literature, there are only a few papers that apply work-stealing in real-time settings or use it for the scheduling of tasks with priorities. In this section we cover the papers that we are aware of.

Nogueira and Pinho [Nogueira and Pinho, 2012] propose a server-based approach combined with work-stealing to support parallel tasks. The same authors, in [Nogueira et al., 2012], propose an approach that combines global EDF with work-stealing, albeit this approach only covers simple fork-join tasks.

Mattheis et al. [Mattheis et al., 2012] devote their attention to the application of work-stealing for stream processing applications running in soft real-time systems. In particular, a few variants of randomized work-stealing are proposed considering different queuing orders, enqueueing policies, and stealing policies, with the objective of achieving fairness in task execution and minimizing latency. For one of the variants, *i.e.,* global enqueueing policy, the authors provide a bound on the latency. Their results show that for streaming applications the proposed bound is safe for strategies that employ a global queue in addition to local queues. Moreover, they also show that these strategies perform better than randomized work-stealing, without reducing the throughput

[2]The scenario will be the same even if task τ_h^2 is executed in core 1, and task τ_l in core 2 (which is common in regular global scheduling systems).

of applications while maintaining a minimal overhead. Even though the target of this work is soft real-time systems, no schedulability analysis is given for the proposed strategies nor tardiness bounds for the execution of the tasks. In addition, we suspect that for tasks with irregular parallelism the proposed bound does not hold as different worker threads may have to execute different amounts of work.

Imam and Sarkar [Imam and Sarkar, 2015], in parallel to the work developed in this dissertation, present a decentralized work-stealing scheduler that considers fixed-priority tasks in a non-preemptive manner. The proposed algorithm takes into account global priorities and steals are performed even if the worker thread is not idle in order to avoid priority inversions. Similar to what is presented in this dissertation, in Section 4.5, tasks are classified according to their priority and consequently, are stored in the respective deque. The authors evaluate how different pool implementations, using different types of queues/deques, perform in different benchmarks. Their results show that centralized queues present more overhead than decentralized ones and that decentralized pools handle the scheduling of global priorities as well as global queues but with low overhead. Moreover, the authors did not observe any increase in overheads due to the increase in the frequency of steals, occurring due to the handling of global priorities.

Li et al. [Li et al., 2016] compare randomized work-stealing against a deterministic centralized greedy scheduler for scheduling soft real-time parallel tasks. The authors divide their study into two parts. In the first part they show that in many scenarios, work-stealing has smaller response-times with low variation and a better speedup than the centralized greedy scheduler. This result is easily understandable due to the higher overheads that the centralized greedy scheduler poses in the manipulation of its centralized queue of ready nodes. The low variation in the response times achieved by the work-stealing scheduler leads to the second part of their work where they study the performance of work-stealing for the scheduling of soft-real time tasks. In the second part of their work, the authors integrate work-stealing into federated scheduling[3].

The motivation for using federated scheduling is to decide the core assignment for the tasks offline and then, following that assignment, use work-stealing online to schedule those tasks exclusively in the dedicated cores. Their results show that work-stealing has a lower deadline miss ratio (*i.e.,* missed deadlines over number of jobs in an interval of time), lower relative response-times and a smaller number of required cores when compared to the centralized greedy scheduler. The conclusion of their study is that work-stealing can improve the response-time of the tasks and therefore is a good candidate for soft real-time systems.

[3]Federated scheduling is a scheduling technique for parallel real-time tasks. Federated scheduling either admits a task set providing as a result a core assignment for each task or declares the task set unschedulable. To provide the core assignment, each task with a utilization greater than 1 (denoted as a high utilization task) is allocated η_i cores according to the following expression $\eta_i = \lceil \frac{C_i - P_i}{D_i - P_i} \rceil$. During runtime, each high utilization task is ensured to execute exclusively in η_i cores. All the remaining tasks, are partitioned in the remaining cores.

4.5 A New Data Structure

As observed in Section 4.3, the randomised behaviour of work-stealing must be modified if one wants to use it for scheduling parallel real-time tasks. However, as we will see next, the deque is no longer a viable data structure and the behaviour of task selection must be slightly modified as well.

Let us focus on a simple approach that considers task priorities and follows the same philosophy of having a single deque per worker thread. Such an approach requires a global data structure that holds information about which worker thread contains the highest priority task. Instead of randomly selecting a victim, and therefore avoiding priority inversion, whenever a worker thread is idle it must first consult the global data structure. Based on the information contained in the structure, it accesses the respective deque to look for the workload to be stolen. The thief iterates through the tasks in the respective deque until it finds one with a priority equal to the highest priority task. The thief steals and executes it. The problem with this approach is that it greatly increases the theft time, and therefore cannot be considered a valid solution.

As alternative, one may think of priority queues, often used in single core schedulers, as a viable solution when moving to a parallel context. Nevertheless, concurrent priority queues are hard to make both scalable and fast [Lenharth et al., 2011]. Furthermore, the semantics of priority queues naturally suggest an ordered insertion method, which is against the work-stealing deque philosophy.

A viable solution that circumvents the limitations of the above approaches consists in having each worker thread store ready tasks in a priority ordered list, where each element is a deque that stores tasks of a given priority, in a similar fashion to Multi-Level Queue Scheduling [Silberschatz et al., 2008]. This list supports the same operations as the ones supported by randomized work-stealing operating in a single deque, that is, *push* and *pop* performed by the owner worker thread on the bottom of the deque to insert and remove a task, respectively; and *steal* which is invoked by a thief in order to steal a task from the top of the highest priority deque. In addition, there is the need of using a global data structure that keeps information about which worker thread holds the highest priority task at any given instant of time. Such a structure is accessed whenever a worker thread becomes idle.

With both of the above data structures, work stealing can be made deterministic (*i.e.,* without random steals) and *more* predictable as now thieves know which victim(s) hold the highest priority task(s). However, this does not suffice to achieve a fully predictable execution as two different worker threads may be executing different tasks with different priorities and spawn new sub-tasks into their deques at any given time instant. Again, a case of priority inversion, but this time due to the scheduling rule that a steal operation may only occur whenever a worker thread becomes idle.

To achieve a fully predictable execution one must avoid priority inversion. Thus, a worker should always check if there is a higher priority task in the system before executing any local tasks. This entails that a worker should steal even when there may be lower priority work in one of its deques.

In our opinion, there is a trade-off that should be decided by the system designer. That is, either steal immediately when there is higher priority work in the system or wait until a worker becomes completely idle before stealing higher priority work at the cost of allowing a few priority inversions.

We believe that work-stealing is amenable of being used in real-time systems but in a controlled setting (at least while there is no deterministic version of the algorithm with the respective schedulability analysis). Thus, we selected a setting where work-stealing is combined with semi-partitioned scheduling with task-level migration and a multiframe task model in order to reduce the average response-time of tasks. Consequently, additional room can be created in the schedule for less-critical tasks (*e.g.,* aperiodic and best-effort tasks).

4.6 Semi-partitioned Scheduling

Semi-partitioned scheduling ([Anderson et al., 2005], [Andersson and Tovar, 2006], [Kato et al., 2009]) combines properties from both partitioned and global scheduling approaches as a way to increase the processor utilization bounds observed in partitioned scheduling (*i.e.,* 50%).

As stated in Chapter 2, in semi-partitioned scheduling, a subset of tasks is statically assigned into the cores as in partitioned scheduling, with no possible migrations for these tasks at runtime; and the remaining tasks in the set are scheduled by using a global scheduling algorithm in order to improve the processor utilization. The globally scheduled tasks are allowed to migrate between the cores.

Considering the time instant at which a migration occurs, semi-partitioned scheduling can be further classified into two subcategories: (1) Task-level migration [Dorin et al., 2010], where several jobs of a migrating task are allowed to be assigned to different cores, but once a job is assigned to a core, migration of this job prior to its completion is forbidden; and (2) Job-level migration [Kato et al., 2009], where several jobs of a migrating task are allowed to be assigned to different cores, and migration of each job prior to its completion is also allowed.

Unfortunately, to the best of our knowledge, very few techniques exist in the literature for the analysis of semi-partitioned scheduling of parallel tasks. Bado et al. [Bado et al., 2012] proposed a semi-partitioned approach with job-level migration for fork-join tasks, which is similar to the one in [Lakshmanan et al., 2010], but due to the assignment methods proposed in their paper for the offsets and local deadlines, they did not provide any guarantee on the fact that parallel jobs (in short p-jobs) actually execute in parallel. While their work is similar to ours with respect to the adopted class of schedulers (semi-partitioned), we differ in that we relax the constraint of restricting the task parallelism and we use task-level migration instead of job-level migration, thus further reducing the number of migrations at runtime.

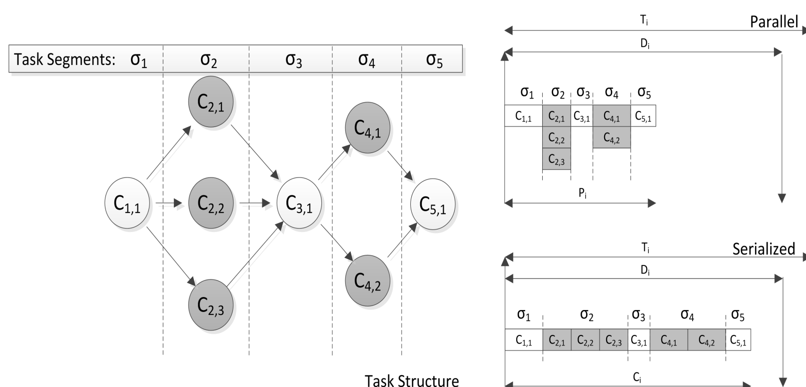


Figure 4.3: Fork-join task. In the figure, the task structure is represented together with its timing properties (upper right side) and its serialized representation (lower right side).

4.7 System Model

In this chapter, we use the fork-join task model. As described in Chapter 2, the fork-join model is more restrictive than the synchronous model. Specifically, in the fork-join model only the even segments of a parallel task can have an arbitrary number of p-jobs. The odd segments have a single p-job that, with the exception of the first p-job, synchronises the computation occurring in the even segments.

Even though we follow a more restrictive task model, most of the notation presented in the previous chapter is still valid and will be used throughout this chapter. New notation is introduced when it is required to do so.

The left side of Figure 4.3 illustrates a fork-join task τ_i with $n_i = 5$ segments, three are sequential segments (σ_1, σ_3 and σ_5) with one p-job each and two are parallel segments: σ_2 containing three p-jobs and σ_4 containing two p-jobs. All the p-jobs in the parallel segments are independent from each other (other than the processing units and segment precedence constraints there are no other shared resources among the p-jobs) and therefore can execute in parallel. On the upper right side of the figure it is possible to observe the task structure framed according to the timing properties of the task (P, D, T) and on the bottom right side it is possible to observe the task's serialized representation (*i.e.,* task execution without parallelism).

Every p-job is assumed to execute on *at most* one core at any time instant and can be preempted prior to its completion by another p-job with a higher priority. A preempted p-job resumes its execution on the same core where it was executing prior to preemption. Moreover, we assume that each preemption is performed at no cost or penalty.

4.7.1 Earliest Deadline First

In this chapter, we assume that each of the m cores runs a fully preemptive Earliest Deadline First (EDF)[4] scheduler (also known as partitioned EDF). The advantage of using EDF is that it

[4] As the reader may recall (from Chapter 2), EDF scheduling policy dictates that the smaller the absolute deadline of a job, the higher its priority.

was proven in [Liu and Layland, 1973] that task sets with a total utilization no greater than 1 are schedulable with EDF. In addition, as we are dealing with sporadic tasks with constrained deadlines (*i.e.,* $D_i \leq T_i$), the schedulability analysis of EDF for this particular task model can be performed using the processor demand approach [Baruah et al., 1990].

The processor demand bound function approach works as follows. For each task τ_i executing in an interval $[t_1, t_2]$, the processor demand is the amount of computation time $c(t_1, t_2)$ requested by τ_i's instances that have release time and deadline within the interval $[t_1, t_2]$. Then, a task set is feasible if and only if for all tasks the processor demand in any interval of time does not exceed the available time $(t_2 - t_1)$, *i.e.,* $\forall t_1, t_2, c(t_1, t_2) \leq t_2 - t_1$. Thus, one needs to compute the number of instances for each task τ_i in the interval $[t_1, t_2]$ in order to compute the processor demand in such interval. For a task τ_i the number of instances can be expressed as follows:

$$\eta_i(t_1, t_2) = \max\left\{0, \left\lfloor \frac{t_2 + T_i - D_i - \mathrm{o}_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - \mathrm{o}_i}{T_i} \right\rceil \right\}$$

, where o_i is the release time of τ_i's first instance.

The processor demand in $[t_1, t_2]$ is given by:

$$c(t_1, t_2) = \sum_i^n \eta_i(t_1, t_2) \cdot C_i$$

When all tasks are released at $t = 0$, η_i is simplified and the processor demand $c(0, t)$, for interval $[0, t]$, is known as the demand bound function, *i.e.,*

$$\mathrm{dbf}(t) = \sum_i^n \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i$$

.

Hence, a synchronous task set composed of sporadic tasks with constrained deadlines is schedulable by EDF if and only if $\mathrm{dbf}(t) \leq t, \forall t \geq 0$.

4.7.2 Multiframe Task Model

The multiframe task model was introduced by Mok and Chen [Mok and Chen, 1997] to generalize the Liu and Layland model [Liu and Layland, 1973]. The idea behind this model is to allow system designers to model tasks in which different instances (or frames as they are named in this model) may have different execution times from one another but follow a known execution pattern. Instead of using a single value for the WCET, each task is modelled by defining a static and finite list of execution times, corresponding to successive jobs or frames. Thus, a periodic sequence that may (possibly) repeat infinitely is obtained where the execution time of each frame is bounded above by the corresponding value in the sequence.

The model introduced by Mok and Chen was generalized by Baruah et al. [Baruah et al., 1999]. In the generalized multiframe model, known as GMF, task parameters such as deadline and period

are allowed to differ from one another.[5]

Formally, the multiframe task model assumed in this chapter is represented by a tuple (E,D,T), where E is an array of $k \geq 1$ execution times $(C^0,C^1,\ldots,C^{k-1})$, *i.e.,* the number of frames of the task, and T is the period that separates each frame. The execution time of the i-th frame is given by $C^{(i-1) \bmod k}$, where $i \geq 1$. The deadline of each frame is D time units after its release, with $D \leq T$.

In this chapter, the multiframe task model is used in the second phase of the proposed approach. As the reader will see in the next section, the second phase of the proposed approach requires one to find an execution pattern such that a task that is deemed unschedulable when fully assigned into a single core, may be deemed schedulable if an execution pattern is found in which different frames of the task execute in different cores.

4.8 Semi-partitioned Scheduling and Work-Stealing

Semi-partitioned scheduling was chosen in the context of our work because it allows one to: (1) apply all the body of knowledge that exists for single core scheduling. This is needed whenever some task is assigned into a core in a partitioned manner; and (2) take advantage of global scheduling in order to schedule parallel tasks simultaneously in different cores and consequently, use work-stealing as a load balancing mechanism.

The proposed approach consists of three phases, referred to as *task assignment*, *offline scheduling*, and *online scheduling*. The intuitive idea behind each phase is summarized below:

1. *Task assignment.* In this phase tasks are categorized according to their density. Then, a task-to-core assignment heuristic is applied to determine the set of non-migrating and the set of migrating tasks. The proposed heuristic considers the demand of each core after each "new" task is assigned by using the demand bound function (see Section 4.7.1). In this process, sequential tasks are evaluated first so that the capacity of the cores is filled as much as possible and thus, let the work stealing mechanism be exploited by parallel tasks in order to potentially decrease their response times.

2. *Offline scheduling.* In this phase the execution pattern of each migrating task is determined so as to meet all the timing requirements of the system. By recurring to the multiframe task model (see Section 4.7.2), the heuristic tries to find an execution sequence so that jobs (or frames) of each migrating task are mapped to the cores. Then, on each core the schedulability can be verified by using uniprocessor schedulability techniques.

3. *Online scheduling.* In this phase the structure of each parallel task is considered and work-stealing is applied among cores that share a copy of a migrating task. Specifically, an idle core with a copy of a migrating task can contribute to the execution of this task by stealing workload from another core and executing the stolen workload. Before stealing any workload, an admission control test is performed on the stealing core in order not to jeopardize

[5] The GMF was further generalized by different authors in different directions. The interested reader is redirected to the following works for more information, [Baruah, 2003], [Moyo et al., 2010], [Baruah, 2010].

the schedulability of the tasks already assigned to this core in Phase 1 (Task assignment) and Phase 2 (Offline Scheduling).

In the proposed approach, all the allocation decisions are made at design time. During runtime, only a selected subset of cores is allowed to execute the few tasks that migrate and this decision is made based on the multiframe task's execution pattern. This behaviour has a direct consequence on the number of migrations when compared to a fully global approach. As only a few tasks migrate in our approach, the number of migrations is reduced. In addition, this number is further reduced by considering task-level migration instead of job-level migration.

4.8.1 Task Assignment Phase

In the task assignment phase a variant of the first-fit decreasing (FFD) heuristic is proposed, hereafter referred to as FFDO. FFDO first divides tasks into two classes:

1. Light tasks with a density $\lambda_i \leq 0.5$.[6] This class most likely consists of sequential tasks and parallel tasks for which work-stealing is of little interest as the gain obtained by load-balancing the workload is small.

2. Heavy tasks with a density $\lambda_i > 0.5$. This class most likely consists of tasks for which the gain relative to applying work-stealing is high.

The next step is to apply the classical FFD to light sequential tasks first and then to heavy sequential tasks. After this step is completed, FFDO selects the light parallel tasks and then the heavy parallel tasks, again using FFD as the packing heuristic. Intuitively, by assigning sequential tasks first followed by the parallel tasks, the probability of having parallel tasks unallocated after the first phase increases.

All the tasks successfully assigned to the cores are referred to as *non-migrating tasks* and the remaining tasks, *i.e.,* those that cannot be assigned by the heuristic to any core without jeopardizing its schedulability, are referred to as *candidate migrating tasks*.

At the end of the assignment phase, if all tasks are assigned to cores, then there are no candidate migrating tasks and therefore no migrating task in the system. In this case there is no need for parallelisation and work-stealing as a fully partitioned assignment of the tasks to the cores has been found. Using work-stealing in this situation would just help in load-balancing the execution workload at the cost of allowing for unnecessary migrations. Due to this observation work-stealing is forbidden for non-migrating parallel tasks. In the other case, if a task cannot be assigned to any core without jeopardizing its schedulability, then this task is deemed as a candidate migrating task and is treated as a multiframe task. The system is deemed schedulable if and only if an execution pattern is found for each candidate migrating task such that all the timing requirements of the system are met.

[6] The threshold for classifying tasks varies in the literature, nevertheless a density of 0.5 is usually regarded as a good threshold for classifying tasks.

The goal of this assignment is to increase the possibility of benefiting from parallelism during the third phase of the approach as a way to reduce the response-time of the tasks. For instance, assuming that some parallel tasks do not fit into the cores in this first phase, then such tasks can be re-checked in the second phase by treating them as multiframe tasks. If an execution pattern is found for each of the multiframe tasks (meaning that the system is schedulable using such execution pattern), then these tasks may benefit from work-stealing during the online phase of the approach.

4.8.2 Offline Scheduling Phase

After the task assignment phase, let τ^{π_j} denote the set of tasks assigned to core π_j (with $1 \leq j \leq m$). It follows that $\tau^{\pi_j} = \tau_{\text{NM}}^{\pi_j} \cup \tau_{\text{M}}^{\pi_j}$ where $\tau_{\text{NM}}^{\pi_j}$ denotes the subset of non-migrating tasks and $\tau_{\text{M}}^{\pi_j}$ denotes the subset of migrating tasks assigned to π_j.

Concerning the migrating tasks, these are modelled as multiframe tasks and consequently, their jobs are distributed among the cores by following an execution pattern that does not jeopardize the schedulability of each individual core. To compute this pattern, the number of frames of each migrating task is computed as follows.

Definition 6 (Number of frames (taken from [Dorin et al., 2010])). *The number of frames k_i to consider for each migrating task τ_i is computed as:*

$$k_i \stackrel{\text{def}}{=} \frac{H}{T_i}, \text{ where } H \stackrel{\text{def}}{=} \text{lcm}_{\tau_j \in \tau}\{T_j\} \tag{4.1}$$

In Equation 4.1, $\text{lcm}_{\tau_j \in \tau}\{T_j\}$ denotes the *least common multiple* of the periods of all the tasks in τ, also known as the task set's *hyperperiod*.[7] Goossens et al. [Goossens et al., 2012] proved that this number of frames per migrating task is conservative and safe.

Definition 7 (Execution pattern (taken from [Dorin et al., 2010])). *The job-to-core assignment sequence ς of each migrating task τ_i is defined through k_i sub-sequences as $\varsigma \stackrel{\text{def}}{=} (\varsigma_1, \varsigma_2, \ldots, \varsigma_{k_i})$ where the sub-sequence ς_s (with $1 \leq s \leq k_i$) is given in turn by the m-tuple $\varsigma_s = (\varsigma_s^1, \ldots, \varsigma_s^m)$. By following a uniform job-to-core assignment, the sth job of task τ_i is assigned to core π_j if and only if:*

$$\varsigma_s^j = \left\lceil \frac{s+1}{k_i} \cdot M[i,j] \right\rceil - \left\lceil \frac{s}{k_i} \cdot M[i,j] \right\rceil = 1 \tag{4.2}$$

To the best of our knowledge the uniform assignment given by Equation 4.2 is the best result found in the literature for finding execution patterns for migrating tasks. An alternative approach is the generation of patterns via enumeration. Equation 4.2 is part of a set of algorithms that were proposed in [Dorin et al., 2010] for the finding of patterns for multiframe tasks. The intuitive idea of these algorithms is to find the largest number of jobs that can be assigned to each core such that a migrating task is deemed schedulable. Besides schedulability, another advantage of such a job-to-core assignment lies in its ability to considerably reduce the number of task migrations

[7] The hyperperiod is the minimum interval of time after which the schedule for the task set repeats itself.

when compared to a pure global approach. While in a pure global approach jobs may migrate between cores without any restriction, with these algorithms each core knows exactly which jobs it must execute after assignment, therefore bounding the number of migrations.

For the sake of completeness, let us describe the algorithm (from [Dorin et al., 2010]) that computes the execution pattern for each migrating task. The algorithm works as follows:

1. In order to track the current job-to-core assignment, a matrix of integers $M[1\dots n, 1\dots m]$ is used where $M[i, j] = x$ means that x jobs of task τ_i out of k_i will execute on core π_j ($1 \leq i \leq n$ and $1 \leq j \leq m$).

2. The matrix $M[i, 1]$ is first initialized to k_i, *i.e.,* all jobs of τ_i are assigned to the first core. Obviously, this assignment is not schedulable, otherwise the migrating task would be assigned to this core in the task assignment phase.

3. The number k_i is decremented by one unit (*i.e.,* $M[i, 1] := k_i - 1$) and an execution pattern for this number of k_i jobs is computed by applying Equation 4.2. For each specific execution pattern, the schedulability of the system is checked and as long as the task is not schedulable, the value of $M[i, 1]$ is decremented. At some point, say when $M[i, 1] := k_i - \alpha_{[i,1]}$ (with $1 < \alpha_{[i,1]} < k_i$) and the system becomes schedulable, $M[i, 1]$ jobs of task τ_i are assigned to this core. An execution pattern which does not jeopardize the schedulability of the core is found and the algorithm moves on to the next core.

4. The number of jobs just allocated (*i.e.,* $M[i, 1]$) is subtracted from k_i and the result is considered as the new value of k_i in Equation 4.2 for the new core, *i.e.,* $k_i := k_i - M[i, 1]$. Step 3 is executed again in order to find a pattern in the new core considering the new value of k_i. This iterative process is performed for all the cores until all the jobs are assigned to a core. Otherwise, the algorithm keeps reducing the value of k_i in step 3 until a number of jobs (eventually zero) can be accommodated in the current core.

At the end of these steps, if all the jobs of τ_i are not allocated, then τ_i is not schedulable, even as a migrating task, and thus the system is deemed not schedulable. Otherwise, τ_i is schedulable and deemed as a migrating task.

The result in [Dorin et al., 2010] was integrated into our approach.

4.8.3 Online Scheduling Phase

This phase takes advantage of the computing capability of the multicore platform and the execution pattern of migrating parallel tasks in order to reduce their average response-time at runtime, and consequently that of other tasks assigned to the intervening cores. This reduction is achieved by allowing work-stealing to balance workload during the execution of parallel segments and among cores that share the execution of a migrating task. The cores that share the execution of a migrating task are referred to as *selected cores*.

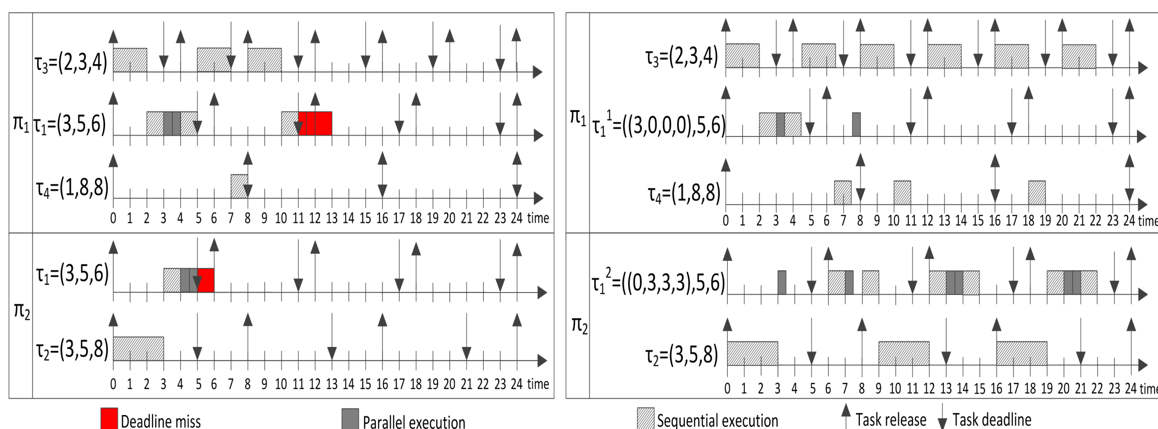


Figure 4.4: Illustrative example of the proposed approach. On the left side of the figure it is possible to observe a schedule under fully partitioned EDF (with deadline miss) and on the right side a schedule with the proposed approach.

Selected cores load the task's code from main memory into the core's local memory (*e.g.,* scratchpad memory) at the beginning of system execution. Then, whenever a new job from a task that is shared between selected cores is released, the core does not have to load its code from main memory as it is already loaded in the core's local memory. *Shared task copies* are used in order to reduce the number of migrations that may occur whenever tasks are scheduled in more than one core.

Below we recall the four necessary rules (R_1 to R_4) for an efficient usage of the work-stealing algorithm when dealing with migrating tasks:

R_1: At least one selected core must be idle when there are p-jobs awaiting for execution in a another's selected core deque;

R_2: Idle selected cores are allowed to steal p-jobs from the deque of another selected core;

R_3: When stealing workload, the idle core must always steal the highest priority p-job from the list of deques (as proposed in Section 4.5) in order to avoid priority inversion (this situation occurs when the number of migrating tasks is greater than 1 and tasks have different priorities);

R_4: After selecting a p-job to steal, say from core A to core B, an admission test must be performed on core B to guarantee that its schedulability is not jeopardized by accepting additional workload.

4.8.4 Example

This section illustrates the proposed approach. We consider the task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ with the following parameters ($\tau_i = (C_i, D_i, T_i)$): $\tau_1 = (3,5,6), \tau_2 = (3,5,8), \tau_3 = (2,3,4), \tau_4 = (1,8,8)$. We assume that all the tasks have a sequential behaviour except τ_1 for which the execution consists

of three segments: (*i*) a sequential segment of one time unit, then (*ii*) a parallel segment of two p-jobs of 0.5 time units each, and finally, (*iii*) a sequential segment of one time unit. We assume that tasks in τ are released synchronously and scheduled on the homogeneous platform $\pi = \{\pi_1, \pi_2\}$. Finally, we assume that an EDF scheduler is running on each core.

During the assignment phase, let us assume that tasks τ_3 and τ_4 are assigned to π_1; and τ_2 is assigned to π_2 as they cannot benefit from any parallelism. Then task τ_1 can neither be assigned to π_1 nor to π_2 without jeopardizing the schedulability of the corresponding core. Figure 4.4 (left side) illustrates the schedules in which τ_1 is tentatively assigned to π_1 (there is a deadline miss at time $t = 11$), and to π_2 (there is a deadline miss at time $t = 5$).

Now let us apply our proposed methodology to this task set. There is a single parallel task in the system:

1. *Task assignment phase:* during this phase, τ_3 and τ_4 are assigned to π_1; and τ_2 is assigned to π_2. For the same reasons as in the previous case task τ_1 can neither be assigned to π_1 nor to π_2, so it is considered as a candidate migrating task.

2. *Offline scheduling phase*: during this phase, an execution pattern which does not jeopardize the schedulability of the cores for the migrating task τ_1 is found. Task τ_1 is then treated as a multiframe task on each core with $k_i = 24/6 = 4$ frames and execution patterns $\tau_1^1 = ((3,0,0,0),5,6)$ and $\tau_1^2 = ((0,3,3,3),5,6)$. The interpretation for each execution pattern is the following: the first job of τ_1 executes in core π_1 and the remaining 3 jobs execute in core π_2.

3. *Online scheduling phase*: during this phase, task τ_1 takes advantage of the work-stealing mechanism in order to reduce its average response-time. Indeed, at time instant $t = 3$, core π_1 is executing the parallel segment of task τ_1 and core π_2 is idle with sufficient resources, so it can steal one p-job from the deque of π_1. The same situation occurs again at time $t = 7.5$. Figure 4.4 (right side) illustrates the resulting schedule, the system is schedulable.

4.9 Tasks with Density Greater Than 1

This dissertation only covers tasks with density no greater than one ($\lambda_i \leq 1$). Nevertheless, it is possible to overcome this limitation by recurring to decomposition-based techniques. This section provides an example of task decomposition using the technique proposed in [Lakshmanan et al., 2010].

Decomposition-based techniques ([Lakshmanan et al., 2010; Qamhieh et al., 2014; Saifullah et al., 2011]) traditionally convert tasks with density greater than one into a set of constrained-deadline sequential sub-tasks, each of which has a density no greater than one. These approaches try to avoid parallel structures by serializing parallel tasks as much as possible so that they can take advantage of schedulability techniques developed for sequential tasks.

In [Lakshmanan et al., 2010], the authors propose the so-called *task stretch transform*, an algorithm that uses the available task's slack[8] to proportionally *stretch* (*i.e.,* serialize) parallel sub-tasks or parts of them in what is called a *master string*.

The master string is assigned to a core and has an execution time length equal to $D_i = T_i$. The remaining parallel sub-tasks that cannot be included in the *master string* are assigned intermediate releases and deadlines so that they become constrained-deadline tasks.

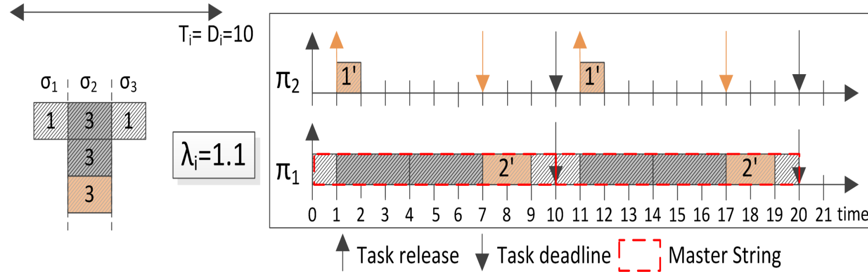


Figure 4.5: Task decomposition with one task

Figure 4.5 illustrates an example of task decomposition. In the example, the task consists of two sequential sub-tasks and three parallel sub-tasks, $C_i = 11$, $D_i = 10$, and therefore $\lambda_i = 1.1$. In order to stretch the task, we compute its slack, assuming an infinite number of cores and no interference from other tasks. In the example, the slack equals $D_i - P_i = 10 - 5 = 5$ time units.

The next step of the algorithm is to proportionally assign the slack to parallel sub-tasks so that they execute sequentially in the master string. Sub-tasks that cannot be completely assigned to the master string either have to be fully parallelised or partly executed in two cores. In the example, two of the sub-tasks execute sequentially in core π_1 in the master string (represented in the figure with the dashed red line) and the remaining sub-task executes partly in core π_2 for one time unit and partly in core π_1 for two time units.

Partial execution of parallel sub-tasks requires the computation of intermediate release offsets and deadlines in order to guarantee execution consistency. For instance, in the example, the sub-task that executes partially in π_2 must complete before it migrates to π_1. Thus, its intermediate deadline is 6 time units after release.

Task stretch transform and similar decomposition-based approaches show that parallel tasks can be scheduled as constrained deadline sub-tasks. This knowledge can be used in the research work proposed in this chapter to support tasks with density greater than one. Thus, by treating a parallel task as a set of constrained-deadline sub-tasks, each of the constrained-deadline sub-tasks can be used as input to an allocation heuristic and consequently, after assigning all the tasks in the set, the schedulability of the task set can be verified.

Figure 4.6 illustrates an example where the above task, let us name it τ_0, is integrated into a task set of four tasks: τ_1, τ_2, τ_3 and τ_4, all with implicit deadlines. Task τ_4 is parallel and the remaining tasks are sequential.

[8] Slack is the maximum amount of time that the remaining computation time of a job can be delayed at a time instant t (with $a_{i,j} \leq t \leq d_{i,j}$, where $a_{i,j}$ is the release instant of job j and $d_{i,j}$ its deadline) in order to complete within its deadline.

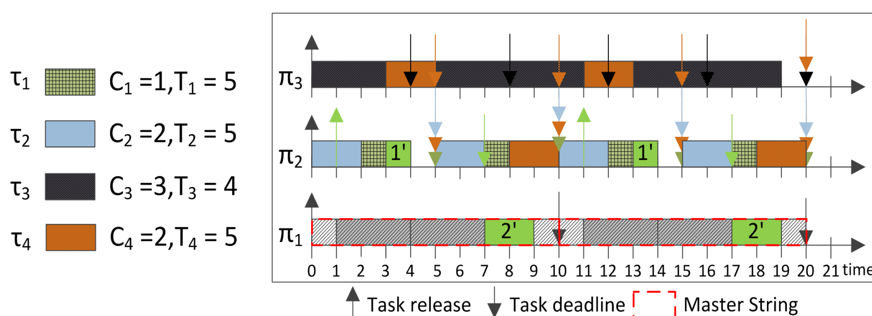


Figure 4.6: Result of applying the proposed approach to a task set that contains a task with density greater than 1

Applying the decomposition-based approach presented above to task τ_0, the result obtained is that task τ_0 has a "stretched" task and a parallel sub-task. The stretched task is exclusively assigned to a core and the parallel sub-task can execute in any core. In this example we opted for allocating the parallel sub-task in core π_2

Then, applying the approach proposed in this chapter, starting by light sequential tasks, followed by heavy sequential tasks, the resulting assignment for sequential tasks is the following: τ_2 and τ_1 in core π_2 and τ_3 is assigned into π_3. Next, the heuristic tries to allocate parallel tasks.

As there is only a single parallel task, there is no need for separating tasks into different classes, and the heuristic selects task τ_4 for allocation. However, τ_4 does not fit in any of the cores and is deemed as a migrating task. Nevertheless, a pattern exists such that the task set is deemed schedulable. Hence, τ_4 is assigned to π_2 with the pattern $(0,1,0,1)$ and π_3 with the pattern $(1,0,1,0)$. The task set is schedulable.

4.10 Schedulability Analysis

This section derives the schedulability analysis of a set of constrained-deadline fork-join tasks executing on a homogeneous multicore platform. A modification of the semi-partitioned model is adopted (see Section 4.8) where each core runs an EDF scheduler while allowing work-stealing among the "selected cores", *i.e.,* cores that share a copy of a migrating task. A schedulability analysis is performed in each phase of the proposed approach and works as follows:

1. *Task assignment phase:* As each processor runs its own instance of EDF, during this phase the schedulability of the system is performed by applying the traditional demand bound function (DBF)-based analysis [Baruah et al., 1990] to non-migrating tasks (as shown in Section 4.7).

2. *Offline scheduling phase*: In this phase, a modified DBF-based schedulability test needs to be used so that the additional workload added to each core, due to the assignment of migrating tasks to cores, is considered in the analysis. In particular, the schedulability analysis needs to consider the execution pattern of each migrating task. Thus, due to the application of the execution pattern assignment algorithms proposed by Dorin et al. [Dorin et al.,

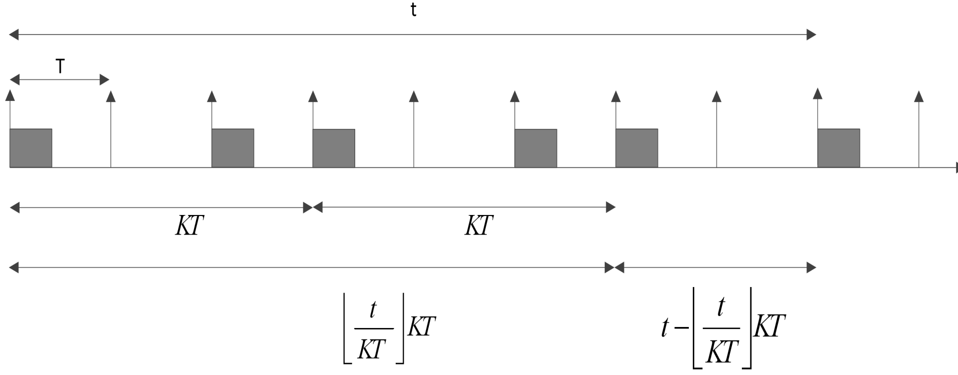


Figure 4.7: Analysis proposed by Dorin et al. in [Dorin et al., 2010].

2010], in our approach we also opted to use their proposed schedulability analysis in this phase.

The analysis proposed in [Dorin et al., 2010] works as follows. First, one needs to compute the number of intervals of length $(k_i \cdot T_i)$ occurring in any interval of length $t \geq 0$, given by $\varphi \stackrel{\text{def}}{=} \lfloor \frac{t}{k_i \cdot T_i} \rfloor$.

Second, the interval $[0,t)$ can be divided into two parts: $[0,t) = [0, \varphi \cdot k_i \cdot T_i) \cup [\varphi \cdot k_i \cdot T_i, t)$, as depicted in Figure 4.7. In the figure, $k_i \cdot T_i = KT$. Consequently, the number of frames that contribute to the additional workload on core π_j consists of two terms: (*i*) The number of non-zero frames in the interval $[0, \varphi \cdot k_i \cdot T_i]$, denoted as $\varphi \cdot \ell_i^j$ (where ℓ_i^j is the number of frames out of k_i that were successfully assigned to π_j); and (*ii*) an upper-bound that considers the execution pattern of the migrating task on the core. That is, one that considers the number of non-zero frames in the interval $[\varphi \cdot k_i \cdot T_i, t)$, denoted as $nb_i(t) = \lfloor \frac{(t \bmod (k_i \cdot T_i)) - D_i}{T_i} \rfloor + 1$. The workload for the first term is given by $\varphi \cdot \ell_i^j \cdot C_i$, while for the second term, the corresponding workload is given by $w_i^j = \max_{c=0}^{k_i-1} (\sum_{\eta=c}^{c+nb_i(t)-1} C_{i,\eta \bmod k_i})$.

It follows that an upper-bound on the total workload associated to task τ_i on core π_j is computed as $\text{DBF}_j(\tau_i, t) \stackrel{\text{def}}{=} \varphi_i \cdot \ell_i^j \cdot C_i + w_i^j$. Consequently, $\text{DBF}(\tau_{\text{M}}^{\pi_j}, t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_{\text{M}}^{\pi_j}} \text{DBF}_j(\tau_i, t)$.

Finally, the schedulability at the end of this phase is guaranteed if:

$$\text{load}(\pi_j) \stackrel{\text{def}}{=} \sup_{t \geq 0} \left\{ \frac{\text{DBF}(\tau_{\text{NM}}^{\pi_j}, t) + \text{DBF}(\tau_{\text{M}}^{\pi_j}, t)}{t} \right\} \leq 1, \forall \pi_j \in \pi \tag{4.3}$$

In Equation 4.3, $\text{DBF}(\tau_{\text{NM}}^{\pi_j}, t)$ represents the demand for the non-migrating tasks assigned to core π_j in the task assignment phase.

3. *Online scheduling phase*: In this phase the schedulability analysis obtained in phase 2 is extended to consider the potential extra workload related to work-stealing. Figure 4.8 illustrates an example of the schedule of a job of a task, say τ_i, on a core, say π_j, after the offline scheduling phase. In this figure, we can see a fork-join task with its fork points (ϕ_1 and ϕ_2), synchronization points (μ_1 and μ_2), and its slack time. In this phase we exploit

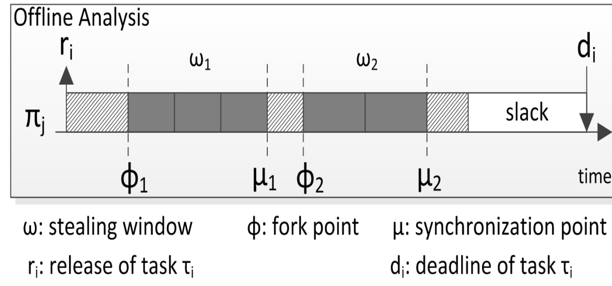


Figure 4.8: Result after the offline analysis. This figure depicts a fork-join task with its fork points (ϕ_1 and ϕ_2), synchronization points (μ_1 and μ_2), and its slack time.

the stealing windows (ω_1 and ω_2 in the example) and the available slack of each job to accommodate the stolen workload.

A work-stealing operation is feasible from one core, say core A, to another core, say core B, if core B can execute the stolen workload (*i.e.,* a p-job stolen from the deque of core A) before the end of each stealing window (μ_1 and μ_2 in the example). Such time instants are denoted as the intermediate deadlines for the stolen p-job.

To compute the intermediate deadline for each stealing window, one can take advantage of the slack available for each job. Thus, the intermediate deadline of the kth stealing window can be computed as:

$$d_{\omega}^{k} \stackrel{\text{def}}{=} \phi_k + m_{i,k} \cdot C_{i,k} + \text{slack}(\phi_{\text{k}}) \tag{4.4}$$

In this equation, ϕ_k denotes the time instant at which the kth parallel segment spawns the p-jobs, $m_{i,k}$ denotes the number of p-jobs spawned in segment $\sigma_{i,k}$, $C_{i,k}$ denotes the worst-case execution time among the tasks in segment $\sigma_{i,k}$, and slack(ϕ_{k}) represents the slack of the job at time ϕ_k.

Figure 4.9 illustrates the computation of the intermediate deadlines for the stealing windows using this equation. In this figure, core π_2 can steal p-jobs from core π_1 in stealing windows ω_1 and ω_2. The intermediate deadline for the p-jobs that may be stolen in ω_1 is computed and the result is d_{ω}^{1}, as depicted in the topmost part of the figure. As the p-job execution takes less time to execute than the intermediate deadline, the stealing operation is successful. Similarly, the intermediate deadline for the p-jobs in ω_2 is computed and the result is d_{ω}^{2}, as depicted in the bottommost part of the figure. For the same reasons as the ω_1 case, the stealing operation is also successful in ω_2.

Before a core, let us denote it as core B, can steal a p-job from another core, let us denote it as core A, an admission control test has to be performed on core B. Two possible scenarios can occur when stealing a p-job released in the kth parallel segment of a task:

- *no release occurs in core B between* ϕ_k *and* d_{ω}^{k}: In this case core B can safely steal a p-job from core A provided that the execution of the stolen p-job meets its intermediate deadline (Case 1 in Figure 4.10);

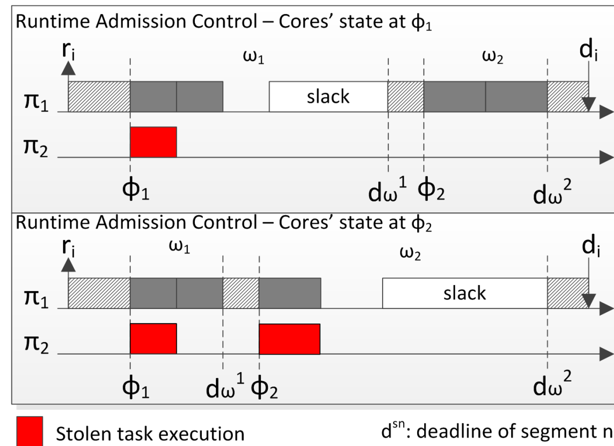


Figure 4.9: Example of work-stealing and intermediate deadline computation. This figure illustrates the computation of the intermediate deadlines in the stealing windows ω_1 and ω_2.

- *at least a release occurs in core B between ϕ_k and d_ω^k*. In this case, we can distinguish two sub-cases. (2.1) some releases have their deadline before d_ω^k: in this sub-case, we should update the idle time interval in the stealing window by subtracting the interference related to the corresponding new job releases from the size of the stealing window (Case 2.1 in Figure 4.10. In the figure task τ_i and τ_j have releases and deadlines within ω_k); (2.2) some releases have their deadline after d_ω^k: in this case, no guarantees can be provided on the schedulability of the system as the stolen job may modify the scheduling decisions initially taken on core B, due to having an earlier deadline than the other tasks released in the core. Therefore no stealing occurs (Case 2.2 in Figure 4.10. In the figure task τ_k has a release in ω_k but deadline outside of the window).

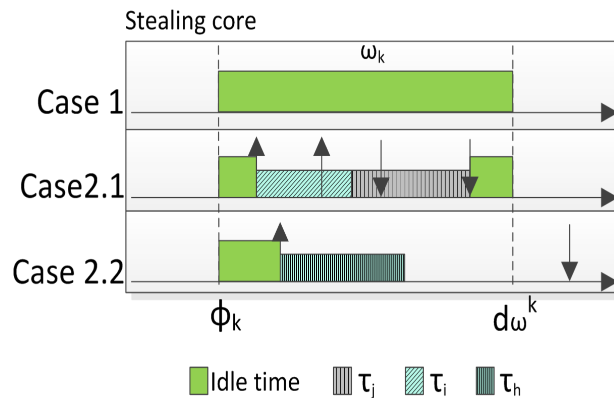


Figure 4.10: Possible cases for the admission control test

4.11 Simulation Results

This section presents the results of simulating our approach on a set of synthetic and randomly generated task sets. The simulation environment is described next.

We consider a platform consisting of two or four homogeneous cores. Tasks are generated until the total utilization of the task set does not exceed the total platform capacity (*i.e.,* $U_\tau \leq m$). While each generated task can be sequential or parallel, the number of each type of tasks generated is not controlled beforehand.

Considering each generated task $\tau_i \in \tau$, the number of segments s_i is selected from the sequence $s_i \in [1,3,5,7]$. When $s_i = 1$ the task is sequential, otherwise it is parallel. In case of a parallel task, the total number of p-jobs is randomly selected in the interval $n_{\text{pjob}} \in [s_i, 10]$. Then, p-jobs are randomly assigned to the segments by taking into account the fork-join task structure, *i.e.,* sequential segments are assigned one p-job and parallel segments are assigned more than one p-job.

The worst-case execution time of the kth p-job in each segment j (*i.e.,* $C_{i,j,k}$) varies in the range $[1, \text{max_Ci_pjob}]$ where $\text{max_Ci_pjob} = 2$. [9]

The worst-case execution time of each task is given by $C_i = \sum_{j=1}^{s_i} C_{i,j}$, where $C_{i,j} = \sum_{k=1}^{m_{i,j}} C_{i,j,k}$. [10]

The remaining parameters, period T_i and utilization U_i, can be derived as follows. The period T_i is uniformly generated in the interval $[C_i, n_{\text{pjob}} * \text{max_Ci_pjob} * 2]$. This interval allows one to have a task utilization (recall that $U_i = \frac{C_i}{T_i}$) that falls in the interval $[0.50, 1]$ if all nodes are assigned max_Ci_pjob, or $[0.25, 1]$ if all nodes are assigned the minimum value for $C_{i,j,k}$.[11] In our experiments $D_i = T_i$.

This task generation procedure is used to generate 1000 task sets with migrating tasks for two and four cores. To generate execution patterns for the migrating tasks we use Equation 4.2 first. If no pattern is found using the equation we follow an enumeration approach in order to find a feasible pattern, if such pattern exists.

4.11.1 Selected Heuristics

In order to evaluate the performance of FFDO, we have conducted benchmarks against other well-known bin-packing heuristics, namely the standard first-fit decreasing (FFD), best-fit decreasing (BFD), and worst-fit decreasing (WFD). All of these heuristics, with the exception of FFDO, group the tasks into sequential and parallel tasks, and subsequently sort each group in decreasing order

---

[9] As we are measuring the improvement in terms of the average response-time of each task by generating real schedules, it is our interest in keeping $C_{i,j,k}$ small in order to make the measurements faster. Moreover, this design decision replicates fine-grained parallelism and we believe that it does not compromise the goal and results of the experiments.

[10] By considering the worst-case execution time for each p-job in the experiments, we are evaluating the benefits of using work-stealing in the worst possible scenario.

[11] As we evaluate the behaviour of each task set in the interval $[0, H]$, where H denotes the least common multiple of the periods of all the tasks in the set, and as T_i in our generation depends on C_i, the higher the C_i, the higher the T_i and consequently, the higher the hyperperiod of the task set. By limiting the time assigned to each p-job in $C_{i,j,k}$ we are also limiting the amount of time we need to generate the schedule.

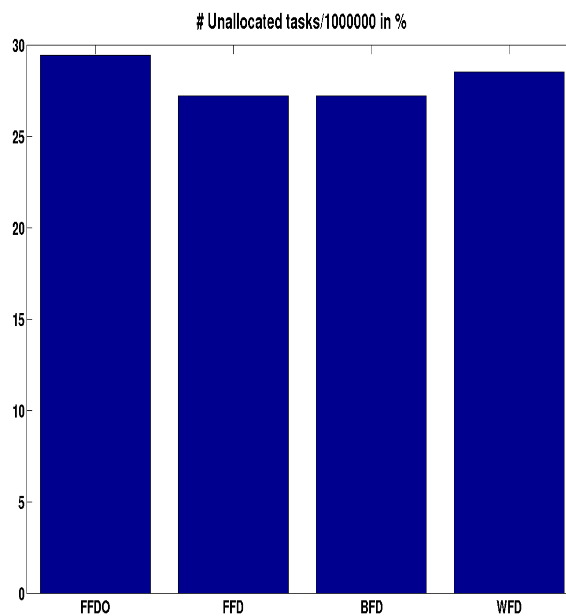


Figure 4.11: Percentage of unallocated tasks

of task utilization. Therefore, aside from the allocation property of each heuristic, again with the exception of FFDO, no special care is taken in further creating the sets of heavy and light tasks.

FFD assigns the next unassigned task into the first core from the set of cores with sufficient idle time to accommodate it; BFD assigns the next unassigned task into the core which leaves the least idle time available after the task is assigned to it; and WFD assigns the next unassigned task into the core which leaves the most idle time available after the task is assigned to it.

The metric selected to compare all the above-mentioned heuristics was the percentage of unallocated tasks. In particular, for a large number of task sets the percentage of unallocated tasks for each heuristic was measured in order to observe which heuristic(s) has(ve) a potential higher number of candidate migrating tasks. [12] To this end, one million task sets were generated for this experiment.

Figure 4.11 depicts the results. It is clear that FFDO and WFD are the heuristics that present a higher number of unallocated tasks, while BFD and FFD allocate nearly the same amount of tasks and, at the same time, present a lower value of unallocated tasks when compared to FFDO and WFD. Due to this result, we selected both FFDO and WFD for a direct comparison in terms of the number of schedulable task sets.

4.11.2 FFDO versus WFD

Two experiments were carried out in order to compare FFDO against WFD, with two distinct goals in mind. The goal of the first experiment was to observe which heuristic schedules more task sets

[12] Recall that the higher the number of candidate migrating tasks, the higher the chance of taking advantage of the proposed approach in the second phase, when some of the candidate tasks can be re-allocated as migrating tasks.

when the same input of randomly generated task sets is taken into account. As for the second experiment, its goal was to measure the gain obtained for each schedulable task set, in terms of the average response-time, when work-stealing is used.

4.11.2.1 First Experiment

For the first experiment a procedure was developed to randomly generate a number of task sets in order to obtain 100 FFDO-schedulable task sets. Then, considering all the generated task sets, the number of WFD-schedulable task sets was measured in order to establish a comparison between both heuristics. The results are depicted in Figure 4.12.

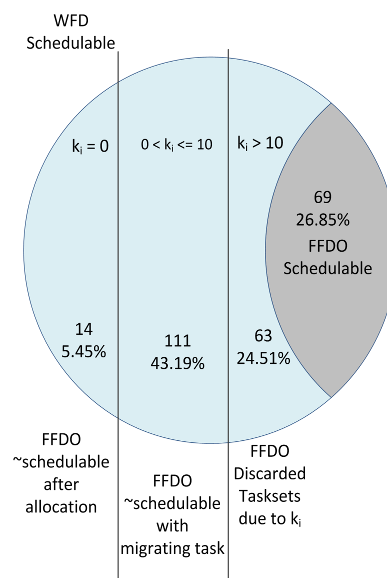


Figure 4.12: Comparison between FFDO and WFD. This figure presents how the data is categorized for WFD when a fixed number of task sets is used as input. For the same number of input task sets used to obtain 100 schedulable task sets for FFDO, we have obtained 257 for WFD.

The WFD-schedulable task sets can be divided into 4 groups (going from right to left on the figure): (1) the group that contains the sets that are schedulable by both heuristics, which account for 26.85%; (2) the group that contains the sets that are not schedulable by FFDO due to k_i[13]. This group contains 24.51% of the sets; (3) the group that contains all the sets that are not schedulable by FFDO with a k_i value in the range of valid values, *i.e.,* with at least one migrating task. These represent 43.19% of the sets; and finally, (4) the group of task sets that are deemed not schedulable by FFDO after the allocation phase (*i.e.,* the 1st phase of the heuristic), which account for 5.45% of the sets.

Overall, in a two-core setting, the total number of task sets that are schedulable by using WFD is 257, which represents an increase of 157% over FFDO for the same input. From the diagram, the majority of the task sets that are schedulable by using WFD fit in a potential feasible region

[13]Task sets that have a number of frames over 10 are rejected as the complexity of computing the migrating patterns increases for large k_i, which leads to higher computation times.

for FFDO heuristic (43.19%) — here, all task sets have migrating tasks and k_i values that fit in the range of valid values but no feasible pattern is found. These results still hold for four cores but to a less extent as only 17.9% more task sets were schedulable by using WFD over FFDO.

We conjecture that WFD behaves better than FFDO for smaller number of cores because of the task-to-core assignment. Depending on the granularity of the utilization of the task sets, more empty space may be available globally in the cores when performing the task allocation for a small number of cores. These idle slots make it possible for the pattern-finding procedure to find enough room to fit a job of a task when computing the execution pattern for a migrating task. However, as the number of cores increases, WFD naturally balances the workload through the cores, whereas FFDO assigns the workload in the initial cores leaving more room in later cores. For this reason, we envision that WFD will have the tendency to behave either equally to or even worse than FFDO with the increase in the number of cores.

4.11.2.2 Second Experiment

In the second experiment, the gain obtained in terms of the average response-time for each schedulable task set was measured for the selected heuristics, namely FFDO and WFD. Specifically, for each task set, the complete schedule is generated considering the two following approaches:

- an approach that schedules migrating tasks without applying the work-stealing mechanism among the selected cores, denoted as Approach-NS;
- an approach that applies the work-stealing mechanism among the selected cores, denoted as Approach-S.

After generating both schedules for each task set, we computed the average response-time of the jobs of each task throughout the hyperperiod by adding the response time of each individual job and dividing the obtained result by the total number of jobs in one hyperperiod. This process is applied to both approaches.

The improvement, *i.e.,* the gain of Approach-S over Approach-NS is computed by applying the following formula for each task τ_i:

$$AV_{\tau_i} = \frac{AV_{\tau_i}^{NS} - AV_{\tau_i}^{S}}{AV_{\tau_i}^{NS}} \cdot 100 \tag{4.5}$$

, where $AV_{\tau_i}^{NS}$ denotes the average response-time for task τ_i in Approach-NS and $AV_{\tau_i}^{S}$ denotes its average response-time in Approach-S. The average gain for each task in the task set is computed as follows:

$$AV_{\tau} = \frac{1}{|\tau|} \cdot \sum_{\tau_i \in \tau} AV_{\tau_i} \tag{4.6}$$

Figure 4.13 illustrates the average gain for two and four cores.

The improvement in terms of average response-time per task (in %) is grouped by utilization — see Figure 4.13 — when using Approach-S over Approach-NS. For all figures, the distribution

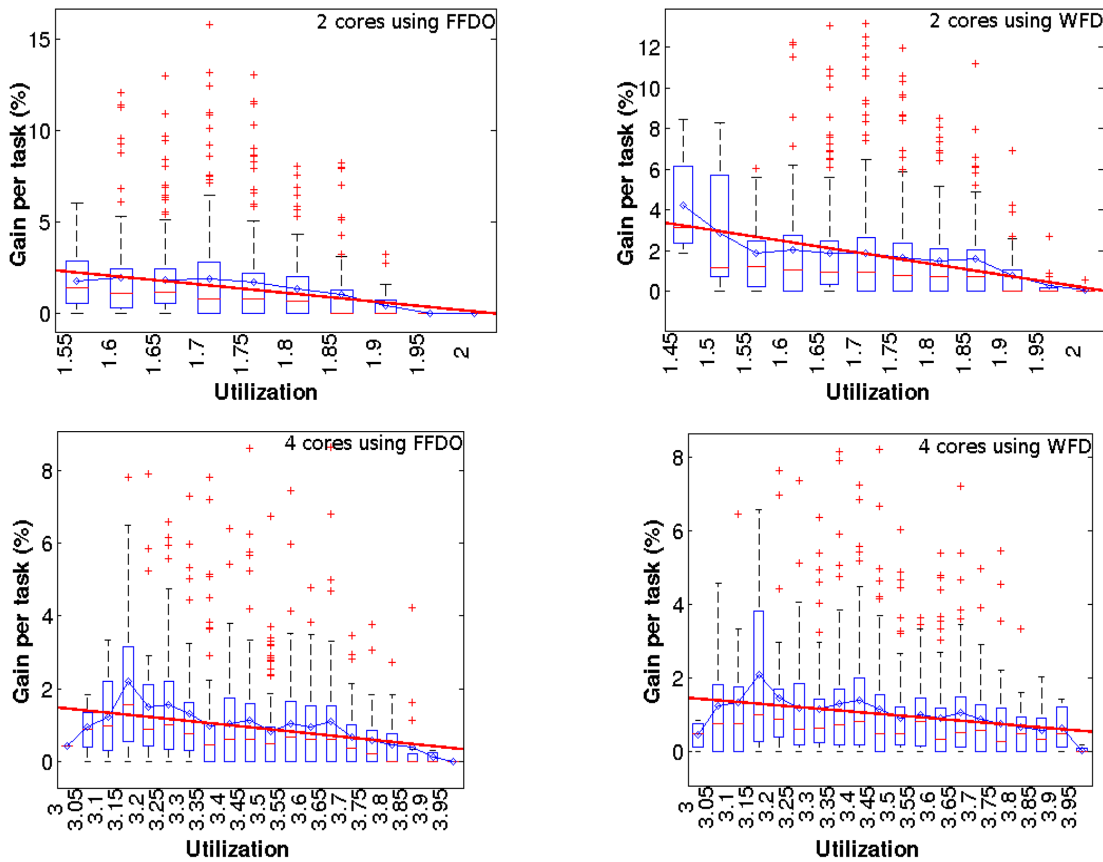


Figure 4.13: Simulation results for FFDO and WFD. This figure presents the improvement in terms of the average response-time when using an approach with work-stealing and an approach without work-stealing. On the top, one can see the results for two cores and on the bottom one can see the results for four cores. On the left side, the results show the improvement for FFDO and on the right the improvement for WFD.

of data is depicted in the form of box plot. In the plot, for each utilization value, it is possible to see the minimum and maximum values of gain per task, the median and the mean (in the form of a diamond shape), the first and third quartiles and finally the outliers in the shape of a cross. The line in red depicts a linear regression on the data (the mean value was used to compute the regression) in order to depict the pattern of prediction of the gain per task.

Considering two cores: for task sets with high utilization (over 1.55), there is a clear illustration of the gain obtained by using work-stealing. In the best case, the gain reaches nearly 15% for FFDO and nearly 12% of the average response-time per task for WFD. As the utilization of the task sets increases the gain per task decreases. This is expected due to the reduction of idle time available for stealing. The trend shows that above 1.95 of utilization, the work-stealing mechanism becomes of little interest. This is explained by the fact that the total workload on each core is very high, thus leaving very small room for improvement. It is important to note that task sets with utilizations below 1.55 using FFDO and 1.45 using WFD are not included in the plot as they do not contain any migrating task.

Considering four cores: the trend is similar to the one depicted for two cores. This trend is also shown by the linear regression line where it is possible to predict the average gain per task as a function of the utilization of the task set. The regression shows that for lower utilizations in two cores the expected improvement starts at 2.3% for FFDO and 3.3% for WFD. For four cores it starts at 1.4% for both heuristics. We can also observe that the expected improvement decreases with the increase in the tasks' utilization. This behaviour suggests that work-stealing is useful for task sets with migrating tasks with an utilization that span from the lowest possible utilization for task sets with migrating tasks up to the platform capacity. Closer to this upper limit, the benefits of using work-stealing are limited. From the observed behaviour in two and four cores, we conjecture that the proposed approach will behave similarly when the number of cores increases.

Results in Figure 4.13 also show that, when the set of schedulable task sets is considered, the improvement obtained by using work-stealing is similar in both heuristics. We believe that this behaviour is obtained due to the characteristics of the generated jobs, in particular, the fine-grained parallelism of the generated p-jobs. However, achieving a decrease in the response-time of the tasks is dependent on several factors, as for instance: the degree of parallelism in a parallel segment; or the WCET of p-jobs in a parallel segment; or the fact that there must be an idle core available and a migrating parallel task executing at the same time instant that a core that shares it is idle. We expect that if larger WCET are used for the p-jobs, the improvement in the response-time will be larger but, at the same time, the number of task sets benefiting from work-stealing decreases because of the job's deadlines and difficulty in keeping the deadlines in the thief core when stealing occurs.

4.11.3 Overheads of the Approach

Based on the results presented in the previous section, one can observe that it is possible to decrease the average response-time of tasks and use the newly created free time slots to execute less critical tasks (*e.g.,* aperiodic or best-effort tasks). While such a decrease presents overhead

costs, such as the number and cost of migrations, or even the impact of online admission control, these costs were not explicitly measured as they are difficult to model in a simulation environment. Nevertheless, an overview of the existing costs and their possible impact on system performance is provided next.

We assume that cores that share a migrating task have a local copy of it in order to prevent fetching the tasks' code from main memory. Thus, instead of fetching data and code, by using task copies, a core fetches data from another core's memory in order to help in the execution of the migrating task. While this is not a task migration *per se*, it has some commonalities as data needs to be moved from one core to another.

As in this work the overhead of fetching data from another core only occurs when stealing occurs, and stealing is performed by an idle core, part of the cost is supported by the idle core (which is negligible due to the idleness of the core). Considering the number of data transfers, it can be bounded as in the worst-case the number of data fetches depends on the number of p-jobs in each segment and the number of cores that share the task.

Another aspect that needs to be considered is that keeping task copies is platform dependent and in some platforms it might not be possible to save local copies due to memory constraints. Moreover, depending on where in memory the copy is stored (for instance, scratchpad, cache, *etc.*), the transfer may be subject to or cause interference in the execution of other tasks in the system (for instance due to the existence of shared resources)[14]. Interference was not considered in the work presented in this chapter.

Considering the online admission control, our test requires the current time instant and the available slack at that specific time instant. Both of these variables can be easily computed in any given platform either by using the platform timing functions and a cumulative function that computes the slack for the current job. Therefore, we consider that this does not pose any significant overhead in the proposed approach.

4.12 Summary

All the works mentioned in Section 4.4 show that using a global centralized approach for scheduling parallel tasks is not beneficial due to the amount of synchronization that must be performed in order to maintain a consistent state in the system. Consequently, a fully decentralized scheduling approach or a combination of both (centralized and decentralized) should be used in order to make the system scalable. Moreover, all of them agree that work-stealing should be modified in real-time systems.

This chapter addressed the application of work-stealing into real-time systems. As it was shown in Section 4.3, the non-deterministic behaviour of work-stealing, that includes random steals and the possibility of priority inversion, makes it difficult to be applied in the real-time systems domain without jeopardizing the schedulability of the system. Thus, in order to circumvent

[14]This problem is discussed in Chapter 2 and Chapter 5 of this dissertation.

this non-deterministic behaviour, a new data structure and a set of rules are proposed (in Section 4.5). In addition, this real-time variant is integrated in a semi-partitioned approach along with the multiframe task model.

The proposed approach has two distinct stages - an offline and an online stage. The offline stage has the objective of allocating all the tasks into the cores using a partitioned approach in combination with a global approach that uses the multiframe task model. The added benefit of using the multiframe task model lies in the reduction of the number of task migrations when compared to a full global approach. The online phase uses work stealing as a way to improve system responsiveness by applying load balancing to parallel tasks.

Results show that with this technique it is possible to reduce the average response-time of tasks and create additional room in the schedule for less-critical tasks (*e.g.,* aperiodic and best-effort tasks). In particular, the proposed approach allows one to achieve an average gain on the response-times of the parallel tasks between 0 and nearly 15% per task.

For future work, we would like to continue pursuing the exploration of this idea of work-stealing in real-time settings as it appears to be very promising. In particular, considering a pure global setting, which is where work-stealing may perform as a natural load-balancing scheduling approach. Other possibilities involve using parallel task models with nested parallelism.

In the subsequent chapter, we continue to explore the parallelism provided by current multicore systems, however in a more specific setting. Thus, the notion of interference is considered and a new task model is introduced - the 3-phase task model.

Chapter 5

Schedulability of the 3-Phase Task Model

5.1 Introduction

The 3-phase task model is a good candidate model to circumvent the uncontrolled sources of interference existing in current Commercial Off-the-shelf (COTS) platforms by isolating concurrent memory accesses. In this model a task is divided in three successive phases: in the first phase, the task loads its instructions and data into a core's local memory, then, in the second phase, it executes non-preemptively using those pre-loaded instructions and data, and finally, in the third phase, the modified data are pushed back to main memory. Following this execution model, tasks never access the bus during their execution phase. Instead, all the bus accesses are performed during the first and third phases.

This model provides two interesting properties from a predictability viewpoint. First, as at most one task can perform memory accesses at a time, memory contention related issues (such as uncontrolled interference) are avoided. Second, by decoupling memory phases from the execution phase it is possible to exploit platform parallelism. The model allows for execution phases of different tasks to execute in parallel with any other phases of co-running tasks. Thus, system predictability is improved by having memory phases and execution phases of different tasks executing in parallel while avoiding, at the same time, memory contention issues when tasks access main memory. Both of these properties make the model suitable for real-time and embedded multicore systems.

This chapter is divided in two parts. The first part (in Section 5.4) presents an empirical study that compares the performance of different priority assignment policies considering the 3-phase task model against an implementation of global Earliest Deadline First (EDF) scheduling policy that considers inter-task interference.

The second part (in Section 5.5) presents a schedulability test for the 3-phase task model. The proposed schedulability test improves current state of the art's test by looking at the schedulability

problem of the 3-phase task model from a different perspective. That is, instead of analysing the system following the standard's core's perspective, a bus perspective is used instead.

5.2 System Model

We consider a system composed of m identical cores where each core accesses the system's main memory using a shared bus. From a core's perspective the shared bus is a shared resource. Consequently, the bus is a source of interference whenever concurrent accesses are made by different cores to fetch data from main memory into the core's local memory.

We assume that Input/Output (I/O) data transfers from or to the main memory are performed using a Direct Memory Access (DMA) controller. We also assume that the local memory of each core is large enough to save any task's code and data. If this is not the case, the task should be divided in smaller entities, each entirely fitting in the core's local memories. At any time, at most one task can be saved in each local memory.

Task Model: We consider a system composed of n independent real-time tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task comprises three distinct phases, namely, the acquisition (A), execution (E) and restitution (R) phases. Phases have a precedence constraint in the sense that a job must first execute its A-phase, then its E-phase and finally, its R-phase. Each phase executes *non-preemptively*.

We let A_i, E_i and R_i denote the maximum execution time of the A, E and R-phase of task τ_i, respectively. The worst-case execution time (WCET) in isolation of τ_i (without suffering any interference) is given by the sum of the execution times of each phase, *i.e.,* $C_i = A_i + E_i + R_i$. Each task is characterized further by a period T_i and a constrained-deadline $D_i \leq T_i$. That is, the A-phases of every two successive jobs of τ_i are released at least T_i time units apart, and the R-phase of a job of τ_i must complete at most D_i time units after the release of the A-phase of that same job. Therefore, for a task to be schedulable, its WCET should be no greater than its relative deadline, *i.e.,* $C_i \leq D_i$.

As defined in Chapter 2, the *utilization* of task τ_i is given by $U_i \overset{\text{def}}{=} \frac{C_i}{T_i}$ while the *total utilization* of the task set τ is given by $U_\tau \overset{\text{def}}{=} \sum_{i=1}^{n} U_i$. Moreover, the *memory utilization* by a task τ_i is given by $M_i \overset{\text{def}}{=} \frac{A_i+R_i}{T_i}$. To ensure the feasibility of the system, the core's utilization should not exceed 100% and consequently, the total system utilization should be no greater than the number of cores in the system, *i.e.,* $U_\tau \leq m$. Similarly, the system bus utilization should not exceed 100%, *i.e.,* $\sum_{i=1}^{n} M_i \leq 1$.

Shared Resource Model: The shared resource covered in this chapter is the system bus. Specifically, whenever a task executes a memory phase (either A or R), one of the cores locks the bus and initiates a DMA request to fetch/store data from/into main memory. The core releases the bus at the end of the memory phase. Therefore, memory phases are non-preemptive and at most one task executes a memory phase at any time instant.

5.3 Runtime Execution Model

A and R-phases are *memory phases* during which each application transfers data between main memory and the core's local memory (*e.g.,* scratchpad, L1 cache). When an A-phase starts, the code and data needed for the task's execution are fetched from main memory into the core's local memory. After completing execution, the R-phase pushes back to the main memory the data modified by the task that were saved in the core's local memory. To avoid interference, memory phases require exclusive access to the system bus. A task will therefore lock the access to the bus whenever it executes a memory phase.

According to this execution model, a task does not require any access to the bus during its E-phase and hence does not suffer unpredictable interference due to tasks executed on other cores. In addition, any E-phase can execute in parallel with any other phases of other tasks executing in the system without interfering with their execution.

5.4 3-Phase vs. G-EDF in COTS Systems

The priority assignment problem is one in which the relative priority ordering of a set of tasks needs to be determined. In fixed task priority systems, heuristic approaches for task priority assignment can be used to try to obtain a feasible solution for the scheduling problem in a reasonable amount of time without having to test all possible $n!$ priority orderings. Recall that multicore scheduling is proven to be a NP-hard problem [Garey and Johnson, 1990]. That is, without testing all the possible combinations of task orderings, there is no means of knowing which ordering leads to a schedule where all tasks are schedulable and the length of the schedule is minimized.

A possibility for testing the schedulability of a system of periodic tasks following a given priority ordering is to generate the actual schedule over the hyperperiod interval of the task set (as defined in the previous chapter, the hyperperiod is the minimum interval of time after which the schedule for the task set repeats itself) and validate if some deadline miss occurs at any time instant within this interval [Cucu and Goossens, 2006], [Cucu and Goossens, 2007]. Using this method, we empirically explore the 3-phase task model by comparing different priority assignment policies (all of them for the 3-phase task model) against a version of global EDF that takes intertask interference into account. Specifically, the goals of this empirical study are: (1) simulate the behaviour of all approaches in a COTS multicore system under global scheduling; (2) observe if any of the proposed priority assignment policies performs better than any other; (3) compare the interference-prone global EDF version against the other proposed policies.

5.4.1 Priority Assignment Policies

The proposed assignment policies are the following:

- **Priority**: The priority of each task in the task set is given by the task period T in non-decreasing order. That is, tasks with smaller periods have higher priority, similarly to the behaviour of the Rate Monotonic algorithm [Liu and Layland, 1973].

- **Minimum Acquisition**: The priority of each task is given by the length of its A-phase. The scheduler selects for execution the job of the task that has the *smallest* Acquisition value among the ready jobs (*i.e.,* jobs that are waiting to access a processor and/or the bus).

- **Maximum Acquisition**: The priority of each task is given by the length of its A-phase. The scheduler selects for execution the job of the task with the *largest* Acquisition value among the ready jobs.

- **Minimum Restitution**: The priority of each task is given by the length of its R-phase. The scheduler selects for execution the job of the task with the *smallest* Restitution value among the ready jobs.

- **Maximum Restitution**: The priority of each task is given by the length of its R-phase. The scheduler selects for execution the job of the task with the *largest* Restitution value among the ready jobs.

In order to understand how the above-mentioned policies compare to an approach that is interference-prone, we propose to compare them against a modified global EDF version that considers inter-task interference due to shared resources. Thus, the standard global EDF version is adapted to accommodate inter-task interference so that we can infer how a global scheduling approach that considers interference behaves when executing in a COTS system.

Standard global EDF algorithm gives priority to the m jobs that have the earliest absolute deadline among all the jobs in the ready queue (a ready queue is a queue of ready jobs). In addition, due to its global behaviour, inter-processor migration is allowed and therefore, any job is allowed to execute in any core of the system, subject to the restriction that it may execute on at most one processor at any given time instant.

The modified global EDF version proposed in this dissertation includes two important modifications that model task execution in current multicore COTS platforms. In these platforms, tasks execute without temporal and spatial isolation and therefore, suffer interference whenever concurrent accesses are made to shared resources. The two proposed modifications are the following.

First, each 3-phase task in the set is converted into a *traditional task*. A traditional task is a task in which all the 3-phases are merged into a single phase and consequently, memory operations are not decoupled from task execution. In the merging process, the WCET of the 3-phase task is reduced by a certain amount[1]. The motivation for a reduction in the WCET relies on the assumption that 3-phase tasks are obtained from traditional tasks. During this transformation process, traditional tasks need to be modified in order to accommodate the code necessary for the

[1] In our experiments, the reduction factor is an input parameter that the user can modify to increase or decrease the traditional task's execution time.

decoupling of memory operations from execution (decoupling the phases allows us to achieve an interference-free deployment). Intuitively, by transforming a traditional task into a 3-phase task there is an increase in the WCET of each 3-phase task due to the extra code needed for phase decoupling. Therefore, for fairness in comparison of both task models (the 3-phase task model and traditional task model) we apply a reduction to the WCET when converting a 3-phase task into a traditional task.

Second, task execution is artificially slowed down by a factor whenever traditional tasks execute in parallel with each other, under the assumption that when a task executes according to modified global EDF it executes without temporal and spatial isolation.

5.4.2 Simulator's Scheduling Behaviour

In order to reason about the behaviour of 3-phase tasks, we developed a simulation tool that generates offline schedules for each of the priority assignment policies presented above and the modified version of global EDF. The goal is to compare the modified version of global EDF against the assignment policies. The tool considers the system and runtime models described in sections Section 5.2 and Section 5.3.

The runtime behaviour of the simulator is described next. The scheduler considers the following rules in the assignment: (1) memory phases can only be assigned to a core if no other core is executing a memory phase (either A or R phases); (2) E-phases never wait to execute once they become ready, therefore they start executing as soon as the preceding A-phase completes its execution.

For any given priority assignment policy (except modified global EDF), the next memory phase to execute is selected from a priority ordered queue containing ready memory phases. From this queue, the scheduler selects the highest priority phase and assigns it to an idle core, if allowed. If the selected phase cannot be assigned to a core, two outcomes are possible. First, if a memory phase is being executed already, the scheduler needs to wait until the next memory phase can be assigned. Second, if all cores are busy executing phases from other jobs than the one being assigned, the scheduler assigns the next higher priority R-phase from one of the currently executing jobs, once the R-phase becomes ready.

Once a job is assigned to a core it remains executing exclusively on that core until all the job's phases complete their execution. That is, no other job can be assigned to a core that already started executing another job. The intuition for this behaviour is that memory phases isolate memory accesses so that tasks can execute without suffering interference. Having tasks moving around from core to core would improve system utilization but at the same time would break the principle of isolation, causing interference and additional overheads due to the task migration occurring between cores. The same reasoning applies if preemptions were allowed after a task started its execution.

Our modified global EDF version considers traditional tasks (as described above) and therefore no distinct phases exist within a task. At any given time instant, the m higher priority tasks are the ones executing in the system. As it typically occurs in global EDF, preemptions are allowed

due to the arrival of a higher priority jobs in the system. In our simulator we do not consider any overhead related to such preemptions. Higher priority tasks may interfere with lower priority tasks but a lower priority task may never block a higher priority task.

5.4.3 Experimental Settings

The task generation in the simulator works as follows. A base task set containing n tasks is generated and its schedulability tested. Then, if the task set with n tasks is schedulable, a new task is added to the base task set so that a new task set is obtained with $n+1$ tasks. The schedulability of this new task set is tested and the procedure continues by adding a new task to the set. This iterative procedure stops when the new task set is deemed unschedulable by all the scheduling policies under test.

The parameters used in the generation of 3-phase tasks are described next. Each phase has an initial execution time value randomly generated in the interval $[1,3]$. This value is multiplied by a factor selected from the interval $[1,3]$ in order to test different configurations of phase lengths.

The WCET for each 3-phase task is given by the sum of the individual phases' execution times, as presented in Section 5.2. The period of each task is obtained by randomly generating a factor value in the range $[2,4]$ which is multiplied by the WCET of the task. For each task, the deadline equals the task period.

For global EDF tasks (also denoted as traditional tasks), the simulator converts each generated 3-phase task into a traditional task in which the respective 3-phase task's WCET is reduced by a certain amount defined as an input parameter. In our simulations we have chosen to reduce the WCET of a 3-phase task by 25%. As explained in Section 5.4, this reduction intends to mimic the increase in task execution time when a traditional task is converted to a 3-phase task due to the addition of code needed to enforce phase decoupling.

Another important aspect that we consider in the simulator, when dealing with global EDF tasks, is the interference that may occur in a COTS platform due to shared resources. In [Nowotsch and Paulitsch, 2012] the authors observed a maximum slowdown of 5.1x in application execution when multiple devices access network and memory concurrently in a platform of $m=4$ cores. In the same line of research, the authors in [Girbal et al., 2015] state that the slowdown that a task suffers when moving from a single-core configuration to a multicore configuration is of 2.7x in a cached-based version of a multicore platform for $m=6$. Therefore, in order to emulate this behaviour when scheduling tasks under global EDF, each traditional task is slowed down at runtime. In particular, the slowdown value applied depends on the number of tasks that execute in parallel in the same time unit. As a side note, as the 3-phase task model is an interference free model, no slowdown needs to be applied to it.

Using the values reported in [Girbal et al., 2015] as a reference (*i.e.,* 2.7x), we can compute an input list of slowdown values (by direct proportion) for m cores by applying the following expression, $S_m = \frac{m \cdot 2.7}{6}$ for $m \geq 2$. Thus, for $m=4$, the list of slow down values becomes $[1,1,1.35,1.8]$. Each value in the list represents the factor of increase in WCET that a traditional task will incur, due to interference, when it executes in parallel with other tasks in the system, in a given time

unit. For instance, when a task executes in parallel with two other tasks, the increase in its WCET will be 1.35x per time unit. Thus, instead of taking 1 time unit to execute, the considered task takes 1.35 time units due to interference. A special remark must be made concerning the first two values in the list: the first value represents the time in isolation (without any interference); while the second value should be 0.9 after computing the slowdown value, but that leads to a speedup in execution instead of a slowdown. Our decision was to round this value to the closest integer[2]. The slowdown value is one of the parameters that we vary in our simulations so that we can evaluate its impact on task schedulability.

As output, the simulator generates a schedule for each of the scheduling policies described in Section 5.4.1; the average response-time and maximum response-time for all the generated task sets; and a plot containing the number of schedulable task sets per utilization value, for each scheduling policy.

5.4.4 Experimental Results

In this section we discuss the results obtained for the scheduling of randomly generated task sets using the different scheduling policies described in Section 5.4.1.

The chosen metric used for scheduling policy comparison is the percentage of schedulable task sets that each policy can schedule per utilization value (*i.e.,* the schedulability ratio). Therefore, in the experiments we evaluated the percentage of schedulable task sets (y-axis) per utilization value (x-axis) for 2000 iterations of randomly generated task sets using different settings. In all the experiments the base task set contained $n = 2$ tasks.

Concerning the slowdown values used in the experiments, we have chosen slowdown values of 1.5x and 2x the values in the slowdown list presented above. Accordingly, the slowdown list values become $[1, 1.5, 2.03, 2.7]$ and $[1, 2, 2.7, 3.6]$, for 1.5x and 2x respectively.

The following settings were used for $m = 2$ and $m = 4$: (1) a setting where the E-phase is smaller than both A and R-phases; (2) a setting where the A-phase is larger than R-phase, while the E-phase is larger than both A and R-phases; (3) a setting where the R-phase is larger than A-phase while the E-phase is larger than both A and R-phases. For $m = 8$ the only tested setting was one where the R-phase is larger than A-phase while the E-phase is larger than both A and R-phases. Combining these settings with the generation parameters used in the generation of each phase's execution time, allows us to obtain in average a ratio of memory to task's execution time ($p = \frac{M_i}{C_i}$) of approximately 43% for those cases where the E-phase is larger than memory phases, and approximately 80% for those cases where the E-phase is smaller than memory phases.

Several experiments were carried out using the above settings in order to find answers for the following questions:

- What is the behaviour of each of the scheduling policies when different configurations of phase lengths of a 3-phase task are used?

[2]Our simulator works with discrete time units. This means that even though the slowdown values are floating point, after computing the results the values will be rounded to the closest integer value. We believe that this is a design decision that mimics the real systems and that does not affect the analysis of resulting data.

- How does the modified global EDF approach compares with the proposed scheduling policies?
- How is modified global EDF affected by interference when the number of cores increases and different slowdown values are applied?
- What is the behaviour of the proposed policies when the number of cores is increased?

Several observations can be made by looking into the following figures: Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6. In the figures, one can see six lines, one for each priority assignment policy (namely, priority (PRIO), minimum Acquisition (MIN.A), maximum Acquisition (MAX.A), minimum Restitution (MIN.R), maximum Restitution (MAX.R) and one for modified global EDF (G-EDF), as detailed in Section 5.4.1).

First Observation: When A and R phases are similar in length the scheduling policies that use the length of A or R phases as a priority criterion (*i.e.,* MIN.A, MIN.R, MAX.A and MAX.R) schedule a similar amount of task sets. This can be seen in Figure 5.1a, Figure 5.1b, Figure 5.3a and Figure 5.3b for two and four cores respectively. Intuitively, as the memory phases have the same length, the outcome is expected to be the same for the policies that use either the minimum phase (MIN.A and MIN.R) or the maximum phase (MAX.A and MAX.R) as selection criterion.

Second Observation: When E-phases are smaller than memory phases as in Figure 5.1a, Figure 5.1b, Figure 5.3a and Figure 5.3b all the proposed heuristics behave poorly because tasks will not benefit from any parallelism provided by the execution of E-phases. In fact, most of the phases will execute in a sequential way in the majority of the time. It can be seen from the figures that even if the number of cores increase (from $m = 2$ to $m = 4$) the percentage of schedulable task sets remains nearly the same.

Third Observation: In all experiments, with exception when both A and R have the same length, the scheduling policies that select tasks based on the minimum length of memory phases (A or R-phases) schedule more task sets than the ones that use the maximum length of memory phases. The reason for this behaviour is likely related to the fact that giving priority to tasks with larger A or R phases reduces the opportunity for parallelism when different tasks execute in parallel. Consequently, no memory phases execute in parallel with E-phases and a higher amount of interference/blocking occurs. This leads to larger response-times in average and consequently, to a higher percentage of unschedulable task sets.

Fourth Observation: As the values of slowdown increase, the number of task sets that are schedulable by G-EDF drastically decreases along with the utilization of the schedulable task sets. Recall that increasing the slowdown means that each traditional task takes longer to execute due to the amount of interference that it suffers when executing in parallel with other tasks.

For $m = 2$ cores, when the slowdown increases from 1.5x to 2x, G-EDF cannot schedule any task set with an utilization greater than 1.5 (as depicted in Figure 5.1a and Figure 5.1b). The same result can be observed for 4 cores, where the percentage of schedulable task sets decreases from 2.1 in the 1.5x slowdown setting to 1.5 in the 2x slowdown setting (as depicted in Figure 5.3a and Figure 5.3b).

When the slowdown is kept constant at 2x and the number of cores is increased from 2 to 4 (see Figure 5.5a and Figure 5.5b) the schedulability ratio for G-EDF remains almost the same.

The above results show that as the number of cores increase, the number of tasks executing in parallel and competing for shared resources also increases. This directly translates into an interference increase which leads to larger execution times than what is estimated in isolation. Depending on the deadline of each task, the effect of interference on task execution may lead to task set unschedulability.

Fifth Observation: For a smaller number of cores ($m = 2$), G-EDF behaves better than any of the proposed priority assignment policies even when considering a slowdown of 1.5x (see for instance Figure 5.2a). Nevertheless, when the number of cores increase, G-EDF behaves worse (as it can be observed in Figure 5.4a). As explained in the previous observation this is caused due to the increase in interference when tasks execute in parallel.

Sixth Observation: Among the proposed policies, the one that uses the period as the priority assignment rule (PRIO in the figures) performs better when compared to the other priority assignment policies (except G-EDF) in terms of schedulable task sets. This can be observed in all figures.

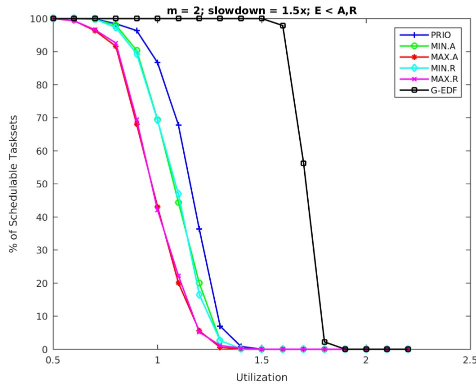
Seventh Observation: By carefully looking at Figure 5.4b and Figure 5.6 one can observe that doubling the number of cores from 4 to 8 leads to approximately the same schedulability ratio for all the policies. For the 3-phase model this is a good indicator of the influence of the memory phases on the schedulability of the system. Having large memory phases limits any advantage that can be obtained by executing E-phases in parallel and this is reflected in this experiment where the memory ratio is kept at around 45%. [3]

This observation is also confirmed by the results obtained by Becker et al. in [Becker et al., 2016]. In their paper, written in parallel to this research work, the authors propose the construction of offline time-triggered schedules, considering the 3-phase task model, using either a linear programming formulation or a heuristic that works very similarly to the runtime behaviour presented in Section 5.4.2. In their experiments, which consider a cluster composed of 14 cores, the authors show that their heuristic reaches a saturation point at 7 cores with an average last schedulable utilization of approximately 2. Any increase in utilization over this value leads to unschedulability, regardless of the number of cores used above 7.

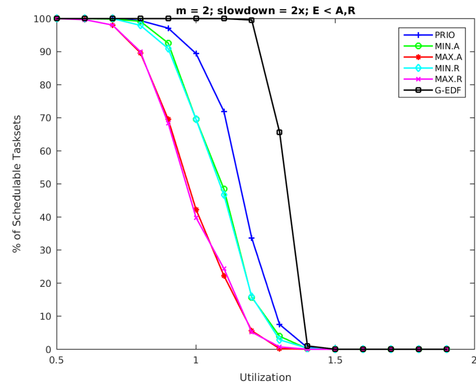
As a concluding remark for this section, all the above results show that the 3-phase task model can be useful in today's COTS multicore architectures in order to avoid task contention due to shared resources. On the positive side, a policy that assigns tasks priorities based on the tasks' periods performs better than an interference-prone version of global EDF. Nevertheless, avoiding interference brings its cost in terms of schedulability and scalability.

[3] Later in this chapter, we explore how the schedulability ratio behaves as a function of the memory utilization per task.

[4] $x \cdot \text{random}(1,3)$ means that a factor x is multiplied by random value generated in the range $[1,3]$

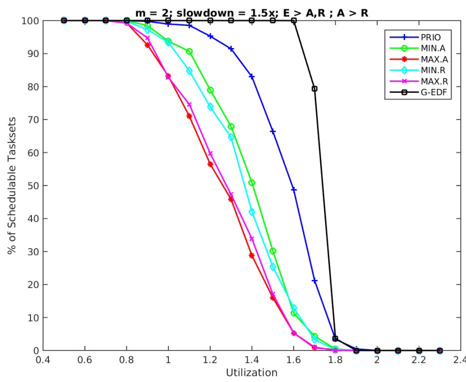


(a) $m = 2$ cores, slowdown values of 1.5x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 1 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$[4]

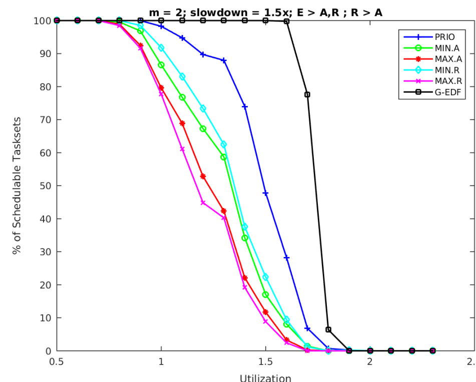


(b) $m = 2$ cores, slowdown = 2x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 1 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

Figure 5.1: Simulation results for $m = 2$, E-phase smaller than A and R-phases

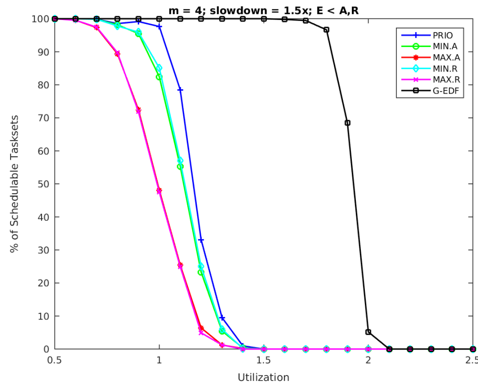


(a) $m = 2$ cores, slowdown = 1.5x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 1 \cdot \text{random}(1,3)$

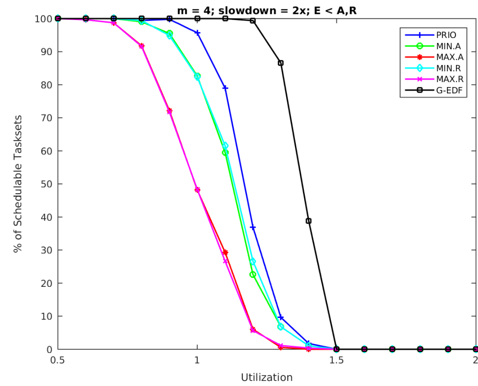


(b) $m = 2$ cores, slowdown = 1.5x. Task parameters: $A = 1 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

Figure 5.2: Simulation results for $m = 2$, E-phase larger than both A and R-phases

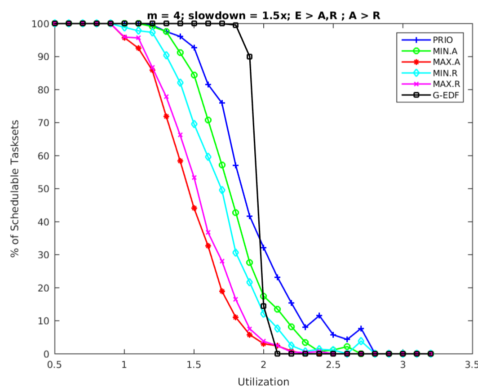


(a) $m = 4$ cores, slowdown = 1.5x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 1 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

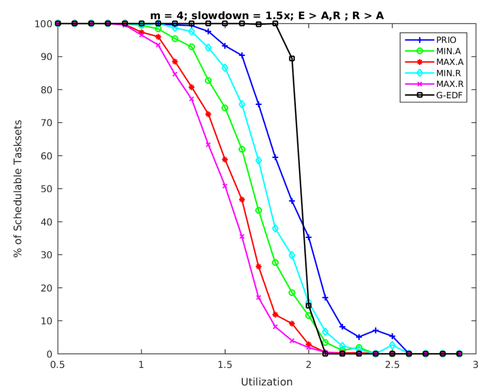


(b) $m = 4$ cores, slowdown = 2x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 1 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

Figure 5.3: Simulation results for $m = 4$, E-phase smaller than A and R-phases

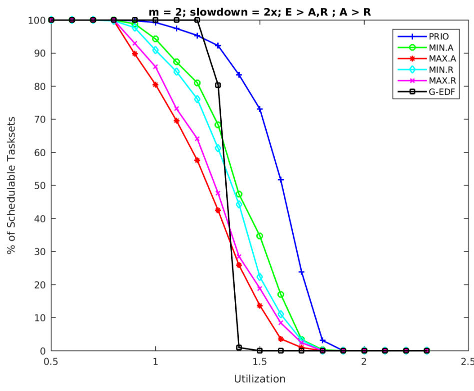


(a) $m = 4$ cores, slowdown = 1.5x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 1 \cdot \text{random}(1,3)$

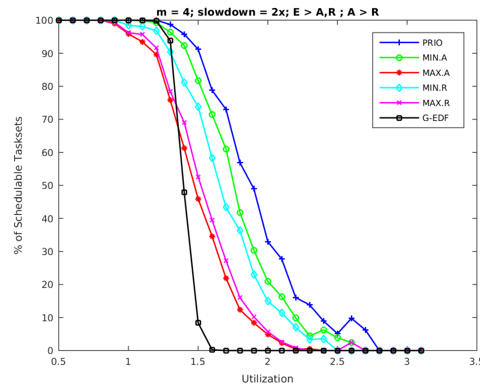


(b) $m = 4$ cores, slowdown = 1.5x. Task parameters: $A = 1 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

Figure 5.4: Simulation results for $m = 4$, E-phase larger than both A and R-phases



(a) $m=2$ cores, slowdown = 2x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 1 \cdot \text{random}(1,3)$



(b) $m=4$ cores, slowdown = 2x. Task parameters: $A = 2 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 1 \cdot \text{random}(1,3)$

Figure 5.5: Comparison between $m=2$ and $m=4$, E-phase larger than both A and R-phases

5.5 Global Fixed-Priority Scheduling of the 3-Phase Task Model

In this section we present a new schedulability test for the global fixed-priority scheduling of the 3-phase task model. This work differs from current state of the art, *i.e.,* [Alhammad and Pellizzoni, 2014]), by analysing the schedulability of the system from a bus perspective instead of a core's perspective and fill in some gaps left open by their analysis (for instance, their work neglects some of the interference generated by R-phases).

While in Section 5.4 we experimented with several priority assignment heuristics, in this section we assume that each task τ_i in τ has a fixed priority. Moreover, we denote the set of tasks with higher or equal priority than τ_i (including τ_i) by $hep(i)$, and we use $lp(i)$ to denote the set of tasks with lower priority than τ_i.

5.5.1 Scheduling Policy

The scheduling policy used to derive the schedulability test for the 3-phase task model is the following.

Jobs released by tasks are executed on cores in a non-preemptive global fixed-priority manner. Once assigned to a core, a job starts the execution of its A-phase, followed by its E-phase and finally its R-phase in a non-preemptive manner. Thus, at any given time instant there are at most m uncompleted jobs that have started their execution.

Even though execution is non-preemptive, a job might have to wait between its E and R-phase to gain access to the bus (remember that the bus is locked by cores to ensure exclusive access to the main memory during an A or R-phase). If a job J must start its R-phase and the bus is already busy serving another memory phase of a job executing on another core, J spin-locks (non-preemptively) waiting for the bus to be freed.

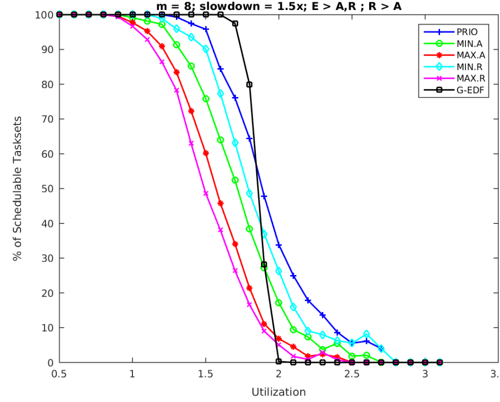


Figure 5.6: Simulation results for $m = 8$ cores, slowdown = 1.5x. Task parameters: $A = 1 \cdot \text{random}(1,3), E = 4 \cdot \text{random}(1,3), R = 2 \cdot \text{random}(1,3)$

We assume that A-phases have always higher priority than R-phases to access the bus. We further assume that R-phases execute in a FIFO order. Serving R-phases in a FIFO order ensures progress. A low priority task cannot be blocked (spin-locking) indefinitely by higher priority tasks running on other cores. However, this means that more than one lower priority task can block higher priority ones during their restitution phase.

The scheduler is event-driven. It is invoked whenever one of the following events happens: (1) a job release; (2) the completion of a A, E or R-phase.

The scheduler uses two different queues to keep track of ready phases. The phases pushed in the first queue (henceforth called PriorityQueue) are sorted in a non-increasing priority order. The phases pushed in the second queue (referred to as FIFOQueue) are ordered following a first-in first-out (FIFO) ordering policy. Following the idea described above, ready A-phases are always pushed into the PriorityQueue, while ready R-phases are pushed in the FIFOQueue. Hence, at a job release, the A-phase of the released job is enqueued into the PriorityQueue. Similarly, when the E-phase of a job completes, the R-phase of that job is inserted into the FIFOQueue.

Algorithm 1 provides a pseudo-code of the scheduling algorithm executed at each scheduler invocation. It first checks if the bus is available. If it is not, then it simply exits and waits for the active memory phase to complete its execution. If the bus is free then the scheduler checks if at least one of the two queues contains ready memory phases. Since, following our assumption, A-phases have higher priority than R-phases, the scheduler checks first if there is an A-phase waiting in the PriorityQueue.

If an A-phase is ready, the scheduler then checks if there is an idle core π_k. If it is the case, the job to which the A-phase belongs to is assigned to π_k, the bus is locked and the DMA is configured to start the A-phase on the bus. Otherwise, if no core is available then the A-phase must wait until another job completes its execution and releases a core. If no A-phase can be started, either because the priority queue is empty or no core is free, the FIFOQueue needs to be checked for ready R-phases so that any job that still has a pending R-phase can be completed and the core be freed to execute other jobs.

Algorithm 1 Scheduling algorithm pseudo-code

1: **if** Bus is Free **then**
2:     **if** PriorityQueue not Empty **then**
3:         **if** Free Core Available **then**
4:             Pull the the task τ_i with the highest priority A-phase from the PriorityQueue;
5:             Assign τ_i to one of the free cores;
6:             Lock the bus and start the A-phase of τ_i;
7:             **return**;
8:         **end if**
9:     **end if**
10:     **if** FIFOQueue not Empty **then**
11:         Take the first R-phase from the FIFOQueue;
12:         Lock the bus and start the R-phase;
13:     **end if**
14: **end if**

When an A-phase completes its execution on a core π_k, the bus is unlocked and the E-phase of the respective job immediately starts its execution on core π_k. Hence, there is no idle time between the completion of an A-phase and the execution of its corresponding E-phase. Further, there is no migration from one core to any other core between phases of a same job. Finally, at the completion of a job E-phase, the R-phase of this job is enqueued into the FIFOQueue. Upon completion, an R-phase releases both the bus and the core on which it executed.

5.5.2 Background

Alhammad and Pellizzoni [Alhammad and Pellizzoni, 2014] provide a schedulability test for the 3-phase task model based on the technique proposed in [Baker, 2003] and [Guan et al., 2008] for global non-preemptive scheduling on multiprocessor systems.

The schedulability test consists in analysing a time interval $[t_0, d_k - (A_k + E_k)]$ of a job of τ_k which is assumed to miss its deadline at time d_k. That job is called the *problem job*. The time instant t_0 is the latest time instant earlier than the release r_k of τ_k's problem job at which at least one processor is idle. The interval $[t_0, d_k - (A_k + E_k)]$ is called *problem window* and is depicted in Figure 5.7.

Intuitively, if the scheduler is work-conserving, for the problem job to miss its deadline, the amount of interference occurring in the problem window must be greater than the computing

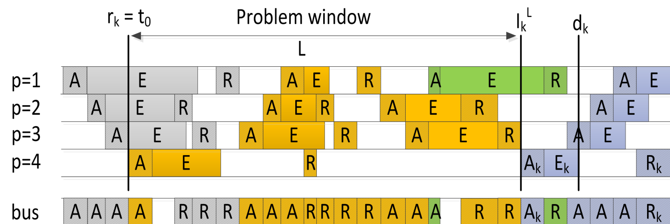


Figure 5.7: Problem Window

supply[5]. Following this observation, computing an upper-bound on the interference suffered by each task and comparing it against a lower bound on the supply available in the problem window allows us to determine whether the system is schedulable or not. In particular, if the maximum interference each task can suffer is less than the minimum supply in their problem window then the system is schedulable.

In [Alhammad and Pellizzoni, 2014], the authors apply the above analysis technique to the 3-phase task model and show that the problem window must be an interval during which either (i) the bus is busy executing memory phases, or (ii) all processors are busy executing E-phases.

Similarly to [Baker, 2003], and to the model used in Chapter 3, the worst-case workload of each higher-priority task within the problem window (and hence the interference generated by each higher priority task on the problem job) is divided into three parts:

1. Carry-in workload: The carry-in workload is composed of jobs (henceforth called carry-in jobs) released before t_0 and with their deadline after t_0. These jobs are represented in gray in Figure 5.7.

2. Body jobs: Body jobs have their release and deadline entirely contained within the problem window. These jobs are represented in yellow in Figure 5.7.

3. Carry-out workload: The carry-out workload is composed of jobs (henceforth called carry-out jobs) released within the problem window but with their deadline outside of the problem window. These jobs are represented in green in Figure 5.7.

The contribution of each of these jobs to the interference suffered by the problem job is analysed in detail in the next sections.

5.5.2.1 Carry-in Workload

Concerning the carry-in jobs, it was proven in [Alhammad and Pellizzoni, 2014] that at most m tasks can have a carry-in job among which at most $(m-1)$ are from higher or equal priority tasks. Furthermore, since we assume a constrained-deadline task model, each task can have at most one carry-in job. It was further proven that the worst-case interference happens when $(m-1)$ cores are busy executing E-phases from carry-in jobs while a lower priority task blocks the execution of τ_k at t_0. This result is formally stated in Theorem 4 below.

Theorem 4. *(from [Alhammad and Pellizzoni, 2014]) In the worst-case, the carry-in workload at time t_0 is limited by $m-1$ computation phases (E-phases) from tasks in $hep(k) \cup lp(k)$ and one full job from a task in $lp(k)$.*

5.5.2.2 Alhammad's Schedulability Analysis

To compute the interfering workload of the body and carry-out jobs on the problem job, the approach in [Alhammad and Pellizzoni, 2014] focuses on what happens on the cores. Specifically,

[5] The supply in a time interval is the total amount of computation that could be performed within the interval.

within a problem window of length L_k, the interfering workload must consider the contribution of $\lfloor \frac{L_k}{T_i} \rfloor$ body jobs and at most one carry-out job of each higher priority task $\tau_i \in hp(k)$.

Due to the restriction that no two memory phases can execute simultaneously on the bus, the schedule contains *scheduling holes* (see the grey blocks in Figure 5.9). A scheduling hole is an interval of time in which a core is idle as a result of a memory phase being executed by another core.

Therefore, to bound the interference suffered by a job in its problem window, one must also upper bound the cumulative length of the holes on the cores. Alhammad and Pellizzoni do it by lower bounding the time during which the execution of E-phases on cores *overlap* with memory phases executed on the bus. The total length of the holes is then given by $(m \times \sum_{i=1}^{\alpha} \mu_i - \text{overlap})$ where $\sum_{i=1}^{\alpha} \mu_i$ is the sum of all memory phases executed in the problem window.

In order to compute a lower-bound on the amount of overlap (equivalently, an upper bound on the length of holes), the authors in [Alhammad and Pellizzoni, 2014] propose the following approach. First, the largest A-phases are combined with the largest R-phases into single memory phases, *i.e.,* $M^i = A^i + R^i$, with $A^i \geq A^{i+1}$ and $R^i \geq R^{i+1}$. Each element M^i is added to a sequence μ sorted in a non-increasing order. The E-phases are sorted in a sequence λ in a non-decreasing order.

Let α be the size of μ and λ, $\rho \leq \frac{\alpha}{m}$ partitions are created as depicted in Figure 5.8. The ρ largest memory phases in the sequence μ and the ρ smallest computation phases in λ are assigned to the first core. The second ρ largest memory phases in the sequence μ and the ρ smallest computation phases in λ are assigned to the second core. This procedure is repeated on each core. Thus, by following this assignment, the largest memory phases overlap with the smallest computation phases leading to a lower-bound on the amount of overlap between the phases. The length of the holes in each partition k is then upper-bounded by the length of the grey blocks in Figure 5.8.

Equation 5.1 summarizes the approach in [Alhammad and Pellizzoni, 2014]. The workload interfering with τ_k within the problem window L_k is given by the sum of the α memory phases plus the sum of all E-phases executing in the interval minus a lower bound on the overlap of the E-phases which can be computed following the procedure described above. For more details please check section 4.3 in their paper.

$$W_k^{NC} = m \cdot \sum_{i=1}^{\alpha} M^i + \sum_{i=1}^{\alpha} \lambda^i - \text{overlap} \tag{5.1}$$

5.5.2.3 Limitations

While the approach proposed in [Alhammad and Pellizzoni, 2014] is interesting from an analysis viewpoint, we note two main limitations:

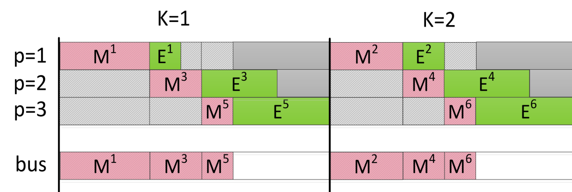


Figure 5.8: Computing the overlap lower-bound for $\rho = 2, m = 3$ in [Alhammad and Pellizzoni, 2014]

- First, it considers that a task is schedulable if it completes its E-phase by its deadline, as depicted in Figure 5.7. Therefore, it omits the time required for the task to execute its R-phase. Since the R-phase is in charge of writing the results of the task computation back to main memory, this can be problematic if tasks have precedence constraints or any form of data dependencies.

- Second, it can be very pessimistic. Specifically, by only looking at the overlap that occurs in each partition, the analysis misses the overlap that exists across partitions. Comparing Figure 5.8 and Figure 5.9, one can see an example of this pessimism. By allowing the memory phases of the second partition to start as soon as possible (as in Figure 5.9) one can decrease the amount of interference by more than E_5 time units in comparison to Figure 5.8 (which is the execution scenario assumed in [Alhammad and Pellizzoni, 2014]).

5.5.3 A Different Perspective

We look at the problem of the 3-phase task model's inter-task interference from a different perspective. While Alhammad and Pellizzoni [Alhammad and Pellizzoni, 2014] analyse the schedulability of each task by modelling the scheduling behaviour on the cores, we consider what occurs on the bus. Analysing the bus instead of the cores reduces the schedulability problem to a single core problem (there is only one bus which executes at most one memory phase at a time) instead of a multicore problem.

Yet, similarly to the fact that scheduling holes can appear on the cores when a memory phase is being processed on the bus, *bus holes* can be observed on the bus whenever all the cores are busy executing E-phases (see Figure 5.10). Bus holes happen because, when all the cores are busy executing E-phases, none of the local memories can accept new content nor can the computation

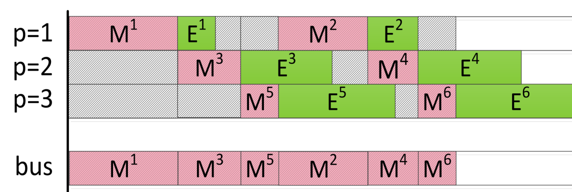


Figure 5.9: Pessimism of the analysis in [Alhammad and Pellizzoni, 2014]

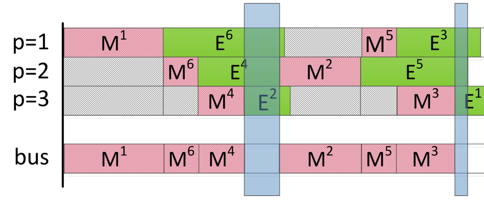


Figure 5.10: Our schedulability analysis approach

result be written back to main memory. Hence, the bus remains idle. Formally, a bus hole is defined as follows.

Definition 8 (Bus Hole)**.** *A bus hole is an interval of time, within the problem window, where all m cores are busy executing E-phases.*

Our analysis builds upon the observation that within the problem window, the contribution of the jobs to the response time of the task under analysis is divided into two parts (see Figure 5.10): (1) the interference of the memory phases (A and R) that execute within the window, and (2) the cumulative length of time during which all cores execute E-phases (if any such interval exists). We denote this latter length by L_i^{holes}.

Upper bounding the length L_i^{holes} and adding its value to the total time required to process memory phases of body, carry-in and carry-out jobs executed in the problem window results in an upper bound on the interference that the task under analysis may suffer in the worst-case. Thus, the worst-case interference that a task τ_i can suffer in an interval of length t is bounded by

$$I_i(t) = L_i^{holes}(t) + I_i^{bus}(t) \tag{5.2}$$

where $L_i^{holes}(t)$ is the maximum cumulative time m different E-phases are simultaneously executing on the m cores in an interval of length t, and $I_i^{bus}(t)$ is an upper bound on the interference τ_i can suffer on the bus due to the execution of memory phases of other jobs during an interval of length t.

To compute the exact length of $L_i^{holes}(t)$ one has to know how the jobs of each task are scheduled on the cores within the problem window. Checking all potential jobs' schedules to find the schedule generating the longest cumulative length $L_i^{holes}(t)$ is intractable. Therefore, we propose a pseudo-polynomial technique to compute an upper bound on $L_i^{holes}(t)$.

To summarize, our proposed technique differs from the technique in [Alhammad and Pellizzoni, 2014] in the sense that we upper-bound the interference on the bus instead of upper-bounding the interference on the cores. To achieve this, we must compute an upper-bound on the length of the so-called bus holes. By definition of bus holes, if one can maximize the length of the intervals where all cores are simultaneously busy executing E-phases, then an upper-bound to the length $L_i^{holes}(t)$ is found.

5.5.4 Schedulability Analysis

As already explained in Section 5.5.2, a task τ_i is schedulable if $I_i(t) \leq t$ where t is a lower-bound on the length of the problem window of τ_i and $I_i(t)$ is the maximum interference suffered by τ_i in that window.

An upper-bound on $I_i(t)$ can be computed using Equation 5.2, where the term $L_i^{holes}(t)$ accounts for the interference suffered by τ_i due to the execution of E-phases while $I_i^{bus}(t)$ considers the interference caused by memory phases executed in an interval of length t. Before computing $L_i^{holes}(t)$ and $I_i^{bus}(t)$, one should know the length t of the problem window on which $L_i^{holes}(t)$ and $I_i^{bus}(t)$ must be computed. In [Alhammad and Pellizzoni, 2014], the authors use $t = D_i - E_i - M_i$ (where $M_i = A_i + R^{\max}$, with $R^{\max}$ being the largest R-phase executed in the problem window) as that length. However, as already pointed out in Section 5.5.2.2, one of the main limitations of the analysis presented in [Alhammad and Pellizzoni, 2014] is that it does not consider the time required by the R-phase of τ_i to write its data back in main memory. Therefore, in this section, we first prove an upper-bound on the time needed for τ_i to complete its R-phase. Then, we use that information to derive a bound on the length t that must be considered in the schedulability test of any task τ_i. Finally, in Sections 5.5.4.2 and 5.5.4.3, we prove upper-bounds on $I_i^{bus}(t)$ and $L_i^{holes}(t)$, respectively.

5.5.4.1 R-phase Worst-Case Response Time and Problem Window Length

Let $\overrightarrow{A}_i$ and $\overrightarrow{R}_i$ be the set of A and R-phases of the tasks in $\tau \setminus \tau_i$ sorted in a non-increasing order (where τ_i is the task under analysis). We denote by $\overrightarrow{A}_i^{(k)}$ (resp., $\overrightarrow{R}_i^{(k)}$), the kth element in $\overrightarrow{A}_i$ (resp., $\overrightarrow{R}_i$). Therefore, $\overrightarrow{A}_i^{(k)}$ is the kth largest A-phase among those executed by tasks in $\tau \setminus \tau_i$.

Lemma 6. *The interference suffered by the R-phase of τ_i is upper-bounded by:*

$$I_i^R = \sum_{k=1}^{m-1} \left(\overrightarrow{A}_i^{(k)} + \overrightarrow{R}_i^{(k)} \right) \tag{5.3}$$

Proof. R-phases are inserted in a FIFO queue. Therefore, the worst-case for the task under analysis τ_i occurs when $(m-1)$ other jobs inserted their R-phases in the queue before τ_i's. Hence, τ_i has to wait until all those other R-phases complete before τ_i's R-phase can start. Further, because tasks have constrained deadlines, each task has at most one active job and hence one active R-phase at any time (assuming the system is schedulable). Therefore, the (at most) $(m-1)$ R-phases interfering with τ_i's R-phase are from different tasks. The contribution of R-phases to the interference of τ_i is thus upper-bounded by the sum of the $(m-1)$ largest R-phases in the system, *i.e.*, by $\sum_{k=1}^{m-1} \overrightarrow{R}_i^{(k)}$.

Furthermore, for each completed R-phase, a core is freed and Algorithm 1 is called. Since A-phases have higher priority than R-phases, the transmission of τ_i's R-phase can be delayed by the transmission of a ready A-phase. Note however that a maximum of $(m-1)$ cores can be freed before the transmission of τ_i's R-phase, and therefore, by Line 3 of Algorithm 1, at most $(m-1)$ A-phases can interfere with τ_i's R-phase. Similarly to the discussion for R-phases, because each

task can have at most one A-phase ready at any time, the $(m-1)$ A-phases interfering with τ_i's R-phase must be from different tasks. The contribution of A-phases to the response-time of τ_i's R-phase is thus upper-bounded by the sum of the $(m-1)$ largest A-phases in the system, *i.e.,* by $\sum_{k=1}^{m-1} \overrightarrow{A}_i^{(k)}$.

Adding both contributions, we get that the interference suffered by τ_i's R-phase is upper-bounded by $\sum_{k=1}^{m-1} \left(\overrightarrow{A}_i^{(k)} + \overrightarrow{R}_i^{(k)} \right)$. □

Corollary 1. *The response time of the R-phase of τ_i is upper bounded by $R_i + I_i^R$.*

Proof. Directly follows from Lemma 6. □

Now that we have an upper bound on the response time of τ_i's R-phase, we derive a bound on the length t of the problem window.

Lemma 7. *If the problem job of τ_i misses its deadline, then $I_i(t) \geq t$ where $t = D_i - A_i - E_i - R_i - I_i^R + \varepsilon$ and ε is an arbitrary small number.*

Proof. Let us assume that the problem job of τ_i is released at time r_i and has its deadline at time $r_i + D_i$. We prove the claim by contradiction. Let us assume that $I_i(t) < t$. Since $I_i(t)$ sums all the instants where the bus is busy executing memory phases or all cores are busy executing E-phases (see Equation 5.2), then, by our contradictory assumption, there must exist an instant t_{idle} such that $r_i \leq t_{idle} < r_i + t$ at which both the bus is idle and at least one core is idle.

By Algorithm 1, the A-phase of τ_i's problem job can start executing on the bus at t_{idle}. Since A and E-phases execute non-preemptively, they complete their execution by $t_{idle} + A_i + E_i$. Furthermore, since the response time of τ_i's R-phase is upper-bounded by $R_i + I_i^R$ (Corollary 1), the R-phase of τ_i's problem job completes by $t_{idle} + A_i + E_i + R_i + I_i^R$. Replacing t_{idle} by its upper-bound, we get $t_{idle} + A_i + E_i + R_i + I_i^R < r_i + D_i - A_i - E_i - R_i - I_i^R + \varepsilon + A_i + E_i + R_i + I_i^R = r_i + D_i + \varepsilon$. Since ε is an arbitrarily small number, the R-phase of τ_i's problem job therefore completes at or before $r_i + D_i$. It is a contradiction with the assumption that the problem job of τ_i misses its deadline, hence the claim. □

Theorem 5. *If for all $\tau_i \in \tau$, $I_i(t) < t$ where $t = D_i - A_i - E_i - R_i - I_i^R + \varepsilon$ and ε is an arbitrary small number, then the system is schedulable.*

Proof. It is the contra-positive of Lemma 7. If $I_i(t) < t$ for any task τ_i, then every job of τ_i meets its deadline. It follows that if the condition is true for all tasks then all jobs meet their deadlines and the system is schedulable. □

5.5.4.2 Upper-bound on $I_i^{bus}(t)$

In this section, we derive an upper-bound on $I_i^{bus}(t)$.

Let $\mathfrak{J}(t)$ be the largest set of jobs that can execute (completely or partially) in an interval of length t and prevent τ_i's A-phase to start executing. We divide the set $\mathfrak{J}(t)$ in two different subsets composed of (i) carry-in jobs, and (ii) body and carry-out jobs, respectively.

With respect to (i), Theorem 4 tells us that the carry-in workload is upper-bounded by $(m-1)$ computation phases (E-phases) from tasks in $hep(k) \cup lp(k)$ and one full job from a task in $lp(k)$. Since every E-phase is followed by an R-phase, and because every full job has both an A and an R-phase, the contribution of the carry-in workload to $I_i^{bus}(t)$ is upper-bounded by

$$A_{low}^{max} + \sum_{k=1}^{m} \overrightarrow{R}^{(k)}$$

where A_{low}^{max} is the largest A-phase among the tasks with lower priority than τ_i and $\overrightarrow{R}^{(k)}$ is the kth largest R-phase among all tasks in τ.

Regarding the number of body and carry-out jobs in $\mathcal{J}(t)$, two cases must be considered:

- $\tau_k \in hp(i)$. The maximum number of jobs of τ_k that can be released and have their deadline within an interval of length t is upper bounded by $\left\lfloor \frac{t}{T_k} \right\rfloor$. The contribution of body jobs of τ_k to $I_i^{bus}(t)$ is thus upper-bounded by $\left\lfloor \frac{t}{T_k} \right\rfloor (A_k + R_k)$. Further, the contribution of τ_k to the carry-out workload is limited to one job of size at most $\min\{(A_k + R_k), (t \bmod T_k)\}$ (*i.e.,* since we have a constrained-deadline model, each task can have at most one carry-out job, that is released no earlier than $(t \bmod T_k)$ time units before the end of the problem window, and the carry-out job cannot execute for more than $(t \bmod T_k)$ in $(t \bmod T_k)$ time units).
- $\tau_k \in lep(i)$. If τ_k's priority is lower than or equal to the priority of τ_i, then, thanks to Line 4 of Algorithm 1, no job released by τ_k after or at the same time than a job of τ_i can interfere with τ_i. Therefore, no body or carry-out job of τ_k participates to $I_i^{bus}(t)$.

Finally, adding the contribution of all jobs in $\mathcal{J}(t)$ together, we get that

$$\begin{aligned} I_i^{bus}(t) \leq A_{low}^{max} + \sum_{k=1}^{m} \overrightarrow{R}^{(k)} + \\ \sum_{\tau_k \in hp(i)} \left(\left\lfloor \frac{t}{T_k} \right\rfloor (A_k + R_k) + \min\{A_k + R_k,\ t \bmod T_k\} \right) \end{aligned} \quad (5.4)$$

5.5.4.3 Upper-bound on $L_i^{holes}(t)$

The length $L_i^{holes}(t)$ provides an upper bound on the total time during which all cores are busy executing E-phases in a window of length t. Hence, $L_i^{holes}(t)$ depends on the jobs' schedule in that window. Finding the worst-case schedule that provides the largest length $L_i^{holes}(t)$ is intractable in the general case. Therefore, we provide an over-approximation of that length by building an artificial schedule of memory and execution phases that is at least as bad as the worst-case schedule.

Intuitively, the length $L_i^{holes}(t)$ is maximized by considering an artificial schedule as follows. Let us assume that k successive E-phases execute on the first core, and ℓ memory phases execute on all other cores, in parallel with those E-phases. Since there is only one memory bus, at most one memory phase can be processed at a time. It results that the ℓ memory phases are executed sequentially. The length of the time interval $L_i^{holes}(t)$ during which *all cores* are busy executing

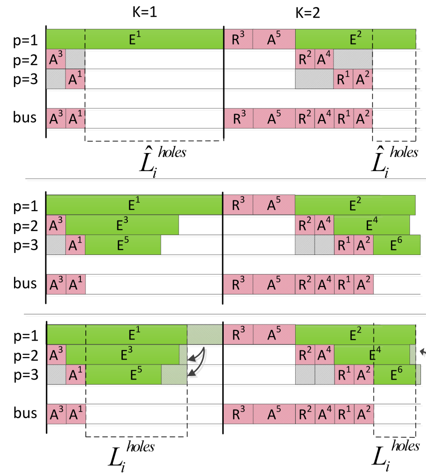


Figure 5.11: Computing an upper-bound on bus holes

E-phases is therefore upper-bounded by the sum of the length of the k E-phases running on the first core, minus the lengths of the ℓ memory phases executed on all the other cores. This length is further maximized if the k E-phases executing on core 1 are the k longest, and the memory phases executed on the other cores are the ℓ shortest. This intuition can be observed in the upper part of Figure 5.11. In the figure, one can observe the longest k E-phases in green running on core 1, $(m-1)$ A- and R-phases running in parallel with each execution phase (in pink), and an upper bound $\hat{L}_i^{holes}(t)$ on $L_i^{holes}(t)$ represented as the difference between the length of the execution and memory phases. Formally, we have in the general case that

$$L_i^{holes}(t) \leq \hat{L}_i^{holes}(t) = \sum_{j=1}^{k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{j=1}^{p} \overleftarrow{\mathcal{A}}^{(j)} - \sum_{j=1}^{q} \overleftarrow{\mathcal{R}}^{(j)} \tag{5.5}$$

where $\overrightarrow{\mathcal{E}}$, $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{R}}$ are, respectively, the set of all E, A and R-phases interfering with τ_i's execution. $\overrightarrow{\mathcal{E}}^{(j)}$ denotes the jth element of the set sorted in a *non-increasing* order, while $\overleftarrow{\mathcal{A}}^{(j)}$ is the jth element of the set sorted in a *non-decreasing* order (note the direction of the arrow on top of the set). Therefore, Equation 5.5 accounts for the k longest E-phases interfering with τ_i and the p and q shortest A and R-phases, respectively (with $\ell = p + q$ in the explanation above).

Even though Equation 5.5 provides an upper bound on $L_i^{holes}(t)$, it is extremely pessimistic due to the fact that it neglects how different E-phases execute in parallel on different cores. Only the E-phases (conservatively assumed to be the k longest ones) running on the first core are considered when computing the bound.

A tighter bound on $L_i^{holes}(t)$ can be obtained by considering the E-phases scheduled on all cores. Assume that each core (and not the first one only) executes the same number k of E-phases, then each E-phase is preceded by an A-phase, and each A-phase is preceded by the R-phase of the previous jobs that executed on the same core (see the middle part of Figure 5.11). Therefore, k A-phases and at least $(k-1)$ R-phases execute on each core. [6] Now, let us build an artificial schedule

[6]Without loss of generality, if $|\overrightarrow{\mathcal{E}}| < (k \times m)$, then the set $\overrightarrow{\mathcal{E}}$ is appended with zero-length E-phases such that

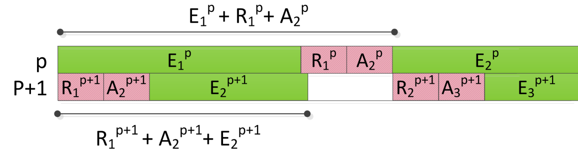


Figure 5.12: Bus holes

as shown on the middle part of Figure 5.11 where the k longest E-phases execute on the first core, the k second longest E-phases execute on the second core, and so on and so forth. Similarly, the k shortest A-phases and the $(k-1)$ shortest R-phases execute on the mth core, the k second shortest A-phases and the $(k-1)$ second shortest R-phases execute on the $(m-1)^{\text{th}}$ core, *etc.* Then, the amount of time the E-phases on the first core do not overlap with memory phases and hence can participate to $L_i^{holes}(t)$ is given by Equation 5.5 with $p = (m-1)\times k$ and $q = (m-1)\times(k-1)$ (*i.e.,* the number of A and R-phases, respectively, executing on the other cores). That is, it is given by

$$\sum_{j=1}^{k} \overrightarrow{\mathcal{E}}(j) - \sum_{j=1}^{(m-1)\times k} \overleftarrow{\mathcal{A}}(j) - \sum_{j=1}^{(m-1)\times(k-1)} \overleftarrow{\mathcal{R}}(j)$$

Similarly, the amount of time the E-phases on the second core do not overlap with memory phases and hence can participate to $L_i^{holes}(t)$ is given by

$$\sum_{j=k+1}^{2\times k} \overrightarrow{\mathcal{E}}(j) - \sum_{j=1}^{(m-2)\times k} \overleftarrow{\mathcal{A}}(j) - \sum_{j=1}^{(m-2)\times(k-1)} \overleftarrow{\mathcal{R}}(j)$$

where $(m-2)\times k$ and $(m-2)\times(k-1)$ are the number of A and R-phases executing on cores 3 to m (see middle part of Figure 5.11). Doing the same for each core and summing all those contributions, we get that the total time during which E-phases do not overlap with memory phases is upper bounded by

$$\sum_{j=1}^{m\times k} \overrightarrow{\mathcal{E}}(j) - \sum_{p=0}^{(m-2)} (m-1-p)\times \Big(\sum_{j=1}^{k} \overleftarrow{\mathcal{A}}(j+p\times k) + \sum_{j=1}^{k-1} \overleftarrow{\mathcal{R}}(j+p\times(k-1)) \Big) \tag{5.6}$$

The maximum amount of time the m cores are all simultaneously busy executing E-phases is thus upper-bounded by the above equation divided by m (see lower part of Figure 5.11). This gives us an upper-bound on $L_i^{holes}(t)$ as formalised in Theorem 6.

$|\overrightarrow{\mathcal{E}}| = k\times m$. Similarly, zero-length A-phases and zero-length R-phases are appended to sets $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{R}}$, respectively, until their cardinality equals $k\times m$.

Theorem 6. *An upper bound on $L_i^{holes}(t)$ is given by*

$$L_i^{holes}(t) \leq \frac{1}{m} \times \max_{k \geq 1} \left\{ \sum_{j=1}^{m \times k} \overrightarrow{\mathcal{E}}^{(j)} - \sum_{p=0}^{(m-2)} (m-1-p) \times \left(\sum_{j=1}^{k} \overleftarrow{\mathcal{A}}^{(j+p \times k)} + \sum_{j=1}^{k-1} \overleftarrow{\mathcal{R}}^{(j+p \times (k-1))} \right) \right\} \tag{5.7}$$

Proof. As explained above, Equation 5.6 provides an upper-bound on $L_i^{holes}(t)$ assuming that: (i) only the longest E-phases and the shortest A and R-phases are running, (ii) core p executes E, A and R-phases that are no smaller than those executed on core $(p+1)$ for all $p \in [1, m-1]$, and (iii) at least k A-phases, k E-phases and $(k-1)$ R-phases are executed on each core. An upper-bound on $L_i^{holes}(t)$ is thus found when Equation 5.6 is maximized over k, which gives us Equation 5.7. However, we still have to prove that the three assumptions hold.

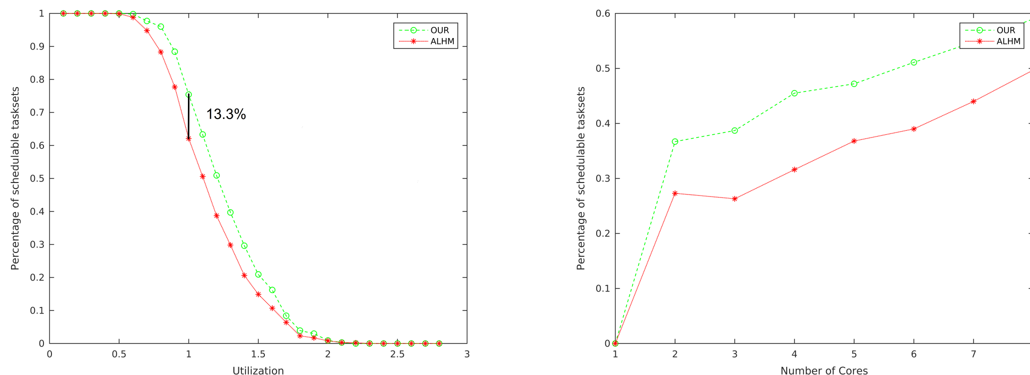
Assumption (i) is quite obvious. If shorter E-phases are executed then the amount of time all cores simultaneously execute E-phases cannot increase. Similarly, if longer memory phases execute, then their overlap with E-phases can only increase, hence reducing the cumulative time all cores execute E-phases simultaneously.

Regarding Assumption (ii), in the schedule seen on the middle part of Figure 5.11, one can see that if memory phases were swapped between cores (*e.g.,* swapping A^5 and A^2), then the time during which memory phases would overlap with E-phases would increase and hence the length of bus holes would decrease. The shortest memory phases must therefore execute on the cores with the largest indexes. Similarly, if E-phases are swapped between cores (*e.g.,* E^2 and E^6 in Figure 5.11), then the amount of time all cores execute E-phases in parallel can only decrease. $L_i^{holes}(t)$ is thus maximized when the largest E-phases execute on the cores of the lowest indexes.

Finally, we prove Assumption (iii). That is, if core 1 executes k A-phases and k E-phases, and if core p executes E, A and R-phases that are no smaller than those executed on core $(p+1)$ for all $p \in [1, m-1]$, then at least k A-phases, k E-phases and $k-1$ R-phases are executed on each core.

The claim obviously holds for core 1 since each E-phase is followed by an R-phase. Hence, at least $k-1$ R-phases execute along with the k A- and E-phases on core 1. The proof for the other cores is by induction. That is, we prove that if core p executes at least k A-phases, k E-phases and $k-1$ R-phases, then core $p+1$ executes at least k A-phases, k E-phases and $k-1$ R-phases.

Let phases E_1^p, R_1^p, A_2^p and E_2^p be executed in a sequence on core p and similarly phases R_1^{p+1}, A_2^{p+1} and E_2^{p+1} be successively executed on core $p+1$ (see Figure 5.12). Since all the E, A and R-phases executing on core $p+1$ are shorter than those executing on core p, we have that $R_1^{p+1} + A_2^{p+1} + E_2^{p+1} \leq E_1^p + R_1^p + A_2^p$. Therefore, as illustrated on Figure 5.12, the E-phase E_2^{p+1} executed on core $p+1$ must complete before the second E-phase E_2^p starts executing on core p. Thus, there are at least as many E-phases executing on core $p+1$ than on core p. Since each E-phase is preceded by an A-phase and followed by an R-phase the number of A and R-phases on core $p+1$ also matches the number of phases on core p. This proves our claim. □



(a) % of schedulable task sets per utilization for $m =$ 4 cores

(b) % of schedulable task sets as a function of the number of cores.

Figure 5.13: Schedulability ratio for $m = 4$ and as a function of the number of cores

5.5.5 Experimental Results

We compare the approach presented in Section 5.5 against the approach presented in [Alhammad, 2016; Alhammad and Pellizzoni, 2014] using randomly synthetically generated task sets. The generation parameters are detailed next.

The number of tasks per task set is set to $n = 5 \times m$ and the total utilization of each task set U_τ ranges from $[0.025 \times m, 0.7 \times m]$ in steps of $0.025 \times m$. UUnifast-Discard [Davis and Burns, 2011b] is used to generate n utilization values such that $U_i \leq 1$ and $\sum_{i=1}^{n} U_i = U_\tau$.

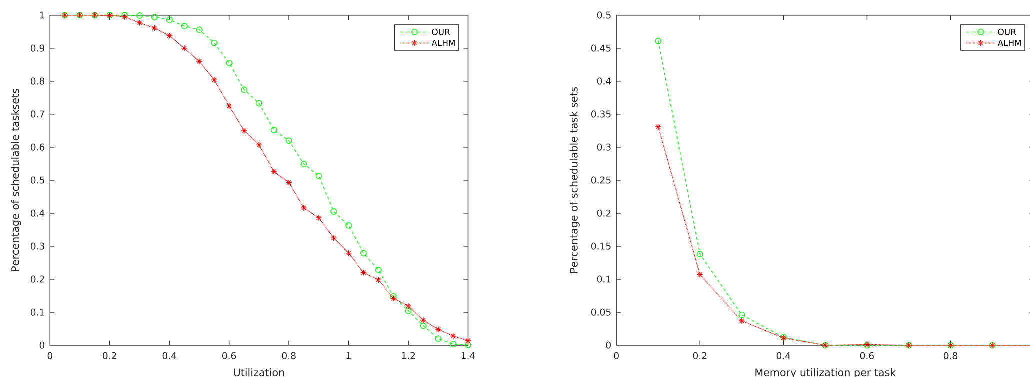
To generate the period of each task T_i, a log-uniform distribution is used with values ranging within $[100, 1000]$.

The tasks' execution times are calculated as $C_i = U_i \times T_i$. The generated tasks are assumed to have implicit deadlines and tasks' priorities are given by their periods following the Rate Monotonic approach, *i.e.,* the lower the period the higher the priority.

Since each task is composed of memory phases ($M_i = A_i + R_i$) and execution phases (E_i), in the experiments the value for the memory phases was set to a percentage p of the execution time C_i of each task. The other $(1 - p) \times C_i$ time units being assigned as the execution time of τ_i's E-phase, *i.e., $E_i = (1 - p) \times C_i$*. The total memory phase value is equally divided between A and R-phases so that $A_i = R_i = \frac{p \times C_i}{2}$. By default, p is set to 0.1.

In all experiments, 1000 random tasks sets were generated for each plotted utilization point. The percentage of task sets deemed schedulable by each analysis (the schedulability ratio) is used to compare the performances between approaches. In Figure 5.13, the green line ('OUR') presents the results for the approach presented in Section 5.5 and the red line ('ALHM') presents the results for [Alhammad, 2016; Alhammad and Pellizzoni, 2014].

The first set of experiments measured the percentage of task sets that are schedulable as a function of the task set total utilization. Figure 5.13a shows the results for $m = 4$ cores considering the generation parameters described above. In the figure, one can observe that 'OUR' approach



(a) % of schedulable task sets per utilization for m=2

(b) % of schedulable task sets as a function of the memory ratio p

Figure 5.14: Schedulability ratio for $m = 2$ and as a function of the memory ratio

performs better than 'ALHM', resulting in an increase of around 10% (up to 15%) of the number of task sets deemed schedulable when the total utilization value varies between 0.7 and 1.5.

In the second set of experiments, depicted in Figure 5.13b, the schedulability ratio is measured as a function of the number of cores, up to $m = 8$. In this experiment, the number of tasks per task set was set to $n = 10$ and the task set utilization was fixed at $U_\tau = 1.0$. As a side note, if the number of tasks per task set varies as a function of the number of cores, a higher percentage of task sets would be schedulable in systems with a large number of cores. This behaviour occurs due to the decrease in the utilization per task, and consequently a decrease in the utilization of memory phases. Therefore, keeping a fixed number of tasks allows one to better observe the influence of the increase in the number of cores.

As it can be seen in Figure 5.13b, both approaches cannot schedule any task set in a system with a single core (which is expected when the total utilization is 100% and fixed priority scheduling is used). But as the number of cores start to increase, the number of schedulable task sets also increases. For the 3-phase model that means that more tasks can execute their E-phases in parallel thus decreasing their response-time when compared to a system with a lower number of cores. Note that the difference between 'ALHM' and 'OUR' remains more or less constant and around 10%.

In the third set of experiments, shown in Figure 5.14b, the schedulability ratio is measured as a function of the ratio $p = \frac{M_i}{C_i}$. This experiment allows us to observe the influence of the bus on the schedulability of the system. In this experiment, the number of cores was set to $m = 4$, the task number $n = 10$ and the total utilization $U_\tau = 1.0$. The value of the memory ratio p varies in the interval $[0.1, 1]$ in increments of 0.1. As expected, increasing the memory utilization, decreases the percentage of schedulable task sets since the bus becomes the more and more loaded, hence increasing the access time to the memory. Another interesting aspect that can be observed is that after 40% of memory utilization both approaches perform almost exactly the same. These results are explained by the restrictions imposed by the bus on the execution of memory phases

in order to avoid interference. In particular, after around 50% of memory utilization per task, bus interference dominates both approaches avoiding any of them to take advantage of parallel execution of E-phases.

Finally, one should note that 'OUR' method does not dominate the analysis in [Alhammad, 2016; Alhammad and Pellizzoni, 2014]. There exist task sets that are deemed unschedulable by our method but which are deemed schedulable by 'ALHM', as depicted in Figure 5.14a. This is usually the case when the interference on the cores is much more constraining than the interference on the bus. It is somewhat understandable since 'ALHM' tackles the problem from a core perspective while we tackle it from a bus perspective. Since their modelling of the interference on the cores is more accurate than ours, they may perform better in such situations. However, we believe that the bus will usually be the limiting factor in multicore systems. Therefore, improving the modelling of the interference on the bus should provide better results in most cases.

5.6 Summary

In this chapter we addressed the problem of global scheduling of 3-phase tasks in COTS multicore systems.

In the first part of the chapter, we proposed an empirical validation of the model in order to understand the effects of interference in COTS multicore systems. In particular, we compared the performance of different interference-free priority assignment policies against a modified version of global EDF that is interference-prone. Results show that a policy that uses the period as a priority assignment criterion performs better than all other proposed policies. Moreover, the results also show that due to task contention the proposed policies for the 3-phase task model perform better than the modified version of global-EDF in those scenarios where several tasks execute in parallel.

In the second part of the chapter, we proposed a schedulability test for the global fixed-priority scheduling of the 3-phase task model. The proposed approach computes an upper-bound on the length of intervals when all cores are busy executing E-phases (the bus holes) and adds this length to the workload of a task due to memory phases. By looking at a problem window and analysing the worst-case interfering workload on a task under analysis the schedulability test is derived. The results show an increase on the schedulability ratio over the state of the art of around 10% in average and up to 15% in some cases.

The main conclusion that one can draw from the research work presented in this chapter is that the 3-phase task model can be useful in today's COTS multicore architectures in order to avoid task contention due to shared resources. Nevertheless, as depicted in all results and also confirmed by Becker et al. in [Becker et al., 2016], avoiding interference brings its cost in terms of system schedulability and scalability.

Future work includes the development of methods that compute tighter upper-bounds on the length of the bus holes so as to improve the accuracy of the schedulability test. A variation of the

schedulability test to compute worst-case response times may also be interesting for the development and analysis of real-time systems running on multicore platforms.

Chapter 6

Conclusion

Current multiprocessor platforms have two important characteristics that need to be considered. The first one is the possibility of executing applications simultaneously in all cores of the platform. The second is that general-purpose multiprocessor platforms (also known as COTS) are designed for the average-case scenario, meaning that resources such as the memory bus and caches are shared among the different cores in the system. From a real-time systems viewpoint, both characteristics bring an important challenge that forcefully needs to be addressed by the community, as the production trend is to deliver platforms with several cores. With this challenge in mind, in this dissertation, we devoted our attention to two important problems that need to be addressed in order to minimize the impact of adopting multiprocessor platforms by the real-time systems industry: (i) the problem of scheduling parallel real-time tasks and (ii) the problem of sharing resources among real-time tasks.

For the first problem, we explored the parallelism offered by multiprocessor platforms, by using real-time task models that focus on intra-task parallelism, such as the fork-join or the synchronous parallel task model. Two different solutions were proposed.

The first solution, presented in Chapter 3, fills the schedulability gap of synchronous parallel tasks by presenting an improved schedulability analysis for globally scheduled fixed-priority synchronous parallel task systems. Using as a base the technique proposed in [Bertogna and Cirinei, 2007] for sequential task sets, a response-time analysis test for synchronous parallel tasks is proposed by deriving a worst-case scenario that leads to the largest possible interference. The test uses novel concepts such as the sliding window technique (Section 3.5) and carry-out decomposition (Section 3.6) in order to make it sustainable and predictable. Results show that the proposed schedulability test significantly improves over the state of the art [Chwa et al., 2013] in terms of the number of schedulable task sets detected among randomly generated workloads.

The second solution, presented in Chapter 4, takes advantage of semi-partitioned scheduling to accommodate fork-join tasks that cannot be scheduled in any pure partitioned environment. Consequently, as the parallel jobs of each fork-join task can execute simultaneously on different cores, we take advantage of the work-stealing mechanism to dynamically load balance each task's workload. This allows one to reduce the average response time of the tasks without jeopardiz-

ing the schedulability of the whole system. To the best of our knowledge, we are the first using work-stealing in the context of a semi-partitioned scheduling scheme. To evaluate the proposed approach, we compare different allocation heuristics and, for two of them, we evaluate the improvement in terms of average response time obtained by using work-stealing. Results show that with this technique it is possible to reduce the average response time of tasks, and create additional room in the schedule for less-critical tasks (*e.g.,* aperiodic and best-effort tasks). In particular, the proposed approach allows one to achieve an average gain in terms of response-time of parallel tasks between 0 and nearly 15% per task.

The second problem addressed in this dissertation is the problem of resource sharing in multiprocessor systems. In particular, our goal was to provide a solution that avoids the undesired effects of the memory bus contention. Thus, in Chapter 5, the 3-phase task model is used in order to circumvent the uncontrolled sources of interference, occurring due to co-running tasks in multiprocessor systems. We start by conducting an empirical validation of the model in order to understand the effects of interference in COTS multicore systems. In particular, we compared the performance of different interference-free priority assignment policies against a modified version of global EDF that is interference-prone. Results show that due to task contention, the proposed policies for the 3-phase task model perform better than the modified version of global-EDF, in those scenarios where several tasks execute in parallel. Then, a schedulability test for the global fixed-priority scheduling of the 3-phase task model is derived. The proposed approach computes an upper-bound in a problem window by considering all intervals of time when all cores are busy executing E-phases (the bus holes) and adds this length to the workload of a task due to memory phases. When compared to the state of the art, the proposed approach shows an increase in the schedulability of around 10% in average and up to 15% in some cases.

Considering the two problems above, the questions posed in Chapter 1[1] and the central proposition of this dissertation, their answer is positive. In fact, we successfully provided ways of computing response-time upper bounds for parallel tasks executing in multiprocessor systems. This was done for the synchronous parallel task model in a global scheduling setting. In addition, we also used the fork-join task model in a semi-partitioned scheduling setting in which, by using dynamic load balancing via the application of work-stealing during runtime, it is possible to reduce the average response-time of real-time tasks. Moreover, we also have computed upper bounds on the interference for the 3-phase task model, executing on a multiprocessor system with a shared resource under a fixed-task priority global scheduling setting. Thus, we conclude that all the models presented in this dissertation are predictable and viable in real-time systems as it was proven by their sound schedulability analysis. However, their analyses present some pessimism that in some cases can be improved, as we show in the next section.

[1]For completeness, the questions posed in Chapter 1 were: (1) Is it possible to compute response-time upper bounds for parallel tasks when executing in multiprocessor systems?; (2) Considering a scenario with co-running tasks and a shared resource, is it possible to compute upper-bounds on the interference imposed by co-running tasks in a multiprocessor system?

6.1 Future Work

Some of the pessimism involved in the analyses can be improved and different future works are foreseen to improve them. For instance, for the contribution proposed in Chapter 3 using synchronous parallel tasks, the analysis could be refined by reducing the number of carry-in instances to consider, in a similar fashion to the analysis proposed in [Guan et al., 2009] for sequential tasks. In fact, this technique was already used in the analysis presented in Chapter 5.

For the analysis of the 3-phase task model presented in Chapter 5, we believe that it is possible to compute tighter upper-bounds on the length of the bus holes so as to improve the accuracy of the schedulability test. In addition, with the obtained results, we believe that it is possible to derive a test to compute the worst-case response times of each task. This test may be useful for the development and analysis of systems built using the 3-phase task model running on multiprocessor platforms.

Regarding work-stealing and its use in real-time systems, a remark must also be made. We believe that work-stealing is a viable algorithm for soft-real time scenarios where occasional deadline misses are allowed. Specially considering a global setting in order to take advantage of its load-balancing properties. Thus, it is also a possibility to pursue this idea of work-stealing in real-time settings as it appears to be very promising.

Finally, this dissertation focused on both problems in an independent manner, the problem of scheduling parallel real-time tasks and the problem of resource sharing in multiprocessor systems. However, for future work, we would like to consider the complex problem of considering both problems in conjunction, and if possible, to achieve an efficient solution.

References

Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In Pedro R. D'Argenio and Hernán Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 25–43, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40184-8.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-185-2. doi: 10.1145/341800.341801. URL http://doi.acm.org/10.1145/341800.341801.

Adapteva. Epiphany architecture reference, April 2014. URL http://www.adapteva.com/docs/epiphany_arch_ref.pdf.

Ahmed Alhammad. *Memory Effcient Scheduling forMulticore Real-time Systems*. PhD thesis, University of Waterloo, 2016.

Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 20:1–20:10, 2014.

James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2400-1. doi: 10.1109/ECRTS.2005.6. URL http://dx.doi.org/10.1109/ECRTS.2005.6.

Björn Andersson and Jan Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proceedings of the 21st IEEE Real-Time Systems Symposium – Work-in-Progress session, Orlando, Florida*, page 53–56, November 2000.

Bjorn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 322–334, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2676-4. doi: 10.1109/RTCSA.2006.45. URL http://dx.doi.org/10.1109/RTCSA.2006.45.

Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, pages 193–202, Dec 2001. doi: 10.1109/REAL.2001.990610.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Dobel, and Hermann Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 215–224, July 2013. doi: 10.1109/ECRTS.2013.31.

H. Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, April 2003. doi: 10.1109/IPDPS.2003.1213225.

Benjamin Bado, Laurent George, Pierre Courbin, and Joël Goossens. A semi-partitioned approach for parallel real-time scheduling. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS, pages 151–160, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1409-1. doi: 10.1145/2392987.2393006. URL http://doi.acm.org/10.1145/2392987.2393006.

Theodore P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 120–129, 2003.

S. Baruah. The non-cyclic recurring real-time task model. In *2010 31st IEEE Real-Time Systems Symposium*, pages 173–182, Nov 2010. doi: 10.1109/RTSS.2010.19.

Sanjoy Baruah and Alan Burns. Sustainable scheduling analysis. In *Proceedings of the 27th International Real-Time Systems Symposium*, RTSS '06, pages 159–168, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2761-2. doi: 10.1109/RTSS.2006.47. URL http://dx.doi.org/10.1109/RTSS.2006.47.

Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1):5–22, July 1999. ISSN 0922-6443.

Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, Jan 2003. ISSN 1573-1383. doi: 10.1023/A:1021711220939. URL https://doi.org/10.1023/A:1021711220939.

Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2 (4):301–324, Nov 1990. ISSN 1573-1383. doi: 10.1007/BF01995675. URL https://doi.org/10.1007/BF01995675.

A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 125–135, July 2011. doi: 10.1109/ECRTS.2011.20.

M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, July 2016. doi: 10.1109/ECRTS.2016.14.

S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Feb 2008. doi: 10.1109/ISSCC.2008.4523070.

Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, RTSS '07, pages 149–160, 2007.

Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 209–218, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2400-1. doi: 10.1109/ECRTS.2005.18. URL https://doi.org/10.1109/ECRTS.2005.18.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, September 1999. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/324133.324234. URL http://doi.acm.org/10.1145/324133.324234.

Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 225–233, July 2013. doi: 10.1109/ECRTS.2013.32.

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0084-1. doi: 10.1145/1811212.1811220. URL http://doi.acm.org/10.1145/1811212.1811220.

Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 25–34, 2013.

Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, May 2008. ISSN 0020-0190. doi: 10.1016/j.ipl.2007.11.014. URL http://dx.doi.org/10.1016/j.ipl.2007.11.014.

L. Cucu and Joel Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007. doi: 10.1109/DATE.2007.364536.

Liliana Cucu and Joel Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In *2006 IEEE Conference on Emerging Technologies and Factory Automation*, pages 397–404, Sept 2006. doi: 10.1109/ETFA.2006.355388.

D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *2013 8th IEEE International Symposium*

on Industrial Embedded Systems (SIES), pages 39–48, June 2013. doi: 10.1109/SIES.2013.6601469.

Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011a. ISSN 0360-0300. doi: 10.1145/1978802.1978814. URL http://doi.acm.org/10.1145/1978802.1978814.

Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011b.

Benôit Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benôit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2013. doi: 10.1109/HPEC.2013.6670342.

Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, Aug 2012. ISSN 1045-9219.

François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. *CoRR*, abs/1006.2637, 2010.

Maciej Drozdowski. Real-time scheduling of linear speedup parallel tasks. *Information Processing Letters*, 57(1):35–40, January 1996. ISSN 0020-0190. doi: 10.1016/0020-0190(95)00174-3. URL http://dx.doi.org/10.1016/0020-0190(95)00174-3.

Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software and Systems (ERTS'14)*, 2014a.

Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Perez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software and Systems (ERTS'14)*, Toulouse, France, February 2014b.

Vincent W. Freeh. A comparison of implicit and explicit parallel programming. *J. Parallel Distrib. Comput.*, 34(1):50–65, April 1996. ISSN 0743-7315. doi: 10.1006/jpdc.1996.0045. URL http://dx.doi.org/10.1006/jpdc.1996.0045.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/277652.277725. URL http://doi.acm.org/10.1145/277652.277725.

Free Software Foundation FSF. Gomp project, November 2014. URL https://gcc.gnu.org/projects/gomp/.

Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1995. ISBN 1-56592-074-0.

Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugère, Claire Pagetti, and Guy Durrieu. A complete tool-chain for an interference-free deployment of avionic applications on multi-core systems. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 1–13, 2015.

Joël Goossens and Vandy Berten. Gang ftp scheduling of periodic and parallel rigid real-time tasks. *CoRR*, abs/1006.2617, 2010.

Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 141–150, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1409-1.

Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2):32:1–32:36, November 2015. ISSN 0360-0300. doi: 10.1145/2830555. URL http://doi.acm.org/10.1145/2830555.

N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *2008 Real-Time Systems Symposium*, pages 137–146, 2008.

Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Proceedings of the 30th Real-Time Systems Symposium*, RTSS '09, pages 387–397, 2009.

Rhan Ha and Jane W.S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 162–171, Jun 1994. doi: 10.1109/ICDCS.1994.302407.

Ching-Chih Han and Kwei-Jay Lin. Scheduling parallelizable jobs on multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 59–67, 1989.

Shams Imam and Vivek Sarkar. Load balancing prioritized tasks via work-stealing. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 222–234, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48096-0.

M. Ivers, R. Ernst, and S. Schliecker. Integrated analysis of communicating tasks in mpsocs. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 288–293, Oct 2006. doi: 10.1145/1176254.1176325.

Klaus Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial-time approximation scheme. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 562–573, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44180-8. URL http://dl.acm.org/citation.cfm?id=647912.740670.

B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance [scheduling problems]. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 214–221, Oct 1995. doi: 10.1109/SFCS.1995.492478.

Jaeyeon Kang and D.G. Waddington. Load balancing aware real-time task partitioning in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 404–407, Aug 2012. doi: 10.1109/RTCSA.2012.71.

S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 249–258, July 2009. doi: 10.1109/ECRTS.2009.22.

Shinpei Kato and Yutaka Ishikawa. Gang edf scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 459–468, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3875-4. doi: 10.1109/RTSS.2009.42. URL http://dx.doi.org/10.1109/RTSS.2009.42.

T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12, July 2011. doi: 10.1109/ECRTS.2011.9.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-89683-4.

Martin Korsgaard and Sverre Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. *Real-Time Computing Systems and Applications, International Workshop on*, 1:3–14, 2011. ISSN 1533-2306. doi: http://doi.ieeecomputersociety.org/10.1109/RTCSA.2011.39.

Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4298-0. doi: 10.1109/RTSS.2010.42. URL http://dx.doi.org/10.1109/RTSS.2010.42.

Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. Technical Report TR-11-39, The University of Texas at Austin, Department of Computer Sciences, November 2011.

Patrick Leteinturier. Multi-core processors: Driving the evolution of automotive electronics architectures, September 2007. URL https://www.embedded.com/design/mcus-processors-and-socs/4007180/Multi-Core-Processors-Driving-the-Evolution-of-Automotive-Electronics-Architectures.

Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982. ISSN 0166-5316. doi: https://doi.org/10.1016/0166-5316(82)90024-4. URL http://www.sciencedirect.com/science/article/pii/0166531682900244.

J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, and C. Lu. Randomized work stealing for large scale soft real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 203–214, Nov 2016. doi: 10.1109/RTSS.2016.028.

Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-469-0. doi: 10.1145/1450095.1450117. URL http://doi.acm.org/10.1145/1450095.1450117.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL http://doi.acm.org/10.1145/321738.321743.

Cong Liu and James H. Anderson. Supporting soft real-time dag-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st Real-Time Systems Symposium*, RTSS '10, pages 3–13, Nov 2010.

Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Supporting real-time parallel task models with work-stealing, March 2012. Research Poster at The Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2012) (DATE Workshop).

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho, and Marko Bertogna. Response-time analysis of fork/join tasks in multiprocessor systems. In *Proceedings of Work-in-Progress Session of the 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, Paris, France, July 2013. Work in Progress Session.

Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 3:3–3:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2727-5. doi: 10.1145/2659787.2659815.

Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. Semi-partitioned scheduling of fork-join tasks using work-stealing. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 25–34, Oct 2015. doi: 10.1109/EUC.2015.30.

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. A closer look into the aer model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sept 2016. doi: 10.1109/ETFA.2016.7733567.

Cláudio Maia, Geoffrey Nelissen, Luís Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, Aug 2017a. doi: 10.1109/RTCSA.2017.8046313.

Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. Real-time semi-partitioned scheduling of fork-join tasks using work-stealing. *EURASIP Journal on Embedded Systems*, 2017(1):31, Sep 2017b. ISSN 1687-3963. doi: 10.1186/s13639-017-0079-5.

G. Manimaran, C. Siva Ram Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15(1):39–60, July 1998. ISSN 0922-6443. doi: 10.1023/A:1008022923184. URL http://dx.doi.org/10.1023/A:1008022923184.

Sebastian Mattheis, Tobias Schuele, Andreas Raabe, Thomas Henties, and Urs Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Proceedings of the 25th international conference on Architecture of Computing Systems*, ARCS'12, pages 172–183, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28292-8. doi: 10.1007/978-3-642-28293-5_15. URL http://dx.doi.org/10.1007/978-3-642-28293-5_15.

A.K. Mok and Deji Chen. A multiframe model for real-time tasks. *Software Engineering, IEEE Transactions on*, 23(10):635–645, Oct 1997. ISSN 0098-5589.

Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multicore scheduling in automotive ECUs. In *Embedded Real Time Software and Systems (ERTS'10)*, Toulouse, France, May 2010.

N. Tchidjo Moyo, E. Nicollet, F. Lafaye, and C. Moy. On schedulability analysis of non-cyclic generalized multiframe tasks. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 271–278, July 2010. doi: 10.1109/ECRTS.2010.24.

MPI. Message passing interface. http://www.mpi-forum.org/, January 2014.

G. J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002. ISSN 1433-0490. doi: 10.1007/s00224-001-1030-6. URL http://dx.doi.org/10.1007/s00224-001-1030-6.

Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. The Variability of Application Execution Times on a Multi-Core Platform. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 6:1–6:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-025-5. doi: 10.4230/OASIcs.WCET.2016.6. URL http://drops.dagstuhl.de/opus/volltexte/2016/6899.

Luís Nogueira and Luís Miguel Pinho. Server-based scheduling of parallel real-time tasks. In *Proceedings of the 10th International Conference on Embedded Software*, EMSOFT '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380374. URL http://doi.acm.org/10.1145/2380356.2380374.

Luís Nogueira, José C. Fonseca, Cláudio Maia, and Luís Miguel Pinho. Dynamic global scheduling of parallel real-time tasks. In *Proceedings of the 15th International Conference on Computational Science and Engineering*, CSE '12, pages 500–507, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4914-9. doi: 10.1109/ICCSE.2012.75. URL http://dx.doi.org/10.1109/ICCSE.2012.75.

J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. doi: 10.1109/EDCC.2012.27.

OpenMP. Openmp. http://openmp.org/, June 2011.

Oracle. Jsr 166: Concurrency utilities, June 2011. URL http://www.jcp.org/en/jsr/detail?id=166.

Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 741–746, 2010.

Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 269–279, 2011.

Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 140–149, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: 10.1145/258533.258570. URL http://doi.acm.org/10.1145/258533.258570.

Luís Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, and Michal Mardiak. P-socrates: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems*, 39(8):1190 – 1203, 2015. ISSN 0141-9331. doi: https://doi.org/10.1016/j.micpro.2015.06.004. URL http://www.sciencedirect.com/science/article/pii/S0141933115000836.

Manar Qamhieh, Laurent George, and Serge Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 13:13–13:22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2727-5. doi: 10.1145/2659787.2659818. URL http://doi.acm.org/10.1145/2659787.2659818.

Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086713. URL http://doi.acm.org/10.1145/2086696.2086713.

A Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill. Parallel real-time scheduling of dags. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2014. ISSN 1045-9219.

Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, volume 0 of *RTSS '11*, pages 217–226, Los Alamitos, CA, USA, Nov 2011. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/RTSS.2011.27.

Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 759–764, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL http://dl.acm.org/citation.cfm?id=1870926.1871108.

A. Schranzhofer, J. J. Chen, and L. Thiele. Timing analysis for tdma arbitration in resource sharing systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720.

John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988. ISSN 0018-9162. doi: 10.1109/2.7053. URL http://dx.doi.org/10.1109/2.7053.

R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016. doi: 10.1109/RTAS.2016.7461321.

L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, volume 4, pages 101–104 vol.4, 2000. doi: 10.1109/ISCAS.2000.858698.

Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2010.78.

Qi Wang and Grabriel Parmer. Fjos: Practical, predictable, and efficient system support for fork/join parallelism. In *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, RTAS '14, Washington, DC, USA, 2014. IEEE Computer Society.

Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL http://doi.acm.org/10.1145/216585.216588.

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736036. URL http://doi.acm.org/10.1145/1736020.1736036.