Mälardalen University Press Dissertations
No. 124

RESOURCE SHARING IN REAL-TIME SYSTEMS ON MULTIPROCESSORS

Farhang Nemati

2012

School of Innovation, Design and Engineering

ISBN 978-91-7485-063-5
ISSN 1651-4238
Printed by Mälardalen University, Västerås, Sweden

Populärvetenskaplig sammanfattning

Klassiska programvarusystem som exempelvis ordbehandlare, bildbehandlare och webbläsare har typiskt en förväntad funktion att uppfylla, till exempel, en användare ska kunna producera typsatt skrift under relativt smärtfria former. Man kan generalisera och säga att korrekt funktion är av yttersta vikt för hur populär och användbar en viss programvara är medans exakt hur en viss funktion realiseras är av underordnad betydelse. Tittar man istället på så kallade realtidssystem så är, utöver korrekt funktionalitet hos programvaran, också det tidsmässiga utförandet av funktionen av yttersta vikt. Med andra ord så bör, eller måste, de funktionella resultaten produceras inom vissa specificerade tidsramar. Ett exempel är en airbag som inte får utlösas för tidigt eller för sent. Detta kan tyckas relativt okomplicerat, men tittar man närmare på hur realtidssystem är konstruerade så finner man att ett system vanligtvis är uppdelat i ett antal delar som körs (exekveras) parallellt. Dessa delar kallas för tasks och varje task är en sekvens (del) av funktionalitet, eller instruktioner, som genomförs samtidigt med andra tasks. Dessa tasks exekverar på en processor, själva hjärnan i en dator. Realtidsanalyser har tagits fram för att förutsäga hur sekvenser av taskexekveringar kommer att ske givet att antal tasks och deras karakteristik.

Utvecklingen och modernisering av processorer har tvingat fram så kallade multicoreprocessorer - processorer med multipla hjärnor (cores). Tasks kan nu, jämfört med hur det var förr, köras parallellt med varandra på olika cores, vilket samtidigt förbättrar effektiviteten hos en processor med avseende på hur mycket som kan exekveras, men även komplicerar både analys och förutsägbarhet med avseende på hur dessa tasks körs. Analys behövs för att kunna förutsäga korrekt tidsmässigt beteende hos programvaran i ett realtidssystem.

I denna doktorsavhandling har vi föreslagit en metod att fördela ett realtidssystems tasks på ett antal processorer givet en multicorearkitektur. Denna metod ökar avsevärt både prestation, förutsägbarhet och resursutnyttjandet hos det multicorebaserade realtidsystemet genom att garantera tidsmässigt korrekt exekvering av programvarusystem med komplexa beroenden vilka har direkt påverkan på hur lång tid ett task kräver för att exekvera.

Inom industriella system brukar stora och komplexa programvarusystem delas in i flera delar (applikationer) som var och en kan utvecklas oberoende av varandra och parallellt. Men det kan hända att applikationer delar olika resurser när de exekverar tillsammans på en multi-core arkitektur. I denna avhandling har vi föreslagit nya metoder för att hantera resurser som delas mellan realtidsapplikationer som exekverar på en multi-core arkitektur.

Abstract

In recent years multiprocessor architectures have become mainstream, and multi-core processors are found in products ranging from small portable cell phones to large computer servers. In parallel, research on real-time systems has mainly focused on traditional single-core processors. Hence, in order for real-time systems to fully leverage on the extra capacity offered by new multi-core processors, new design techniques, scheduling approaches, and real-time analysis methods have to be developed.

In the multi-core and multiprocessor domain there are mainly two scheduling approaches, global and partitioned scheduling. Under global scheduling each task can execute on any processor at any time while under partitioned scheduling tasks are statically allocated to processors and migration of tasks among processors is not allowed. Besides simplicity and efficiency of partitioned scheduling protocols, existing scheduling and synchronization techniques developed for single-core processor platforms can more easily be extended to partitioned scheduling. This also simplifies migration of existing systems to multi-cores. An important issue related to partitioned scheduling is the distribution of tasks among the processors, which is a bin-packing problem.

In this thesis we propose a blocking-aware partitioning heuristic algorithm to distribute tasks onto the processors of a multi-core architecture. The objective of the proposed algorithm is to decrease the blocking overhead of tasks, which reduces the total utilization and has the potential to reduce the number of required processors.

In industrial embedded software systems, large and complex systems are usually divided into several components (applications) each of which is developed independently without knowledge of each other, and potentially in parallel. However, the applications may share mutually exclusive resources when they co-execute on a multi-core platform which introduce a challenge for the techniques needed to ensure predictability. In this thesis we have proposed a

new synchronization protocol for handling mutually exclusive resources shared among real-time applications on a multi-core platform. The schedulability analysis of each application is performed in isolation and parallel and the requirements of each application with respect to the resources it may share are included in an interface. The protocol did not originally consider any priorities among the applications. We have proposed an additional version of the protocol which grants access to resources based on priorities assigned to the applications. We have also proposed an optimal priority assignment algorithm to assign unique priorities to the applications sharing resources. Our evaluations confirm that the protocol together with the priority assignment algorithm outperforms existing alternatives in most cases.

In the proposed synchronization protocol each application is assumed to be allocated on one dedicated core. However, in this thesis we have further extended the synchronization protocol to be applicable for applications allocated on multiple dedicated cores of a multi-core platform. Furthermore, we have shown how to efficiently calculate the resource hold times of resources for applications. The resource hold time of a resource for an application is the maximum duration of time that the application may lock the resource whenever it requests the resource. Finally, the thesis discusses and proposes directions for future work.

Acknowledgments

First, I want to thank my supervisors, Thomas Nolte, Christer Norström, and Anders Wall for guiding and helping me during my studies. I specially thank Thomas Nolte for all his support and encouragement.

I would like to give many thanks to the people from whom I have learned many things in many aspects; Hans Hansson, Ivica Crnkovic, Paul Pettersson, Sasikumar Punnekkat, Björn Lisper, Mikael Sjödin, Lars Asplund, Mats Björkman, Kristina Lundkvist, Jan Gustafsson, Cristina Seceleanu, Frank Lüders, Jan Carlson, Dag Nyström, Andreas Ermedahl, Radu Dobrin, Daniel Sundmark, Rikard Land, Damir Isovic, Kaj Hänninen, Daniel Flemström, and Jukka Mäki-Turja.

I also thank people at IDT; Carola, Gunnar, Malin, Åsa, Jenny, Ingrid, Susanne, for making many things easier. During my studies, trips, coffee breaks and parties I have had a lot of fun and I wish to give many thanks to Aida, Aneta, Séverine, Hongyu, Rafia, Kathrin, Sara A., Sara D., Shahina, Adnan, Andreas H., Andreas G., Moris, Hüseyin, Bob (Stefan), Nima, Luis (Yue Lu), Mohammad, Mikael Å., Daniel H., Hang, Jagadish, Nikola, Federico, Saad, Mehrdad, Mobyen, Johan K., Abhilash, Juraj, Luka, Leo, Josip, Antonio, Tibi, Sigrid, Barbara, Batu, Fredrik, Giacomo, Guillermo, Svetlana, Raluca, Eduard and all others for all the fun and memories.

I want to give my gratitude to my parents for their support and love in my life. Last but not least, my special thanks goes to my wife Samal, for all the support, love and fun. I would also wish to thank my lovely daughter Ronia just for existing and making our family complete.

This work has been supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

Farhang Nemati
Västerås, May, 2012

List of Publications

Papers Included in the PhD Thesis[1]

Paper A *Partitioning Real-Time Systems on Multiprocessors with Shared Resources*. Farhang Nemati, Thomas Nolte, Moris Behnam. In 14th International Conference On Principles Of Distributed Systems (OPODIS'10), pages 253-269, December, 2010.

Paper B *Independently-developed Real-Time Systems on Multi-cores with Shared Resources*. Farhang Nemati, Moris Behnam, Thomas Nolte. In 23rd Euromicro Conference on Real-Time Systems (ECRTS'11), pages 251-261, July, 2011.

Paper C *Resource Sharing among Prioritized Real-Time Applications on Multi-cores*. Farhang Nemati, Thomas Nolte. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-265/2012-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2012 (submitted to conference).

Paper D *Resource Sharing among Real-Time Components under Multiprocessor Clustered Scheduling*. Farhang Nemati, Thomas Nolte. Journal of Real-Time Systems (under revision).

Paper E *Resource Hold Times under Multiprocessor Static-Priority Global Scheduling*. Farhang Nemati, Thomas Nolte. In 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pages 197-206, August, 2011.

[1] The included articles have been reformatted to comply with the PhD layout

Additional Papers, not Included in the PhD Thesis

Journals

1. *Sharing Resources among Independently-developed Systems on Multi-cores*. Farhang Nemati, Moris Behnam, Thomas Nolte. ACM SIGBED Review, vol 8, nr 1, pages 46-53, ACM, March, 2011.

Conferences and Workshops

1. *Towards Resource Sharing by Message Passing among Real-Time Components on Multi-cores*. Farhang Nemati, Rafia Inam, Thomas Nolte, Mikael Sjödin. In 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress (WiP) session, pages 1-4, September, 2011.

2. *Towards an Efficient Approach for Resource Sharing in Real-Time Multiprocessor Systems*. Moris Behnam, Farhang Nemati, Thomas Nolte, Håkan Grahn. In 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), Work-in-Progress (WiP) session, pages 99-102, June, 2011.

3. *Independently-developed Systems on Multi-cores with Shared Resources*. Farhang Nemati, Moris Behnam, Thomas Nolte. In 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'10) in conjunction with the 31th IEEE Real-Time Systems Symposium (RTSS'10), December, 2010.

4. *A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors*. Farhang Nemati, Thomas Nolte. In 15th IEEE International Conference on Emerging Techonologies and Factory (ETFA'10), Work-in-Progress (WiP) session, pages 1-4, September, 2010.

5. *Multiprocessor Synchronization and Hierarchical Scheduling*. Farhang Nemati, Moris Behnam, Thomas Nolte. In 38th International Conference on Parallel Processing (ICPP'09) Workshops, pages 58-64, September, 2009.

6. *Investigation of Implementing a Synchronization Protocol under Multiprocessors Hierarchical Scheduling*. Farhang Nemati, Moris Behnam,

Thomas Nolte, Reinder J. Bril. In 14th IEEE International Conference on Emerging Technologies and Factory (ETFA'09), pages 1670-1673, September, 2009.

7. *Efficiently Migrating Real-Time Systems to Multi-Cores*. Farhang Nemati, Moris Behnam, Thomas Nolte. In 14th IEEE Conference on Emerging Technologies and Factory (ETFA'09), pages 1-8, 2009.

8. *Towards Hierarchical Scheduling in AUTOSAR*. Mikael Åsberg, Moris Behnam, Farhang Nemati, Thomas Nolte. In 14th IEEE International Conference on Emerging Techonologies and Factory (ETFA'09), pages 1181-1188, September, 2009.

9. *An Investigation of Synchronization under Multiprocessors Hierarchical Scheduling*. Farhang Nemati, Moris Behnam, Thomas Nolte. In the 21st Euromicro Conference on Real-Time Systems (ECRTS'09), Work-in-Progress (WiP) session, pages 49-52, July, 2009.

10. *Towards Migrating Legacy Real-Time Systems to Multi-Core Platforms*. Farhang Nemati, Johan Kraft, Thomas Nolte. In 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), Work-in-Progress (WiP) session, pages 717-720, September, 2008.

11. *Validation of Temporal Simulation Models of Complex Real-Time Systems*. Farhang Nemati, Johan Kraft, Christer Norström. In 32nd IEEE International Computer Software and Application Conference (COMPSAC'08), pages 1335-1340, July, 2008.

Contents

I

Thesis

Chapter 1

Introduction

Inherent in problems with power consumption and related thermal problems, multi-core platforms seem to be the way forward towards increasing performance of processors, and single-chip multiprocessors (multi-cores) are today the dominating technology for desktop computing.

The performance improvements of using multi-core processors depend on the nature of the applications as well as the implementation of the software. To take advantage of the concurrency offered by a multi-core architecture, appropriate algorithms have to be used to divide the software into tasks (threads) and efficient scheduling techniques and partitioning algorithms to distribute tasks fairly on processors are required to increase the overall performance.

Two main approaches for scheduling real-time systems on multiprocessors exist [1, 2, 3, 4]; global and partitioned scheduling. Under global scheduling protocols, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor. A single global queue is used for storing tasks. A task can be preempted on a processor and resumed on another processor, i.e., migration of tasks among cores is permitted. Under a partitioned scheduling protocol, tasks are statically assigned to processors and the tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) and EDF. Each processor is associated with a separate ready queue for scheduling task jobs. There are systems in which some tasks cannot migrate among cores while other tasks can migrate. For such systems neither global or partitioned scheduling methods can be used. A two-level hybrid scheduling approach [4], which is a mix of global and partitioned scheduling methods, is used for those systems.

In the multiprocessor research community, considerable work has been done on scheduling algorithms where it is assumed that tasks are independent. However, synchronization in the multiprocessor context has not received enough attention. Under partitioned scheduling, if all tasks that share the same resources can be allocated on the same processor then the uniprocessor synchronization protocols can be used [5]. This is not always possible, and some adjustments have to be done to the protocols to support synchronization of tasks across processors. The uniprocessor lock-based synchronization protocols have been extended to support inter processor synchronization among tasks [6, 7, 8, 9, 10, 11, 12]. However, under global scheduling methods, the uniprocessor synchronization protocols [13, 1] can not be reused without modification. Instead, new lock-based synchronization protocols have been developed to support resource sharing under global scheduling methods [9, 14].

Partitioned scheduling protocols have been used more often and are supported widely by commercial real-time operating systems [15], inherent in their simplicity, efficiency and predictability. Besides, the well studied uniprocessor scheduling and synchronization methods can be reused for multiprocessors with fewer changes. However, partitioning is known to be a bin-packing problem which is a NP-hard problem in the strong sense; hence finding an optimal solution in polynomial time is not realistic in the general case. Thus, to take advantage of the performance offered by multi-cores, partitioned scheduling protocols have to be coordinated with appropriate partitioning algorithms [15, 16]. Heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been developed to find a near-optimal partitioning [2, 3]. However, the scheduling protocols and existing partitioning algorithms for multiprocessors mostly assume independent tasks.

The availability of multi-core platforms has attracted a lot of attention in multiprocessor embedded software analysis and runtime policies, protocols and techniques. As the multi-core platforms are to be the defacto processors, the industry must cope with a potential migration towards multi-core platforms. The industry can benefit from multi-core platforms as these platforms facilitate hardware consolidation by co-executing multiple real-time applications on a shared multi-core platform.

An important issue for industry when it comes to migration to multi-cores is the *existing* applications. When migrating to multi-cores it has to be possible that several applications can co-execute on a shared multi-core platform. The (often independently-developed) applications may have been developed with different techniques, e.g., several real-time applications that will co-execute on a multi-core may have different scheduling policies. However, when the appli-

cations co-execute on the same multi-core platform they may share resources that require mutual exclusive access. Two challenges to overcome when migrating existing applications to multi-cores are how to migrate the applications with minor changes, and how to abstract key properties of applications sufficiently, such that the developer of one application does not need to be aware of particular techniques used in other applications.

Looking at industrial software systems, to speed up their development, it is not uncommon that large and complex systems are divided into several semi-independent subsystems each of which is developed independently. The subsystems which may share resources will eventually be integrated and co-execute on the same platform. This issue has got attention and has been studied in the uniprocessor domain [17, 18, 19]. However, new techniques are sought for scheduling semi-independent subsystems on multi-cores.

1.1 Contributions

The main contributions of this thesis are in the area of partitioning heuristics and synchronization protocols for multi-core real-time systems. In the following two subsections we present these contributions in more details.

1.1.1 Partitioning Heuristic Algorithm

As mentioned in Section 1, the partitioning algorithms that partition an application on a multi-core have not considered resource sharing. Considering resource sharing in partitioning algorithms leads to decreased blocking and better schedulability of a task set. We have proposed a partitioning algorithm, based on bin-packing, for allocating tasks onto processors of a multi-core platform (Chapter 3). Tasks may access mutually exclusive resources and the aim of the algorithm is to decrease the overall blocking overhead in the system. An efficient partitioning algorithm may consequently increase the schedulability of a task set and reduce the number of processors. We proposed the partitioning algorithm in Paper A and we compared it to a similar algorithm originally proposed by Lakshmanan et al. [15]. Our new algorithm has shown to have the potential to decrease the total number of required processors and it mostly performs better than the similar existing algorithm.

1.1.2 Synchronization Protocols for Real-Time Applications in an Open System on Multiprocessors

The multi-core platforms offer an opportunity for hardware consolidation and open systems where multiple independently-developed real-time applications can co-execute on a shared multi-core platform. The applications may, however, share mutually exclusive resources, imposing a challange when trying to achieve independence. Methods, techniques and protocols are needed to support handling of shared resources among the co-executing applications. We aim to tackle this important issue:

1. **Synchronization Protocol for Real-Time Applications under Partitioned Scheduling**

   (a) In Paper B we proposed a synchronization protocol for resource sharing among independently-developed real-time applications on a multi-core platform, where each application is allocated on a dedicated core. The protocol is called Multiprocessors Synchronization protocol for real-time Open Systems (MSOS). In the paper, we have presented an interface-based schedulability condition for MSOS. The interface abstracts the resource sharing of an application allocated on one processor through a set of requirements that have to be satisfied to guarantee the schedulability of the application. In Paper B, we further evaluated and compared MSOS to two existing synchronization protocols for partitioned scheduling.

   (b) The original MSOS assumes no priority setting among the applications, i.e., applications waiting for shared resources are enqueued in a First-In First-Out (FIFO) manner. We extended MSOS to support prioritized applications which increases the schedulability of the applications. This contribution is directed by Paper C. In the paper, we extended the interface of applications and their schedulability analysis to support prioritized applications. To distinguish the extended MSOS from the original one we call the original MSOS and the extended one as MSOS-FIFO and MSOS-Priority respectively. In Paper C, by means of simulations, we evaluated and compared MSOS-Priority to the key state-of-the-art synchronization protocols as well as to MSOS-FIFO.

   (c) In Paper C, we proposed an optimal priority setting algorithm which assigns priorities to the applications under MSOS-Priority. As con-

firmed by the evaluation results, the algorithm increases the schedulability of applications significantly.

2. **Synchronization Protocol for Real-Time Applications under Clustered Scheduling**

   (a) In Paper D, we proposed a synchronization protocol, called Clustered MSOS (C-MSOS), for supporting resource sharing among real-time applications where each application is allocated on a dedicated set of cores (cluster). In the paper we derived the interface-based schedulability analysis for four alternatives of C-MSOS. The alternatives are distinguished by the way the queues in which applications and tasks wait for shared resources are handled. In a simulation-based evaluation in Paper D we have compared all four alternatives of C-MSOS.

   (b) In Paper D, in order to minimize the interference of applications regarding the shared resources, we let the priority of a task holding a *global resource* (i.e., a global resource is shared among multiple applications) be raised to be higher than any priority in its application. In this way no other task executing in non-critical sections can delay a task holding a global resource. This means that the *Resource Hold Times* (RHT) of global resources are minimized. The RHT of a global resource in an application is the maximum time that any task in the application may hold (lock) the resource. However, boosting the priority of any task holding a global resource may make an application unschedulable. Therefore the priorities of tasks holding global resources are raised as long as the application remains schedulable, i.e., boosting the priorities should never compromise the schedulability of the application. Under uniprocessor platforms, it has been shown [20, 21] that it is possible to achieve one single optimal solution, when trying to set the best priority ceilings for global resources. However, this is not the case when an application is scheduled on multiple processors (i.e., tasks in the application are scheduled by a global scheduling policy). In Paper E we calculated the RHT's for global resources while assuming that the priorities of tasks holding global resources can be boosted as far as the application remains schedulable. We have shown that despite of uniprocessor platforms where there exists

one optimal solution, on multiprocessors there can exist multiple Pareto-optimal solutions.

1.2 Thesis Outline

The outline of the thesis is as follows. In Chapter 2 we give a background describing real-time systems, scheduling, multiprocessors, multi-core architectures, the problems and the existing solutions, e.g., scheduling and synchronization protocols. Chapter 3 gives an overview of our proposed heuristic partitioning algorithm. In Chapter 4 we have presented our proposed synchronization protocol for both non-proiritized and prioritized applications. In the chapter we have further presented the extension of our proposed protocol to clustered scheduling, i.e., where one application can be allocated on multiple dedicated cores. In Chapter 4 we have also discussed efficient resource hold time calculations. In Chapter 5 we present our conclusion and future work. We present the technical overview of the papers that are included in this thesis in Chapter 6, and we present these papers in Chapters 7 - 11 respectively.

Chapter 2

Background

2.1 Real-Time Systems

In a real-time system, besides the functional correctness of the system, the output has to satisfy timing attributes as well [22], e.g., the outputs have to be delivered within deadlines. A real-time system is typically developed following a concurrent programming approach in which a system may be divided into several parts, called *tasks*, and each task, which is a sequence of operations, executes in parallel with other tasks. A task may issue an infinite number of instances called *jobs* during run-time.

Each task has timing attributes, e.g., *deadline* before which the task should finish its execution, *Worst Case Execution Time* (WCET) which is the maximum time that a task needs to perform and complete its execution when executing without interference from other tasks. The execution of a task can be periodic or aperiodic; a periodic task is triggered with a constant time, denoted as *period*, in between instances, and an aperiodic task may be triggered at any arbitrary time instant.

Real-time systems are generally categorized into two categories; *hard real-time systems* and *soft real-time systems*. In a hard real-time system tasks are not allowed to miss their deadlines, while in a soft real-time system some tasks may miss their deadlines. A safety-critical system is a type of hard-real time system in which missing deadlines of tasks may lead to catastrophic incidents, hence in such a system missing deadlines are not tolerable.

2.2 Multi-core Platforms

A multi-core (single-chip multiprocessor) processor is a combination of two or more independent processors (cores) on a single chip. The cores are connected to a single shared memory via a shared bus. The cores typically have independent L1 caches and may share an on-chip L2 cache.

Multi-core architectures are today the dominating technology for desktop computing and are becoming the defacto processors overall. The performance of using multiprocessors, however, depends on the nature of the applications as well as the implementation of the software. To take advantage of the concurrency offered by a multi-core architecture, appropriate algorithms have to be used to divide the software into tasks (threads) and to distribute tasks on cores to increase the system performance. If an application is not (or cannot) be fairly divided into tasks, e.g., one task does all the heavy work, a multi-core will not help improving the performance significantly. Real-time systems can highly benefit from multi-core processors, as they are typically multi-threaded, hence making it easier to adapt them to multi-cores than single-threaded, sequential programs, e.g., critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast.

While multi-core platforms offer significant advantages, they also introduce big challenges. Existing software systems need adjustments to be adapted on multi-cores. Many existing legacy real-time systems are very large and complex, typically consisting of huge amount of code. It is normally not an option to throw them away and to develop a new system from scratch. A significant challenge is to adapt them to work efficiently on multi-core platforms. If the system contains independent tasks, it is a matter of deciding on which processor each task should be executed. In this case scheduling protocols from single-processor platforms can easily be reused. However, tasks are usually not independent and they may share resources. This means that, to be able to adapt the existing systems to be executed on a multi-core platform, synchronization protocols are required to be changed or new protocols have to be developed.

For hard real-time systems, from a practical point of view, a static assignment of processors, i.e., partitioned scheduling (Section 2.3.1), is often the more common approach [2], often inherent in reasons of predictability and simplicity. Also, the well-studied and verified scheduling analysis methods from the single-processor domain has the potential to be reused. However, fairly allocating tasks onto processors (partitioning) is a challenge, which is a

bin-packing problem.

Finally, the processors on a multi-core can be identical, which means that all processors have the same performance, this type of multi-core architectures are called *homogenous*. However, the architecture may suffer from heat and power consumption problems. Thus, processor architects have developed multi-core architectures consisting of processors with different performance in which tasks can run on appropriate processors, i.e., the tasks that do not need higher performance can run on processors with lower performance, decreasing energy consumption.

2.3 Real-Time Scheduling on Multiprocessors

The major approaches for scheduling real-time systems on multiprocessors are *partitioned scheduling*, *global scheduling*, and the combination of these two called *hybrid scheduling* [1, 2, 3, 4].

2.3.1 Partitioned Scheduling

Under partitioned scheduling tasks are statically assigned to processors, and the tasks within each processor are scheduled by a single-processor scheduling protocol, e.g., RM and EDF [23]. Each task is allocated to a processor on which its jobs will run. Each processor is associated with a separate ready queue for scheduling its tasks' jobs.

An advantage of partitioned scheduling is that well-understood and verified scheduling analysis from the uniprocessor domain has the potential to be reused. Another advantage is the run-time efficiency of these protocols as the tasks and jobs do not suffer from migration overhead. A disadvantage of partitioned scheduling is that it is a bin-packing problem which is known to be NP-hard in the strong sense, and finding an optimal distribution of tasks among processors in polynomial time is not generally realistic. Another disadvantage of partitioned scheduling algorithms is that prohibiting migration of tasks among processors decreases the utilization bound, i.e., it has been shown [3] that task sets exist that are only schedulable if migration among processors is allowed. Non-optimal heuristic algorithms have been used for partitioning a task set on a multiprocessor platform. An example of a partitioned scheduling algorithm is Partitioned EDF (P-EDF) [2].

2.3.2 Global Scheduling

Under global scheduling algorithms tasks are scheduled by a single system-level scheduler, and each task or job can be executed on any processor. A single global queue is used for storing ready jobs. At any time instant, at most m ready jobs with highest priority among all ready jobs are chosen to run on a multiprocessor consisting of m processors. A task or its jobs can be preempted on one processor and resumed on another processor, i.e., migration of tasks (or its corresponding jobs) among cores is permitted. An example of a global scheduling algorithm is Global EDF (G-EDF) [2]. The global scheduling algorithms are not necessarily optimal either, although in the research community new multiprocessor scheduling algorithms have been developed that are optimal. Proportionate fair (Pfair) scheduling approaches are examples of such algorithms [24, 25]. However, this particular class of scheduling algorithms suffers from high run-time overhead as they may have to increase the number of preemptions and migrations significantly. However, there have been research works on decreasing this overhead in the multiprocessor scheduling algorithms; e.g., the work by Levin et al. [26].

2.3.3 Hybrid Scheduling

There are systems that cannot be scheduled by either pure partitioned or pure global scheduling; for example some tasks cannot migrate among cores while other tasks are allowed to migrate. An example approach for those systems is the two-level hybrid scheduling approach [4], which is based on a mix of global and partitioned scheduling methods. In such protocols, at the first level a global scheduler assigns jobs to processors and at the second level each processor schedules the assigned jobs by a local scheduler.

Recently more general approaches, such as cluster-based scheduling [27, 28], have been proposed which can be categorized as a generalization of partitioned and global scheduling protocols. Using such an approach, tasks are statically assigned to clusters and tasks within each cluster are globally scheduled. Cluster-based scheduling can be physical or virtual. In physical cluster-based scheduling the virtual processors of each cluster are statically mapped to a subset of physical processors of the multiprocessor [27]. In virtual cluster-based scheduling [28] the processors of each cluster are dynamically mapped (one-to-many) onto processors of the multiprocessor. Virtual clustering is more general and less sensitive to task-cluster mapping compared to physical clustering.

2.4 Resource Sharing on Multiprocessors

Generally there are two classes of resource sharing, i.e., lock-based and lock-free synchronization protocols. In the lock-free approach [29, 30], operations on simple software objects, e.g., stacks, linked lists, are performed by retry loops, i.e., operations are retried until the object is accessed successfully. The advantages of lock-free algorithms is that they do not require kernel support and as there is no need to lock, priority inversion does not occur. The disadvantage of these approaches is that it is not easy to apply them to hard real-time systems as the worst case number of retries is not easily predictable. In this thesis we have focused on the lock-based approach, thus in this section we present an overview of a non-exhaustive list of the existing lock-based synchronization methods.

On a multiprocessor platform a job, besides lower priority jobs, can be blocked by higher priority jobs that are assigned to different processors as well. This does not rise any problem on uniprocessor platforms. Another issue, which is not the case in the existing uniprocessor synchronization techniques, is that on a uniprocessor, a job J_i can not be blocked by lower priority jobs arriving after J_i. However, on a multiprocessor, a job J_i can be blocked by the lower priority jobs arriving after J_i if they are executing on different processors. Those cases introduce more complexity and pessimism into schedulability analysis.

The existing lock-based synchronization protocols can be categorized as *suspend-based* and *spin-based* protocols. Under a suspend-based protocol a task requesting a resource that is shared across processors suspends if the resource is locked by another task. Under a spin-based protocol a task requesting the locked resource keeps the processor and performs spin-lock (busy wait).

2.4.1 The Multiprocessor Priority Ceiling Protocol (MPCP)

Rajkumar proposed MPCP (Multiprocessor Priority Ceiling Protocol) [6], that extends PCP (Priority Ceiling Protocol) [13] to shared memory multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS (Fixed Priority Scheduling). MPCP is a suspend-based protocol under which tasks waiting for a global resource suspend and are enqueued in an associated prioritized global queue. Under MPCP, the priority

of a task within a global critical section (gcs), in which it requests a global resource, is boosted to be greater than the highest priority among all local tasks. This priority is called remote ceiling. A gcs can only be preempted by other gcs's that have higher remote ceiling. Lakshmanan et al. [15] extended a spin-based alternative of MPCP.

MPCP is used for synchronizing a set of tasks sharing lock-based resources under a partitioned FPS protocol, i.e., RM. Under MPCP, resources are divided into *local* and *global* resources. The local resources are protected using a uniprocessor synchronization protocol, i.e., PCP.

Under MPCP, the blocking time of a task, in addition to the local blocking, has to include the remote blocking terms where a task is blocked by tasks executing on other processors. However, the maximum remote blocking time of a job is bounded and is a function of the duration of critical sections of other jobs. This is a consequence of assigning any gcs a ceiling greater than the priority of any other task, hence a gcs can only be blocked by another gcs and not by any non-critical section. Assume ρ_H is the highest priority among all tasks. The remote ceiling of a job J_i executing within a gcs equals to $\rho_H + 1 + \max\{\rho_j | \tau_j$ *requests* R_k *and* τ_j *is not on* J_i*'s processor*$\}$.

Global critical sections cannot be nested in local critical sections and vice versa. Global resources potentially lead to high blocking times, thus tasks sharing the same resources are preferred to be assigned to the same processor as far as possible. We have proposed an algorithm that attempts to reduce the blocking times by assigning tasks to appropriate processors (Chapter 3).

2.4.2 The Multiprocessor Stack Resource Policy (MSRP)

Gai et al. [7] presented MSRP (Multiprocessor SRP), which is an extension of SRP (Stack-based Resource allocation Protocol) [1] to multiprocessors and it is a spin-based synchronization protocol. MSRP is used for synchronizing a set of tasks sharing lock-based resources under a partitioned EDF (P-EDF). The shared resources are classified as either local or global resources. Tasks are synchronized on local resources using SRP, and access to global resources is guaranteed a bounded blocking time. Further, under MSRP, when a task is blocked on a global resource it performs *busy wait* (spin lock). This means that the processor is kept busy without doing any work, hence the duration of the spin lock should be as short as possible which means locking a global resource should be reduced as far as possible. To achieve this goal under MSRP, the tasks executing in global critical sections become non-preemptive. The tasks blocked on a global resource are added to a FIFO queue. Global critical sec-

tions are not allowed to be nested under MSRP.

Gai et al. [8] compared their implementation of MSRP to MPCP. They pointed out that the complexity of implementation as a disadvantage of MPCP and that wasting more local processor time (due to busy wait) as a disadvantage of MSRP. They have performed two case studies for the comparison. The results show that MPCP works better when the duration of global critical sections are increased while MSRP outperforms MPCP when critical sections become shorter. Also in applications where tasks access many resources, and resources are accessed by many tasks, which lead to more pessimism in MPCP, MSRP has a significant advantage compared to MPCP.

2.4.3 The Flexible Multiprocessor Locking Protocol (FMLP)

Block et al. [9] presented FMLP (Flexible Multiprocessor Locking Protocol) which is a synchronization protocol for multiprocessors. FMLP can be applied to both partitioned and global scheduling algorithms, e.g., P-EDF and G-EDF.

In FMLP, resources are categorized into *short* and *long* resources, and whether a resource is short or long is user specified. There is no limitation on nesting resource accesses, except that requests for long resources cannot be nested in requests for short resources.

Under FMLP, deadlock is prevented by grouping resources. A group includes either global or local resources, and two resources are in the same group if a request for one is nested in a request for the other one. A group lock is assigned to each group and only one task can hold the lock of the group at any time.

The jobs that are blocked on short resources perform busy-wait and are added to a FIFO queue. Jobs that access short resources hold the group lock and execute non-preemptively. A job accessing a long resource holds the group lock and executes preemptively using priority inheritance, i.e., it inherits the highest priority among all jobs blocked on any resource within the group. Tasks blocked on a long resource are added to a FIFO queue.

Under global scheduling, FMLP actually works under a variant of G-EDF for Suspendable and Non-preemptable jobs (GSN-EDF) [9] which guarantees that a job J_i can only be blocked, with a constraint duration, by another non-preemptable job when J_i is released or resumed.

Brandenburg and Anderson in [10] extended partitioned FMLP to the fixed priority scheduling policy and derived a schedulability test for it. Under partitioned FMLP global resources are categorized into long and short resources. Tasks blocked on long resources suspend while tasks blocked on short re-

sources perform busy wait. However, there is no concrete solution how to assign a global resource as long or short and it is assumed to be user defined. In an evaluation of partitioned FMLP [31], the authors differentiate between long FMLP and short FMLP where all global resources are only long and only short respectively. Thus, long FMLP and short FMLP are suspend-based and spin-based synchronization protocols respectively. In both alternatives the tasks accessing a global resource executes non-preemptively and blocked tasks are waiting in a FIFO-based queue.

2.4.4 Parallel PCP (P-PCP)

Easwaran and Andersson proposed a synchronization protocol [14] under the global fixed priority scheduling protocol called Parallel PCP (P-PCP). The authors have derived schedulability analysis for the previously known Priority Inheritance Protocol (PIP) under global scheduling algorithms as well as for P-PCP. For resource sharing under global fixed priority scheduling policies, this is the first work that provides a schedulability test.

Under PIP, while a job J_j accesses a resource, the job's *effective priority* is raised to the highest priority of any job waiting for the resource if there is any, otherwise J_j executes with its base priority. A synchronization protocol may temporarily raise the priority of a job which is called effective priority of the job. Under PIP the priority of a job locking a global resource is not raised unless a higher priority job is waiting for the resource. We call this alternative of PIP as Basic PIP (B-PIP). In [32] we extended the schedulability analysis to Immediate PIP (I-PIP) where the effective priority of a job locking a resource is *immediately* raised to the highest priority of any task that may request the resource.

P-PCP is a generalization of PCP to the global fixed priority scheduling policy. For each task sharing resources, P-PCP offers the possibility of a tradeoff between the interference from lower priority jobs and the amount of parallel executions that can be performed. The tradeoff for each task is adjusted based on an associated tuning parameter, noted by α. A higher value for α of the task means that more lower priority jobs may execute at effective priority higher than the task's base priority thus introducing more interference to the task. However, at the same time a higher value of α will increase the parallelism on a multiprocessor platform.

2.4.5 O(m) Locking Protocol (OMLP)

Brandenburg and Anderson [11] proposed a new suspend-based locking protocol, called OMLP (O(m) Locking Protocol). OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended tasks are assumed to occupy processors and thus blocking is counted as demand. To test the schedulability, the worst-case execution times of tasks are inflated with blocking times. In difference with OMLP, other suspend-based protocols are suspend-aware where suspended tasks are not assumed to occupy their processors. OMLP works under both global and partitioned scheduling. OMLP is *asymptotically optimal*, which means that the total blocking for any task set is a constant factor of blocking that cannot be avoided for some task sets in the worst case. An asymptotically optimal locking protocol however does not mean it can perform better than non-asymptotically optimal protocols.

Under global OMLP, each global resource is associated with two queues in which requesting jobs are enqueued, i.e., a FIFO queue of size m where m is the number of processors and a prioritized queue. Whenever a job requests a resource if its associated FIFO queue is not full the job will be added to the end of the FIFO queue, otherwise it is added to the prioritized queue of the resource. The job at the head of the FIFO queue is granted access to the resource. As soon as the full FIFO gets a free place, i.e., the job at the head of the FIFO queue releases the resource, the highest priority job from the prioritized queue is added to the end of the FIFO queue.

Under partitioned OMLP, each processor has a unique token and any local task requesting any global resource should hold the token to be able to access its requested resource. The tasks requesting global resources are enqueued in a prioritized queue to receive the token. The tasks waiting for a global resource are also enqueued in a global FIFO queue associated with the resource. Any task accessing a global resource cannot be preempted by any task until it releases the resource.

Recently, the same authors extended OMLP to clustered scheduling [33]. In this work, in despite of global and partitioned OMLP where each resource needs two queues (a FIFO and a prioritized), the authors have simplified OMLP under clustered scheduling so that it only needs a FIFO queue for each global resource in order to be asymptotically optimal. To achieve this, instead of priority inheritance and boosting in global and partitioned OMLP respectively, they propose a new concept called priority donation which is an extension of priority boosting. With priority boosting a job can be repeatedly preempted while with priority donation, each job can be preempted at most once. Under

priority donation a higher priority job may suspend and donate its priority to a lower priority job requesting a resource to accomplish accessing the resource.

2.4.6 Multiprocessor Synchronization Protocol for Real-Time Open Systems (MSOS)

We proposed MSOS (Multiprocessor Synchronization protocol for real-time Open Systems) [12] which is a suspend-based synchronization protocol for handling resource sharing among real-time applications in an open system on a multi-core platform. In an open system, applications can enter and exit during run-time. The schedulability analysis of each application is performed in isolation and its demand for global resources is summarized in a set of requirements which can be used for the global scheduling when co-executing with other applications. Validating these requirements is easier than performing the whole schedulability analysis. Thus, an run-time admission control program would perform much better when introducing a new application or changing an existing one.

We refer to the original MSOS as MSOS-FIFO. The protocol assumes that each real-time application is allocated on a dedicated core. Furthermore, MSOS-FIFO assumes that the applications have no assigned priority and thus applications waiting for a global resource are enqueued in an associated global FIFO-based queue. However, in real-time systems assigning priorities often increases the schedulability of systems. We have proposed an alternative of MSOS, called MSOS-Priority [34] to be applicable for prioritized applications when accessing mutually exclusive resources. MSOS-Priority together with an optimal priority assignment algorithm that is proposed in the same paper mostly outperforms any existing suspend-based synchronization protocol and in many cases, e.g., for lower preemption overhead, it even outperforms spin-based protocols as well. More details about MSOS (both MSOS-FIFO and MSOS-Priority) are presented in Chapter 4.

2.5 Assumptions of the Thesis

With respect to the above presented background material, the work presented in this thesis has been developed under the following limitations:

Real-Time Systems:
We assume hard real-time systems.

Multi-core Architecture:
We assume identical multi-core architectures. However, as a future work we believe that this assumption can be relaxed.

Scheduling Protocol:
The different contributions of the thesis focus on different scheduling classes, i.e., partitioned global as well as clustered scheduling approaches.

Synchronization Protocol:
In the partitioning algorithm we have focused on MPCP as the synchronization protocol under which our heuristic attempts to decrease blocking overhead. The major focus of the thesis is the synchronization protocols that we have developed and improved. However, for the experimental evaluations we have considered other existing synchronization protocols, i.e., MPCP, MSRP, FMLP, OMLP, and PIP.

System Model and Related Work:
In the included papers there may be some differences in the terminologies and notions, e.g., in some papers we use real-time applications while in some other papers we have used real-time components. Thus, we have provided different task and platform models throughout the thesis. We have also presented the related work separately, i.e., the work related to each approach is presented previous to the approach in its respective chapter.

Chapter 3

Blocking-aware Algorithms for Partitioning Task Sets on Multiprocessors

In this chapter we present our proposed partitioning algorithm in which a task set is attempted to be efficiently allocated onto a shared memory multi-core platform with identical processors.

3.1 Related Work

A scheduling framework for multi-core processors was presented by Rajagopalan et al. [35]. The framework tries to balance between the abstraction level of the system and the performance of the underlying hardware. The framework groups dependant tasks, which, for example, share data, to improve the performance. The paper presents Related Thread ID (RTID) as a mechanism to help the programmers to identify groups of tasks.

The *grey-box* modeling approach for designing real-time embedded systems was presented in [36]. In the grey-box task model the focus is on task-level abstraction and it targets performance of the processors as well as timing constraints of the system.

In Paper A [16] we have proposed a heuristic blocking-aware algorithm to allocate a task set on a multi-core platform to reduce the blocking overhead of

tasks.

Partitioning (allocation tasks on processors) of a task set on a multiprocessor platform is a bin-packing problem which is known to be a NP-hard problem in the strong sense; therefore finding an optimal solution in polynomial time is not realistic in the general case [37]. Heuristic algorithms have been developed to find near-optimal solutions.

A study of bin-packing algorithms for designing distributed real-time systems was presented in [38]. The presented method partitions a software into modules to be allocated on hardware nodes. In their approach they use two graphs; one graph which models software modules and another graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of required bins (processors) and the required bandwidth for the communication between nodes.

Liu et al. [39] presented a heuristic algorithm for allocating tasks in multicore-based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this thesis) are assigned to processing nodes, and the second round allocates tasks in a process to the cores of a processor. However, the algorithm does not consider synchronization between tasks.

Baruah and Fisher have presented a bin-packing partitioning algorithm, the *first-fit decreasing* (FFD) algorithm [40] for a set of independent sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines, and the algorithm assigns the tasks to the processors in first-fit order. The tasks on each processor are scheduled under uniprocessor EDF.

Lakshmanan et al. [15] investigated and analyzed two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. They have developed a blocking-aware task allocation algorithm, an extension to the *best-fit decreasing* (BFD) algorithm, and evaluated it under both execution control policies. Their blocking-aware algorithm is of great relevance to our proposed algorithm, hence we have presented their algorithm in more details in Section 3.3. Together with our algorithm we have also implemented and evaluated their blocking-aware algorithm and compared the performances of both algorithms.

3.2 Task and Platform Model

Our target system is a task set that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i is the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks share a set of resources, R, which are protected using semaphores. The set of critical sections, in which task τ_i requests resources in R, is denoted by $\{Cs_{i,q,p}\}$, where $Cs_{i,q,p}$ indicates the worst case execution time of the p^{th} critical section of task τ_i, in which the task accesses resource $R_q \in R$. The tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

We have also assumed that the multi-core platform is composed of identical unit-capacity processors with shared memory. The task set is partitioned into partitions $\{P_1, \ldots, P_m\}$, where m represent the number of required processors and each partition is allocated onto one processor.

3.3 Partitioning Algorithms with Resource Sharing

In this section we present our blocking-aware heuristic algorithm to allocate tasks onto the processors of a multi-core platform. The algorithm extends a bin-packing algorithm with synchronization factors. The results of our experimental evaluation [16] shows a significant performance increment compared to the existing similar algorithm [15] and a reference *blocking-agnostic* bin-packing algorithm. The blocking-agnostic algorithm, in the context of this thesis, refers to a bin-packing algorithm that does not consider blocking parameters to increase the performance of partitioning, although blocking times are included in the schedulability test.

In our algorithm task constraints are identified, e.g., dependencies between tasks, timing attributes, and resource sharing preferences, and we extend the best-fit decreasing (BFD) bin-packing algorithm with blocking time parameters. The objective of the heuristic is to decrease the blocking overheads, by assigning tasks to appropriate partitions with respect to the constraints and preferences.

In a blocking-agnostic BFD algorithm, the processors are ordered in non-increasing order of their utilization and tasks are ordered in non-increasing order of their size (utilization). Beginning from the top of the ordered processor

list, the algorithm attempts to allocate the task from the top of the ordered task set onto the first processor that fits it, i.e., the first processor on which the task can be allocated without making any processor unschedulable. If none of the processors can fit the task, a new processor is added to the processor list and the task is allocated to this processor. At each step the schedulability of all processors must be tested, because allocating a task to a processor can increase the remote blocking time of tasks previously allocated to other processors and may make the other processors unschedulable. This means, it is possible that some of the previous processors become unschedulable even if a task is allocated to a new processor.

The algorithm proposed in [15] was called Synchronization-Aware Partitioning Algorithm, and we call our algorithm Blocking-Aware Partitioning Algorithm. Both algorithms have the same objective, i.e., consideration of resource sharing factors during partitioning to decrease the overall blocking overheads. However, to ease refereing them, we refer them as SPA and BPA respectively. Both our algorithm (BPA) and the existing one (SPA) assume that MPCP is used for lock-based synchronization. Thus, we derive heuristics based on the blocking parameters in MPCP. However, our algorithm can be easily extended to other synchronization protocols as well, e.g., MSRP, FMLP and OMLP.

3.3.1 Blocking-Aware Algorithm (BPA)

The algorithm attempts to allocate a task set onto processors in two rounds. The output of the round with better partitioning results will be chosen as the output of the algorithm. In each round the tasks are allocated to the processors in a different way. When a bin-packing algorithm allocates an object (task) to a bin (processor), it usually attempts to put the object in a bin that fits it better, and it does not consider the unallocated objects. The rationale behind the two rounds is that the heuristic tries to consider both the past and the future by looking at tasks allocated in the past and those that are not yet allocated. In the first round the algorithm considers the tasks that are not allocated to any processor yet, and attempts to take as many as possible of the best related tasks with the current task by considering remote blocking parameters. In the second round it considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. In the second round, the algorithm performs more like the usual bin packing algorithms, i.e., it attempts to find the best bin for the current object. Briefly, the algorithm in the first round looks at the future and in the second round it considers the past.

Before starting the two rounds the algorithm performs some basic steps:

- A heuristic weight is assigned to each task which is a function of task's utilization as well as the blocking parameters that lead to potential remote blocking time introduced by other tasks. The heuristic weight for a task τ_i, denoted by w_i, is calculated as follows:

$$w_i = u_i + \left\lceil \left(\sum_{\rho_i < \rho_k} \mathrm{NC}_{i,k} \beta_{i,k} \left\lceil \frac{T_i}{T_k} \right\rceil + \mathrm{NC}_i \max_{\rho_i \geq \rho_k} (\beta_{i,k}) \right) / T_i \right\rceil \quad (3.1)$$

  where, $\mathrm{NC}_{i,k}$ is the number of critical sections of τ_k in which it shares a resource with τ_i and $\beta_{i,k}$ is the longest critical section among them, and NC_i is the total number of critical sections of τ_i.

  Considering the remote blocking terms of MPCP [6], the rationale behind the definition of the weight is that the tasks that have the potential to be punished more by remote blocking become heavier. Thus, they can be allocated earlier and attract as many as possible of the tasks with which they share resources.

- Next, the *macrotasks* are generated. A macrotask is a group of tasks that directly or indirectly share resources, e.g., if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask. A macrotask has two alternatives; it can either be broken or unbroken. A macrotask is set as broken if it cannot fit in one processor, i.e., it cannot be scheduled by a single processor even if no other task is allocated onto the processor, otherwise it is set as unbroken. If a macrotask is unbroken, the partitioning algorithm always allocate all tasks in the macrotask to the same processor. Thus, all resources shared by tasks within the macrotask will be local. However, tasks within a broken macrotask have to be distributed into more than one partition. Similar to tasks, a weight is assigned to each macrotask, which is the summation of the weights of its tasks.

- After generating the macrotasks, the unbroken macrotasks along with the tasks not belonging to any unbroken macrotasks (i.e., the tasks that either do not share any resource or they belong to a broken macrotask) are ordered in a single list in non-increasing order of their weights. We call this list as the *mixed list*.

In the both rounds the strategy of task allocation depends on attraction between tasks. Co-allocation of tasks on the same processor is based on a cost

function which is called *attraction function*. The attraction of task τ_k to a task τ_i is defined based on the potential remote blocking overhead that task τ_k can introduce to task τ_i if they are allocated onto different processors. We represent the attraction of task τ_k to task τ_i as $v_{i,k}$ which is calculated as follows:

$$v_{i,k} = \begin{cases} \mathrm{NC}_{i,k}\beta_{i,k}\left\lceil \frac{T_i}{T_k} \right\rceil & \rho_i < \rho_k; \\ \mathrm{NC}_i\beta_{i,k} & \rho_i \geq \rho_k \end{cases} \tag{3.2}$$

The rationale behind the attraction function is to allocate the tasks which may remotely block a task τ_i to the same processor as τ_i's in the order of remote blocking overhead, as far as possible.

The weight function (Equation 3.1) and attraction function (Equation 3.2) are heuristics to guide the algorithm under MPCP. These functions may differ under other synchronization protocols, e.g., MSRP, which have different remote blocking terms.

After the basic steps the algorithm continues with the rounds:

First Round The following steps are repeated within the first round until all tasks are allocated to processors:

- All processors are ordered in non-increasing order of their size (utilization).
- The object (a task or an unbroken macrotask) at the top of the mixed list is picked to be allocated.

  (i) If the object is a task and it does not belong to any broken macrotask it will be allocated onto the first processor that fits it, beginning from the top of the ordered processor list. If none of the processors can fit the task a new processor is added to the list and the task is allocated onto it.

  (ii) If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them, i.e., all processors can successfully be scheduled. If none of the processors can fit the tasks (i.e., at least one processor becomes unschedulable), they will be allocated onto a new processor.

  (iii) If the object is a task that belongs to a broken macrotask, the algorithm orders the not allocated tasks in the macrotask in non-increasing order of attraction to the task based on Equation 3.2. We denote this list as *attraction list* of the task. The task itself will be on the top of its attraction list. Although creation of an attraction list begins from a task,

in continuation tasks are added to the list that are most attracted to all of the tasks in the list, i.e., the sum of its attraction to the tasks in the list is maximized. The best processor for allocation which is the processor that fits the most tasks from the attraction list is selected, beginning from the top of the list. If none of the existing processors can fit any of the tasks, a new processor is added and as many tasks as possible from the attraction list are allocated to the processor. However, if the new processor cannot fit any task from the attraction list, i.e., at least one of the processors become unschedulable, the first round fails and the algorithm moves to the second round.

Second Round The following steps are repeated until all tasks are allocated to the processors:

- The object at the top of the mixed list is picked.

  (i) If the object is a task and it does not belong to any broken macrotask, this step is performed the same way as in the first round.

  (ii) If the object is an unbroken macrotask, in this step the algorithm performs the same way as in the first round.

  (iii) If the object is a task that belongs to a broken macrotask, the ordered list of processors is a concatenation of two ordered lists of processors. The top list contains the processors that include some tasks from the macrotask of the picked task; this list is ordered in non-increasing order of processors' attraction to the task based on Equation 3.2, i.e., the processor which has the greatest sum of attractions of its tasks to the picked task is the most attracted processor to the task. The second list of processors is the list of the processors that do not contain any task from the macrotask of the picked task and are ordered in non-increasing order of their utilization. The picked task will be allocated onto the first processor from the processor list that will fit it. The task will be allocated to a new processor if none of the existing ones can fit it. The second round of the algorithm fails if allocating the task to the new processor makes at least one of the processors unschedulabe.

If both rounds fail to schedule a task set the algorithm fails. If one of the rounds fails the result will be the output of the other round. Finally, if both rounds succeed to schedule the task set, the one with less processors will be the output of the algorithm.

3.3.2 Synchronization-Aware Algorithm (SPA)

In this section we present the partitioning algorithm originally proposed by Lakshmanan et al. [15].

- Similar to BPA, the macrotasks are generated (in [15], macrotasks are denoted as bundles). A sufficient number of processors that fit the total utilization of the task set, i.e., $\lceil \sum u_i \rceil$, are added.
- The utilization of macrotasks and tasks are considered as their size and all the macrotasks together with all other tasks are ordered in a list in non-increasing order of their utilization. The algorithm attempts to allocate each macrotask onto a processor. Without adding any new processor, all macrotasks and tasks that fit are allocated onto the processors and the macrotasks that cannot fit are put aside. After each allocation, the processors are ordered in their non-increasing order of utilization.
- The remaining macrotasks are ordered in the order of the cost of breaking them. The cost of breaking a macrotask is defined based on the estimated cost (blocking time) introduced into the tasks by transforming a local resource into a global resource, i.e., the tasks sharing the resource are allocated to different processors. The estimated cost of transforming a local resource R_q into a global resource is defined as follows.

$$\mathrm{Cost}(R_q) = \text{Global Overhead} - \text{Local Discount} \tag{3.3}$$

The Global Overhead is calculated as follows.

$$\text{Global Overhead} = Cs_q / \min_{\forall \tau_i}(T_i) \tag{3.4}$$

where Cs_q is the length of the longest critical section accessing R_q.

And the Local Discount is defined as follows.

$$\text{Local Discount} = \max_{\forall \tau_i \text{ accessing } R_q} (Cs_{i,q}/T_i) \tag{3.5}$$

where $Cs_{i,q}$ is the length of the longest critical section of τ_i accessing R_q.

The cost of breaking any macrotask, mTask_k, is calculated as the maximum of blocking overhead caused by transforming its accessed resources into global resources.

$$\mathrm{Cost}(\mathrm{mTask}_k) = \sum_{\forall R_q \text{ accessed by } \mathrm{mTask}_k} \mathrm{Cost}(R_q) \qquad (3.6)$$

- The macrotask with minimum breaking cost is picked and is broken in two pieces such that the size of one piece is as close to the largest utilization available among processors as possible. This means, the tasks within the selected macrotask are ordered in decreasing order of their size (utilization) and the tasks from the ordered list are added to the processor with the largest available utilization as far as possible. In this way, the macrotask has been broken in two pieces; (1) the one including the tasks allocated to the processor and (2) the tasks that could not fit in the processor. If the fitting is not possible a new processor is added and the whole algorithm is repeated again.

The SPA algorithm does not consider any blocking parameters while it allocates the current task to a processor, but only its utilization, i.e., the tasks are ordered in order of their utilization only. The BPA, on the other hand, assigns a heuristic weight (Equation 3.1) which besides the utilization includes the blocking parameters as well. Another issue is that in SPA no relationship based on blocking parameters among individual tasks within a macrotask is considered which as in the BPA could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times. The attraction function in Equation 3.2 facilitates the BPA to allocate the most attracted tasks from the current task's broken macrotask on the same processor. As our experimental results in Paper A show, considering these issues can improve the partitioning significantly.

Chapter 4

Resource Sharing among Real-Time Applications on Multiprocessors

In this chapter we present our work on resource sharing among multiple real-time applications (independently-developed systems) when they co-execute on a shared multi-core platform.

Co-executing of real-time applications on a multi-core platform may have one (or a combination) of the following alternatives: (i) One application is statically allocated on one dedicated processor, (ii) Multiple applications are statically allocated on one dedicated processor, (iii) Each application is distributed over multiple dedicated processors (one cluster).

There are more alternatives which are different from those that we mentioned here. The framework presented by Lipari and Bini [41] and the framework proposed by Shin et al. [28] are examples of those alternatives. In these works a component (application) is allocated on a virtual multiprocessor (virtual cluster) which consists of a set of virtual processors. The virtual processors are allocated on the physical processors (dynamically or statically) and components may share physical processors. However, in this thesis we have only focused on the cases where the components are allocated on dedicated processors/clusters and the components do not share processors.

In Paper B [12] we developed the synchronization protocol MSOS with focus on the first alternative, i.e., one application per processor. The original

MSOS, which we call MSOS-FIFO, assumed no priority among applications on accessing resources. In Paper C [34] we developed a new version of MSOS, called MSOS-Priority which is applicable for prioritized applications. For the second alternative, the well studied techniques for integrating real-time applications on uniprocessors can be reused, e.g., the methods presented in [42] and [17]. These techniques usually abstract the timing requirements of the internal tasks of each application and by using this, each application is abstracted as one (artificial) task, hence from outside of the containing processor there will be one application on the processor. Thus by reusing uniprocessor techniques in this area the second alternative becomes similar to the first alternative. We extended our work to the third alternative, where one application is allocated on one dedicated cluster, in Paper D [43].

Regarding co-executing real-time applications in a shared open environment on a uniprocessor platform, a considerable amount of work has been done. A non-exhaustive list of research in this domain includes [44, 45, 46, 47, 48, 42, 49]. Hierarchical scheduling has been studied and developed as a solution for temporal isolation among real-time applications (components) when they execute on the processor. Most of work in this domain has not considered shared resources among the applications. A non-exhaustive list of work presenting the techniques for resource sharing among real-time applications on uniprocessors includes [17, 18, 19].

Hierarchical scheduling techniques have also been developed for multiprocessors (multi-cores) [27, 28]. However, the systems (called clusters in the mentioned papers) are assumed to be independent and do not allow for sharing of mutually exclusive resources.

Recently, Faggioli et al. proposed a server-based resource reservation protocol for resource sharing called Multiprocessor BandWidth Inheritance protocol (M-BWI) [50] which can be used for open systems on multiprocessors where hard, soft and non real-time systems may co-execute. M-BWI uses a mixture of spin-based and suspend-based approaches for tasks waiting for resources. The underlying scheduling policy is not required to be known. However, M-BWI assumes that the number of processors are known. The implementation of M-BWI seems to be complex as various states for servers have to be preserved during run-time. Furthermore, under M-BWI tasks have to be aware of each other, e.g., to establish the chain of blocks, which may make it difficult to use M-BWI with black box or legacy components.

4.1 The Synchronization Protocol for Real-Time Applications under Partitioned Scheduling

As mentioned above we developed the synchronization protocol MSOS (Multiprocessors Synchronization protocol for real-time Open Systems) for handling resource sharing among independently-developed real-time applications on a shared multi-core platform; MSOS-FIFO and MSOS-Priority for synchronization on mutually exclusive resources shared among non-prioritized and prioritized real-time applications respectively.

4.1.1 Assumptions and Definitions

We assume that one core of the underlying multi-core contains at most one real-time application A_k. Application A_k is represented by an interface I_k which abstracts the information regarding shared resources. Each application may use a different scheduling policy, however in this thesis we concentrate on fixed priority scheduling within applications.

An application A_k consists of a task set denoted by τ_{A_k} which consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time (period) between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its unique priority. The tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. A task, τ_h, has a higher priority than another task, τ_l, if $\rho_h > \rho_l$. The tasks in application A_k share a set of mutually exclusive resources R_{A_k} that are protected using semaphores. The set of shared resources R_{A_k} consists of two subsets of different types of resources; local and global resources. A local resource is only used by tasks of one application while a global resource is shared by tasks from multiple applications. The sets of local and global resources accessed by tasks in application A_k are denoted by $R^L_{A_k}$ and $R^G_{A_k}$ respectively. The set of critical sections, in which task τ_i requests resources in R_{A_k} is denoted by $\{Cs_{i,q,p}\}$, where $Cs_{i,q,p}$ is the worst-case execution time of the p^{th} critical section of task τ_i in which the task locks resource R_q. We denote $Cs_{i,q}$ as the worst-case execution time of the longest critical section in which τ_i requests R_q. We further assume non-nested critical sections.

The above assumptions are valid for both MSOS-FIFO and MSOS-Priority. However, in MSOS-FIFO an application A_k is represented by $A_k(I_k)$ while in MSOS-Priority the application is represented by $A_k(\rho A_k, I_k)$ where ρA_k is the priority of application A_k.

Resource Hold Time (RHT) The RHT of a global resource R_q by task τ_i in application A_k denoted by $\mathrm{RHT}_{q,k,i}$, is the maximum duration of time the global resource R_q can be locked by τ_i, i.e., $\mathrm{RHT}_{q,k,i}$ is the maximum time interval starting from the time instant when τ_i locks R_q and ending at the time instant when τ_i releases R_q. Thus, the resource hold time of a global resource, R_q, by application A_k denoted by $\mathrm{RHT}_{q,k}$, is as follows:

$$RHT_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{4.1}$$

where $\tau_{q,k}$ is the set of tasks in application A_k sharing R_q.

The concept of resource hold times for composing multiple independently-developed real-time applications on uniprocessors has been studied [20, 21]. On a multi-core (multiprocessor) platform we compute resource hold times for global resources in a different way.

Maximum Resource Wait Time For a global resource R_q in application A_k, denoted by $\mathrm{RWT}_{q,k}$, the maximum resource wait time is the worst-case time that any task τ_i within A_k may wait for other applications on R_q whenever τ_i requests R_q.

4.1.2 MSOS-FIFO

Application Interface In MSOS-FIFO an application A_k is represented by an *interface* $I_k(Q_k, Z_k)$ where Q_k represents a set of requirements. When an application A_k is co-executing with other applications on a multi-core platform, it is said to be schedulable if all the requirements in Q_k are satisfied. A requirement in Q_k is a linear inequality which only depends on the maximum resource wait times of one or more global resources, e.g., $2RWT_{1,k} + 3RWT_{3,k} \leq 18$. The requirements of each application are extracted from its schedulability analysis in isolation. Z_k in the interface represents a set; $Z_k = \{\ldots, Z_{q,k}, \ldots\}$, where $Z_{q,k}$, called Maximum Application Locking Time (MALT), represents the maximum duration of time that any task τ_x in any other application A_l $(l \neq k)$ may be blocked by tasks in A_k whenever τ_x requests R_q.

General Description

Access to the local resources is handled by a uniprocessor synchronization protocol, e.g., PCP or SRP. Under MSOS-FIFO each global resource is associated with a global FIFO queue in which applications requesting the resource are enqueued. Within an application the tasks requesting the global resource are enqueued in a local queue; either priority-based or FIFO-based queues. When the resource becomes available to the application at the head of the global FIFO, the eligible task, e.g., at the top of the local FIFO queue, within the application is granted access to the resource.

To decrease interference of applications, they have to release the locked global resources as soon as possible. In other words, the lengths of resource hold times of global resources have to be as short as possible. This means that a task τ_i that is granted access to a global resource R_q should not be delayed by any other task τ_j, unless τ_j holds another global resource. To achieve this, the priority of any task τ_i within an application A_k requesting a global resource R_q is increased immediately to $\rho_i + \rho^{max}(A_k)$, where $\rho^{max}(A_k) = \max\{\rho_i | \tau_i \in \tau_{A_k}\}$. Boosting the priority of τ_i when it is granted access to a global resource will guarantee that τ_i can only be delayed or preempted by higher priority tasks executing within a gcs. Thus, the RHT of a global resource R_q by a task τ_i is computed as follows:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} \tag{4.2}$$

where $H_{i,q,k} = \sum_{\substack{\forall \tau_j \in \tau_{A_k},\, \rho_i < \rho_j \\ \wedge\ R_l \in R^G_{A_k},\, l \neq q}} Cs_{j,l}$.

An application A_l can block another application A_k on a global resource R_q up to $Z_{q,l}$ time units whenever any task within A_k requests R_q. The worst-case waiting time $RWT_{q,k}$ of A_k to wait for R_q whenever any of its tasks requests R_q is calculated as follows:

$$RWT_{q,k} = \sum_{A_l \neq A_k} Z_{q,l} \tag{4.3}$$

In Paper B we have derived the calculation of $Z_{q,k}$ of a global resource R_q for an application A_k, as follows:
- For FIFO-based local queues:

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} RHT_{q,k,i} \tag{4.4}$$

- For Priority-based local queues:

$$Z_{q,k} = |\tau_{q,k}| \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{4.5}$$

where $|\tau_{q,k}|$ is the number of tasks in application A_k sharing R_q.

4.1.3 MSOS-Priority

General Description

The general idea in MSOS-Priority is to manage access to mutually exclusive global resources among prioritized applications. To handle accessing the resources the global queues have to be priority-based. When a global resource becomes available, the highest priority application in the associated global queue is eligible to use the resource. Within an application the tasks requesting a global queue are enqueued in either a priority-based or a FIFO-based local queue. When the highest priority application is granted access to a global resource, the eligible task within the application is granted access to the resource. If multiple requested global resources become available for an application they are accessed in the priority order of their requesting tasks within the application.

It has been shown [51] that cache-related preemption overhead, depending on the working set size (WSS) of jobs, can be significant. WSS of a job is the amount of memory that the job needs during its execution. Thus, performing busy wait in spin-based protocols in some cases may benefit the schedulability as they decrease preemptions comparing to suspend-based protocols. As the results of our experimental evaluations in Paper C show, the larger preemption overheads generally decrease the performance of suspend-based protocols significantly. However, the experiments show that MSOS-Priority almost always outperforms all other suspend-based protocols. Furthermore, in many cases MSOS-Priority performs better than spin-based protocols even if the preemption overhead is relatively high. In this thesis we did not consider the system overhead, e.g., the overhead regarding queue manipulating, which will favor spin-based protocols significantly, and for relatively large amount of system overhead it will be very hard for suspend-based protocols to outperform spin-based protocols. For MSOS-Priority to reach its highest performance with regard to schedulability, an efficient priority assignment algorithm has to be used. Our proposed optimal priority assignment algorithm (Section 4.2) contributes to the efficiency of MSOS-Priority significantly.

Under MSOS-FIFO, a gcs of a lower priority task τ_l can be preempted by a gcs of a higher priority task τ_h if they are accessing different resources. This increases the number of preemptions which adds up the preemption overhead to gcs's and thus making RHT's longer. To avoid this, we modify this rule in MSOS-Priority to reduce preemptions. To achieve this, tasks have to execute non-preemptively while accessing a global resource, i.e., within gcs's. The RHT of a global resource R_q by a task τ_i is computed similar to MSOS-FIFO except that, under MSOS-Priority, at most one gcs from lower priority tasks may further increase the length of RHT:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} + \max_{\substack{\forall \tau_l \in \tau_{A_k},\, \rho_i > \rho_l \\ \wedge\ R_s \in R^G_{A_k},\, s \neq q}} \{Cs_{l,s}\} \tag{4.6}$$

Maximum Application Locking Time (MALT), denoted by $Z_{q,k}(t)$ represents the maximum delay any task τ_x in any other lower priority application A_l may incur from tasks in A_k during time interval t, each time τ_x requests resource R_q.

The maximum execution (workload) of all critical sections of a task τ_j locking R_q during time interval t, denoted by $W_j(t, R_q)$, is computed as follows (more details in Paper C):

$$W_j(t, R_q) = (\lceil \tfrac{t}{T_j} \rceil + 1)\ n^G_{j,q}\ RHT_{q,k,j} \tag{4.7}$$

where $n^G_{j,q}$ is the maximum number of requests of any job of τ_j to R_q.

Using the workload function for one task in Equation 4.7, we can calculate the total maximum workload of all critical sections of all tasks in application A_k in which they use a global resource R_q during time interval t, i.e., $Z_{q,k}(t)$. This is the maximum delay introduced by tasks in A_k to any task requesting R_q in any lower priority application during any time interval t. $Z_{q,k}(t)$ is calculated as follows:

$$Z_{q,k}(t) = \sum_{\tau_j \in \tau_{q,k}} W_j(t, R_q) \tag{4.8}$$

The maximum Resource Wait Time (RWT) for a global resource R_q incurred by task τ_i in application A_k, denoted by $\mathrm{RWT}_{q,k,i}(t)$, is the maximum duration of time that τ_i may wait for the remote applications on resource R_q during any time interval t.

A RWT under MSOS-Priority, considering delays from lower priority applications and higher priority applications can be calculated as follows:

$$RWT_{q,k,i}(t) = \sum_{\rho A_k < \rho A_l} Z_{q,l}(t) + n^G_{i,q} \max_{\rho A_k > \rho A_l} \{RHT_{q,l}\} \tag{4.9}$$

Under MSOS-FIFO, a RWT for a global resource is a constant value which is the same for any task sharing the resource. However, a RWT under MSOS-Priority is a function of time interval t and may differ for different tasks. The RWT for a global resource R_q of a task τ_i in application A_k during the period of τ_i equals to $RWT_{q,k,i}(T_i)$ which covers all delay introduced from both higher priority and lower priority applications sharing R_q:

$$RWT_{q,k,i} = \sum_{\rho A_k < \rho A_l} Z_{q,l}(T_i) + n^G_{i,q} \max_{\rho A_k > \rho A_l} \{RHT_{q,l}\} \tag{4.10}$$

where $RWT_{q,k,i}(T_i)$ is denoted by $RWT_{q,k,i}$.

Application Interface In MSOS-Priority the interface of an application A_k has to contain the requirements that have to be satisfied for A_k to be schedulable. Furthermore, the interface has to provide information required by other applications sharing resources with A_k.

Looking at Equation 4.10, the calculation of the RWT of a task τ_i in application A_k for a global resource R_q requires MALT's, e.g., $Z_{q,h}(t)$, from the higher priority applications as well as RHT's, e.g., $RHT_{q,l}$, from the lower priority applications. This means that to be able to calculate the RWT's, the interfaces of the applications have to provide both RHT's and MALT's for global resources they share. Thus the interface of an application A_k is represented by $I_k(Q_k, Z_k, RHT)$ where Q_k represents a set of requirements, Z_k is a set of MALT's and a MALT is a function of time interval t. MALT's in the interface of application A_k are needed for calculating the total delay introduced by A_k to the lower priority applications sharing resources with A_k. RHT in the interface is a set of RHT's of global resources shared by application A_k. RHT's are needed for calculating the total delay introduced by A_k to the higher priority applications.

4.2 An Optimal Algorithm for Assigning Priorities to Applications

In this section we present our optimal algorithm which assigns unique priorities to the applications. The algorithm only needs information in the interfaces. The algorithm is optimal in the sense that if it fails to assign unique priorities to applications such that all applications become schedulable, any hypothetically optimal algorithm will also fail.

Audsley's Optimal Priority Assignment (OPA) [52] for priority assignment in uniprocessors is the most similar work to our priority assignment algorithm. Davis and Burns [53] showed that OPA can be extended to fixed priority multiprocessor global scheduling if the schedulability of a task does not dependent on priority ordering among higher priority or among lower priority tasks. Our proposed algorithm is a generalization of OPA which can be applicable for assigning priorities to applications based on their requirements. However, our algorithm can perform more efficiently than OPA because the schedulability test that is used by our algorithm is much simpler than that used in [53]. Furthermore, as we will show later in this section, although in the worst case the maximum number of schedulability tests performed by our algorithm is the same as OPA, in some cases our algorithm performs less schedulability tests than OPA.

The pseudo code of the algorithm is shown in Figure 4.1. The algorithm starts by initially assigning the lowest priority (i.e., 0) to all applications. Then the algorithm in different stages tries to increase the priority of applications. In each stage it leaves the priorities of the applications that are schedulable (Line 10) and it increases the priority of the applications that are not schedulable (the for-loop in Line 18). The priority of all unschedulable applications is increased by the number of the schedulable applications in the current stage (Line 19). If the number of applications that become schedulable in the current stage is more than one, their priorities are increased in such a way that each application gets a unique priority; the first application's priority is increased by 0, the second's is increased by 1, the third's is increased by 2, etc (the for-loop in Line 22). When testing the schedulability of an application A_k, the algorithm assumes that all the applications that have the same priority as A_k are higher priority applications. This assumption helps to test whether A_k can tolerate all the remaining applications if they get priority higher than that of A_k. Thus, when calculating RWT's based on Equation 4.10 the algorithm changes condition $\rho A_k < \rho A_l$ in the first term to $\rho A_k \leq \rho A_l$.

```
1  List remainedAppList ← all applications sharing resources;
2  for each application A in remainedAppList
3      A.priority ← 0;
4  end for
5  while (remainedAppList is not empty)
6        List SchedulableApps ← { };
7        List NotSchedulableApps ← { };
8        for each application A in remainedAppList
9            if all requirements of A are satisfied
10                 add A to SchedulableApps;
11           else
12                 add A to NotSchedulableApps;
13           end if
14       end for
15       if SchedulableApps is empty
16             return fail;
17       remainedAppList ← NotSchedulableApps;
18       for each application A in remainedAppList
19           A.priority ← A.priority + (number of applications in SchedulableApps);
20       end for
21       incr ← 0;
22       for each application A in SchedulableApps
23           A.priority ← A.priority + incr;
24           incr ← incr + 1;
25       end for
26 end while
27 return succeed;
```

Figure 4.1: The Priority Assignment Algorithm

Figure 4.2 illustrates an example of the algorithm. In this example, there are four applications sharing resources. The algorithm succeeds to assign priorities to them in three stages. First the algorithm gives the lowest priority to them, i.e., $\rho A_i = 0$ for each application. In this stage the algorithm realizes that applications A_1 and A_3 are schedulable but A_2 and A_4 are not schedulable, thus the priority of A_2 and A_4 are increased by 2 which is the number of schedulable applications, i.e., A_1 and A_3. Both A_1 and A_3 are schedulable, hence to assign unique priorities, the algorithm increases the priority of A_1 and A_3 by 0 and 1 respectively. Please notice that increasing the priority of the schedulable applications can be done in any order since their schedulability has been tested assuming that all the other ones have higher priority. Thus the order in which the priorities of these applications are increased will not make any of them unschedulable. In the second stage, only applications A_2 and A_4

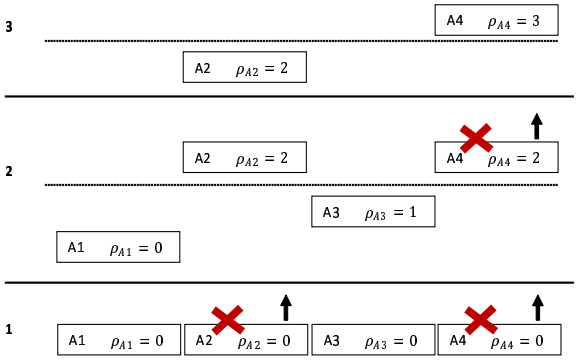


Figure 4.2: Illustrative Example for the Priority Assignment Algorithm

are left. At this stage the algorithm finds that A_4 is not schedulable, hence its priority has to be increased. In the last stage, A_4 also becomes schedulable and since all applications are now schedulable the algorithm succeeds. If at any stage the algorithm cannot find any schedulable application, meaning that none of the remaining applications can tolerate the other ones to have higher priorities, the algorithm fails.

In Audsley's priority assignment algorithm [52] to find a solution (if any) at most $m(m+1)/2$ schedulability tests will be performed where m is the number of tasks to be prioritized. Similarly, in our algorithm to find a solution (if any), in the worst case at each stage only one application is schedulable and is assigned a priority. In the next stage the schedulability of all the remaining applications has to be tested again. In this case, after the algorithm is finished, the schedulability test for the applications with priority $m, m-1, \ldots, 2, 1$ has been performed $m, m-1, \ldots, 2, 1$ times respectively, and hence the maximum number of schedulability tests is $m(m+1)/2$ where m is the number of applications to be prioritized.

However, it may happen that at a stage, x number of applications are schedulable where $x > 1$. In this case the priority of all remaining applications (i.e. applications that are unschedulable at the current stage) will be increased by x (Figure **??**, Line 19 of the algorithm). This means that, the maximum number of schedulability tests for each of the remaining applications would be decreased by x, i.e., the number of stages the algorithm runs is decreased by x. The more similar stages exist the lower the maximum number of schedulability tests will be. As a result the maximum number of stages and consequently

the number schedulability tests are decreased. This is not the case in Audsley's OPA; depending on the order of selecting tasks (or applications), it is still possible that $m(m+1)/2$ schedulability tests would be performed, e.g., OPA finds a solution in exactly m stages. E.g., in the illustrative example in Figure **??**, OPA will assign priorities in 4 stages, and if it selects the applications in order A_4, A_2, A_3, A_1, it will perform 4, 3, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 9 tests will be performed. On the other hand, our algorithm assigns priorities in 3 stages and it performs 3, 2, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 7 tests are performed.

4.3 Synchronization Protocol for Real-Time Applications under Clustered Scheduling

As mentioned in the beginning of this chapter, the third alternative where applications co-execute on a shared multi-core platform is that one application is allocated on a dedicated cluster (multiple cores). We have generalized MSOS to be applicable to this alternative. In this section we present the extended MSOS which we call Clustered MSOS (C-MSOS).

4.3.1 Assumptions and Definitions

We consider a set of real-time components, i.e., real-time applications, aimed to be allocated on the multiprocessor platform. A real-time component consists of a set of real-time tasks. A component may also include components, i.e., hierarchical components, however in this thesis we focus on components composed of tasks only. Each component is allocated on a dedicated subset of processors, called cluster. Each component has its local scheduler which can be any multiprocessor global scheduling algorithm, e.g., G-EDF. The jobs generated by tasks of a component can migrate among its processors, however migration of jobs among clusters is not allowed.

A component C_k consists of a task set denoted by τ_{C_k} which includes n_k sporadic tasks $\tau_i(T_i, E_i, D_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time E_i, relative deadline D_i and ρ_i as its unique base priority. A task τ_i has a higher priority than another task τ_j if $\rho_i > \rho_j$. The set of mutually exclusive resources shared by tasks in component C_k is denoted by R_{C_k}. The set of shared resources R_{C_k} consists of two sets of different types of resources;

local and global resources. The sets of local and global resources accessed by tasks in component C_k are denoted by $R^L_{C_k}$ and $R^G_{C_k}$ respectively. Similar to MSOS-FIFO and MSOS-Priority, the set of critical sections, in which task τ_i requests resources in R_{C_k} is denoted by $\{Cs_{i,q,p}\}$, where $Cs_{i,q,p}$ is the worst case execution time of the p^{th} critical section of task τ_i in which the task uses resource R_q. We define $Cs_{i,q}$ to be the worst case execution time of the longest critical section in which τ_i uses R_q. We also denote $CsT_{i,q}$ as the maximum total amount of time that τ_i uses R_q, i.e., $CsT_{i,q} = \sum Cs_{i,q,p}$. The set of tasks in component C_k sharing R_q is denoted by $\tau_{q,k}$, and $n_{i,q}$ is the total number of critical sections of task τ_i in which it accesses resource R_q. We assume non-nested critical sections. Unlike MSOS-FIFO and MSOS-Priority, we assume constrained-deadline tasks, i.e., $D_i \leq T_i$ for any τ_i.

Component C_k will be allocated on a cluster comprised of m_k processors; $m_k^{(min)} \leq m_k \leq m_k^{(max)}$ where $m_k^{(min)}$ and $m_k^{(max)}$ are the minimum and maximum number of processors required by C_k respectively. In Paper D we have shown how to efficiently determine the number of processors which C_k will be allocated on in the integration phase. In the paper, we have shown that using the information in the interfaces of components the integration of all the real-time components on a multiprocessor platform can be formulated as a Nonlinear Integer Programming (NIP) problem [54]. By formulating the integration phase as a NIP problem, by means of the techniques in this domain [54] it is possible to minimize the total number of required processors on which all components will be schedulable, i.e., their requirements are satisfied.

Resource Hold Time (RHT) of a global resource R_q by task τ_i in component C_k, assuming that C_k is allocated on m_k processors, is denoted by $RHT_{q,k,i}(m_k)$ and is the maximum duration of time that the global resource R_q can be locked by τ_i. Consequently, the resource hold time of a global resource R_q by component C_k, denoted by $RHT_{q,k}(m_k)$, is calculated as follows:

$$RHT_{q,k}(m_k) = \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}(m_k)\} \tag{4.11}$$

Maximum Resource Wait Time (RWT) for a global resource R_q in component C_k, denoted as $RWT_{q,k}$, is the worst-case duration of time that whenever any task τ_i within C_k requests R_q it can be delayed by other components, i.e., R_q is held by tasks from other components.

Component Interface A component C_k is abstracted and represented by an interface denoted by $I_k(Q_k(m_k)\,,\, Z_k(m_k)\,,\, m_k^{(min)}\,,\, m_k^{(max)})$. The index of a component, i.e., k, in the specification of the interface is used to clarify the relationships in the analysis and does not indicate any order among the components, neither does it show that the number of the components is known.

Global resource requirements of C_k are encapsulated in the interface by $Q_k(m_k)$ which is a set of resource requirements that have to be satisfied for C_k to be schedulable on m_k processors. The parameter m_k in $Q_k(m_k)$ indicates that the requirements are dependent on m_k, and hence for different values of m_k the requirements may be different. For C_k to be schedulable on any m_k processors ($m_k^{(min)} \leq m_k \leq m_k^{(max)}$), all requirements in $Q_k(m_k)$ have to be satisfied. Each requirement $r_l(m_k)$ in $Q_k(m_k)$ which depends on m_k, is represented as a linear inequality which indicates that an expression of the maximum resource wait times of one or more global resources should not exceed a value $g_l(m_k)$, e.g., $r_1(m_k) \stackrel{def}{=} 4RWT_{2,k} + 3RWT_{3,k} \leq g_1(m_k)$. Each requirement is extracted from one task requesting at least one global resource. Thus, the number of requirements equals to the number of tasks in component C_k that may request global resources. A formal definition of the requirements is as follows:

$$Q_k(m_k) = \{r_l(m_k)\} \tag{4.12}$$

where

$$r_l(m_k) \stackrel{def}{=} \sum_{\substack{\forall R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \leq g_l(m_k) \tag{4.13}$$

where $\alpha_{i,q}$ is a constant, i.e., it only depends on internal parameters of C_k (more details can be found in Paper D).

The global resource requirements in $Q_k(m_k)$ of a component C_k are extracted from the schedulability analysis of the component in isolation, i.e., to extract the requirements of a component, no information from other possible existing components on the same multi-core platform is required.

$Z_k(m_k)$ in the interface, represents a set $Z_k(m_k) = \{Z_{q,k}(m_k)\}$ where $Z_{q,k}(m_k)$ is the *Maximum Component Locking Time* ($MCLT$). $Z_{q,k}(m_k)$ represents the maximum duration of time that C_k can delay the execution of any task τ_x in any component C_l ($l \neq k$) whenever τ_x requests R_q, i.e., any time any task in C_l requests R_q its execution can be delayed by C_k for at most $Z_{q,k}(m_k)$ time units.

4.3.2 C-MSOS

Under C-MSOS, sharing local resources is handled by multiprocessor PIP. Each global resource is associated with a *global queue* in which components requesting the resource are enqueued. We assume non-prioritized components, hence the global queues can be implemented in either FIFO or Round-Robin manner. Since the resource queues are also shared among tasks and components it may cause contention. We assume that access to queues is performed in an atomic manner, e.g., the index of a FIFO queue has to be an atomic variable. However, we do not consider the overhead regarding contention on resource queues.

Within a component the jobs requesting a global resource are enqueued in a *local queue*. To reduce interference among components and shorten the RHT's, the blocking time on global resources should only depend on the duration of global critical sections. This bounds blocking times on global resources as a function of length and number of global critical sections only. Thus the priority of jobs accessing global resources have to be boosted to be higher than any base priority within the component. The *boosted priority* of any job of task τ_i requesting any global resource equals to $\rho^{max}(C_k) + 1$, where $\rho^{max}(C_k) = \max\{\rho_i | \tau_i \in C_k\}$. Boosting the priority of a job when it executes within a gcs ensures that it can only be delayed by jobs within gcs's. However, boosting the priorities such that they are higher than any priority in the component may cause problem, i.e., make the component unschedulable. We have motivated this problem and proposed a solution for it in Paper E [32] which we have discussed in Section 4.3.3.

4.3.3 Efficient Resource Hold Times

The usual way of decreasing interference among tasks/applications regarding global resources in the existing synchronization protocols under partitioned scheduling has been boosting the priority of a task accessing a global resource to be higher than any base priority of any task that may preempt the task holding the resource. However, although boosting the priorities of tasks holding global resources in this way makes RHT's shorter, it may make a component unschedulable. Thus, to shorten the RHT's the priorities of tasks holding global resources have to be boosted only as far as the application remains schedulable, i.e., boosting the priorities must never compromise the schedulability of an application. On uniprocessor platforms, it has been shown [20, 21] that it is possible to achieve one single optimal solution, when trying to decrease RHT's

within an application. However, in Paper E we have shown that this is not the case when the application is scheduled on multiple processors and there can exist multiple Pareto-optimal solutions.

Considering that the effective priority of a task holding a global resource may not necessarily be high enough to prevent it from being preempted by any task in an application, the RHT's have to be calculated differently. We assume that the priorities of jobs within an application A_k that are granted access to a global resource R_q are boosted to a *boost level* without compromising schedulability of the application. We denote the boost level of R_q in A_k by $boost_{q,k}$, i.e., the priority of any job J_i in A_k that is granted access to R_q is immediately raised to $boost_{q,k}$. With this assumption, we have derived calculation of RHT's:

When a job J_i holds the lock of a global resource R_q and its effective priority is immediately raised to $boost_q$, its execution can be delayed by any other job generated by any other task that belongs to at least one of following three categories:

- The set of tasks with base priority higher than or equal to $boost_q$. We denote $Rh_{q,i}$ as an upper bound for the maximum cumulative execution of the jobs generated by these tasks, while J_i holds the lock of R_q.
- The set of tasks with priorities lower than $boost_q$, whose generated jobs may hold any local resource R_p that satisfy condition $\lceil R_p \rceil \geq boost_q$, where $\lceil R_p \rceil$ is the highest priority of any task that may request R_q. In this case these generated jobs may delay the execution of J_i while J_i holds R_q since their effective priority is at least as high as J_i's boosted priority. The upper bound for the maximum cumulative execution (workload) of these jobs when they hold R_p during the interval that J_i holds R_q is denoted by $Rl_{q,i}$.
- The third category represents the set of tasks with priorities lower than $boost_q$, whose generated jobs hold the lock of any global resource R_l other than R_q with a boost level higher than or equal to R_q's boost level, i.e., $boost_l \geq boost_q$. These jobs holding R_l may delay the execution of J_i while J_i holds R_q because they have a boosted priority at least as high as J_i's boosted priority. The maximum delay from jobs of these tasks is denoted by $Rb_{q,i}$.

When J_i itself uses resource R_q it will hold the resource up to $Cs_{i,q}$ time units, hence the RHT of R_q for τ_i in Application A_k, i.e., $RHT_{q,k,i}$ is calculated as follows:

$$RHT_{q,k,i} = Cs_{i,q} + Rh_{q,i} + Rl_{q,i} + Rb_{q,i} \tag{4.14}$$

4.3.4 Decreasing Resource Hold Times

Considering the task categories shown in Section 4.3.3 that contribute to the calculation of RHT of a global resource R_q, the RHT of R_q in application A_k (i.e., $RHT_{q,k}$) can be decreased by increasing the boost level of R_q (i.e., $boost_q$) as far as A_k remains schedulable.

The goal is to reduce all RHT's of all global resources in an application A_k as far as possible. For uniprocessors, it has been shown that a single optimal solution can be achieved [20, 21]. However, under global scheduling and depending on the order of selecting the resources to increase their boost level, the final solution may differ. In Paper E we have shown that for multiprocessors, e.g., for fixed-priority global scheduling algorithm and PIP as the synchronization protocol, there can exist a set of Pareto-optimal allocations of boosting levels to global resources. In a Pareto-optimal allocation of booting levels, none of the boosting levels can further be increased without decreasing boosting level of any other global resources.

4.3.5 Summary

In this chapter we presented our work on resource sharing among real-time applications (components) on a shared multi-core platform. We have presented our proposed synchronization protocol called MSOS for resource handling among real-time applications where each application is allocated on one processor (core). We originally proposed MSOS for applications with no assigned priorities (called MSOS-FIFO). Later we developed a new version of MSOS called MSOS-Priority which extends MSOS for prioritized applications. We have also proposed an optimal priority assignment algorithm to assign unique priorities to the applications on accessing resources.

We have further extended MSOS to clustered scheduling where each real-time component is allocated on multiple dedicated processors and the tasks within each component are scheduled using a global scheduling. The new protocol which is called C-MSOS has been developed with different queue handling techniques. Finally, we have shown that boosting the priorities of the tasks holding global resources may make their component unschedulable. Thus the priority boosting should not compromise the schedulability of the

component. However, we have shown that there may exist a set of Pareto-optimal solutions when trying to minimize the resource hold times.

The details regarding our proposed protocols and algorithm, their analysis and their experimental evaluations can be found in the respective papers.

Chapter 5

Conclusions

5.1 Summary

In this thesis we have pointed out the increasing interest in multiprocessor methods and techniques as the multi-core architectures are becoming the de-facto processors. We have explained some of the challenges regarding resource management on these platforms. We have briefly discussed the existing scheduling approaches, e.g., partitioned and global scheduling as well as an overview of the existing synchronization protocols for lock-based resource sharing on multiprocessor platforms with real-time properties.

We have proposed a heuristic blocking-aware partitioning algorithm which extends a bin-packing algorithm with synchronization factors. The algorithm allocates a task set onto the processors of an identical unit-capacity multi-core platform. The objective of the algorithm is to decrease the overall blocking times of tasks by means of allocating the tasks that directly or indirectly share resources onto the same processors as far as possible. This generally increases schedulability of a task set and can lead to fewer required processors compared to a blocking-agnostic bin-packing algorithm.

In the thesis, we have also discussed that in industry it is not uncommon to divide large and complex systems into several components (applications) where each of them can be developed independently and in isolation. When these applications are integrated and co-execute on a multi-core platform, a challenge to overcome is how to manage the mutually exclusive resources that these applications may share. We have proposed a synchronization protocol, called MSOS, for resource management among real-time applications when

they co-execute on a multi-core platform with the assumption that each application is allocated on a dedicated core. We have provided the methods to perform the schedulability analysis of each application in isolation where the resource requirements of each application are summarized in an interface. Interface-based global scheduling of MSOS facilitates resource management in open systems where applications can enter and exist during run-time.

The first proposed synchronization protocol MSOS, called MSOS-FIFO, and only supported un-prioritized applications in which applications waiting for locked resources are enqueued in FIFO queues. However, to increase the schedulability of applications we proposed a new version of MSOS, called MSOS-Priority, to support prioritized applications. Under MSOS-Priority, applications are granted access to shared resources based on their priorities. We have proposed an optimal priority assignment algorithm which assigns unique priorities to applications. Our experimental evaluations showed that MSOS-Priority together with the priority assignment algorithm mostly outperform the existing alternatives.

We have further extended MSOS to be applicable for the cases where each application is allocated on a sub-set of cores (cluster). Under the extended MSOS which is called C-MSOS, each application is assigned on multiple cores and hence within an application tasks are scheduled using a global scheduling policy. Finally, we presented how to efficiently extract and calculate the resource hold times of shared resources. To decrease the interference of applications on a shared multi-core platform, resource hold times have to be as short as possible. However, shortening the resource hold times should not compromise the schedulability of an application. We have shown that a set of Pareto-optimal solutions may exist when an application is allocated on multiple cores.

5.2 Future Work

In the future we plan to work further on the resource management issues on multi-core platforms and we will investigate the possibility of improvement of the existing protocols as well as development of new approaches.

One future work will be to extend our partitioning algorithm to other synchronization protocols, e.g., MSRP, FMLP and OMLP, under partitioned scheduling.

In this thesis we have focused on resource management on shared memory multi-cores where resources are protected by semaphores. In a fault-tolerant system, applications have to be protected from other applications that may mal-

function. If the applications are allowed to access shared memory, a malfunctioning application may corrupt parts of the memory that is also shared by other applications. To avoid this, the applications are isolated such that each of them can only access its dedicated portion of memory. However, in this case using resource sharing protocols that rely on shared memory (semaphores) is not feasible. In the future we aim to work on resource management among real-time applications on multi-cores by means of message passing.

Chapter 6

Overview of Papers

6.1 Paper A

Farhang Nemati, Thomas Nolte and Moris Behnam. *Partitioning Real-Time Systems on Multiprocessors with Shared Resources*. In 14th International Conference On Principles Of Distributed Systems (OPODIS'10), pages 253-269, December, 2010.

Summary In this paper we propose a blocking-aware partitioning algorithm which allocates a task set on a multiprocessor (multi-core) platform in a way that the overall amount of blocking times of tasks are decreased. The algorithm reduces the total utilization which, in turn, has the potential to decrease the total number of required processors (cores). In this paper we evaluate our algorithm and compare it with an existing similar algorithm. The comparison criteria includes both number of schedulable systems as well as processor reduction performance.

My contribution I was the main driver in writing the paper and I was responsible for further evaluation of the algorithm. I was also responsible for implementing an algorithm similar to the algorithm proposed in Paper B, and comparing the two algorithms.

6.2 Paper B

Farhang Nemati, Moris Behnam and Thomas Nolte. *Independently-developed Real-Time Systems on Multi-cores with Shared Resources*. In 23rd Euromicro Conference on Real-Time Systems (ECRTS'11), pages 251-261, July, 2011.

Summary In this paper we propose a synchronization protocol for resource sharing among independently-developed real-time systems on multi-core platforms. The systems may use different scheduling policies and they may have their own local priority settings. Each system is allocated on a dedicated processor (core).

In the proposed synchronization protocol, each system is abstracted by an interface which abstracts the information needed for supporting global resources. The protocol facilitates the composability of various real-time systems with different scheduling and priority settings on a multi-core platform.

We have performed experimental evaluations and compared the performance of our proposed protocol (MSOS) against the two existing synchronization protocols MPCP and FMLP. The results show that the new synchronization protocol enables composability without any significant loss of performance. In fact, in most cases the new protocol performs better than at least one of the other two synchronization protocols. Hence, we believe that the proposed protocol is a viable solution for synchronization among independently-developed real-time systems executing on a multi-core platform.

My contribution I was the main driver in writing the paper and I was responsible for further evaluation of the proposed protocol.

6.3 Paper C

Farhang Nemati and Thomas Nolte. *Resource Sharing among Prioritized Real-Time Applications on Multi-cores*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-265/2012-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2012 (submitted to conference).

Summary MSOS (Multiprocessors Synchronization protocol for real-time Open Systems) is a synchronization protocol for handling resource sharing

among independently-developed real-time applications (components) on multi-core platforms. MSOS does not consider any priority setting among applications. To handle resource sharing based on the priority of applications, in this paper we extend MSOS such that it allows for resource sharing among prioritized real-time applications on a multi-core platform. We propose an optimal priority assignment algorithm which assigns unique priorities to the applications based on information in their interfaces. We have performed experimental evaluations to compare the extended MSOS (called MSOS-Priority) to the existing MSOS as well as to the current state of the art locking protocols under multiprocessor partitioned scheduling, i.e., MPCP, MSRP, FMLP and OMLP. The evaluations show that MSOS-Priority mostly performs significantly better than alternative approaches.

My contribution I was the main driver in writing the paper and I was responsible for evaluation of the protocol.

6.4 Paper D

Farhang Nemati and Thomas Nolte. *Resource Sharing among Real-Time Components under Multiprocessor Clustered Scheduling*. Journal of Real-Time Systems (under revision).

Summary In this paper we generalize our previously proposed synchronization protocol (MSOS) for resource sharing among independently-developed real-time applications (components) on multi-core platforms. Each component is statically allocated on a dedicated subset of processors (cluster) whose tasks are scheduled by its own scheduler. In this paper we focus on multiprocessor global fixed priority preemptive scheduling algorithms to be used to schedule the tasks of each component on its cluster. Sharing the local resources is handled by the Priority Inheritance Protocol (PIP). For sharing the global resources (shared across components) we have studied the usage of FIFO and Round-Robin queues for access across the components and the usage of FIFO and prioritized queues within components for handling sharing of these resources. We have derived schedulability analysis for the different alternatives and compared their performance by means of experimental evaluations. Finally, we have formulated the integration phase in the form of a nonlinear integer programming problem whose techniques can be used to minimize the total number of processors required to guarantee the schedulability of all components.

My contribution I was the main driver in writing the paper and I was also responsible for experimental evaluation of the protocol.

6.5 Paper E

Farhang Nemati and Thomas Nolte. *Resource Hold Times under Multiprocessor Static-Priority Global Scheduling*. In 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pages 197-206, August, 2011.

Summary Co-executing independently-developed real-time applications on a shared multiprocessor system, where each application executes on a dedicated subset of processors, requires to overcome the problem of handling mutually exclusive shared resources among those applications. To handle resource sharing, it is important to determine the Resource Hold Time (RHT), i.e., the maximum duration of time that an application locks a shared resource.

In this paper, we study resource hold times under multiprocessor static-priority global scheduling. We present how to compute RHT's for each resource in an application. We also show how to decrease the RHT's without compromising the schedulability of the application. We show that decreasing all RHT's for all shared resources is a multiobjective optimization problem and there can exist multiple Pareto-optimal solutions.

My contribution I was the main driver in writing the paper.

Bibliography

[1] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[2] T. Baker. A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report, 2005.

[3] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[4] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[5] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[6] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[7] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[8] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[9] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[10] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31st IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[12] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of 23th Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 251–261, 2011.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[14] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[15] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[16] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *Proceedings of 14th International Conference on Principles of Distributed Systems (OPODIS'10)*, pages 253–269, 2010.

[17] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *Proceedings of 7th ACM & IEEE International conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[18] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[19] N. Fisher, M. Bertogna, and S. Baruah. The Design of an EDF-Scheduled Resource-Sharing Open Environment. In *Proceedings of 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, 2007.

[20] N. Fisher, M. Bertogna, and S. Baruah. Resource-Locking Durations in EDF-Scheduled Systems. In *Proceedings of 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, 2007.

[21] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of 21st IEEE Parallel and Distributed Processing Symposium (IPDPS'07) Workshops*, pages 1–8, 2007.

[22] J. A. Stankovic and K. Ramamritham, editors. *Tutorial: hard real-time systems*. IEEE Computer Society Press, 1989.

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[24] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Journal of Algorithmica*, 15(6):600–625, 1996.

[25] J. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real-time tasks on multiprocessors. In *Handbook of Scheduling*, 2005.

[26] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proceedings of 22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, pages 3–13, 2010.

[27] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of 19th Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[28] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of 20th Euromicro Conference on Real-time Systems (ECRTS'08)*, pages 181–190, 2008.

[29] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of 18th Euromicro Conference on Real-time Systems (ECRTS'06)*, pages 75–84, 2006.

[30] P. Tsigas and Y. Zhang. Non-blocking Data Sharing in Multiprocessor Real-Time Systems. In *Proceedings of 6th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'99)*, pages 247–254, 1999.

[31] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP , D-PCP , and FMLP on LITMUS. In *Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 105–124, 2008.

[32] F. Nemati and T. Nolte. Resource hold times under multiprocessor static-priority global scheduling. In *Proceedings of 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, pages 197–206, 2011.

[33] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of 11th ACM and IEEE International Conference on Embedded Software (EMSOFT'11)*, pages 69–78, 2011.

[34] F. Nemati and T. Nolte. Resource sharing among prioritized real-time applications on multiprocessors. Technical report, April, 2012.

[35] M. Rajagopalan, B. T. Lewis, and T. A. Anderson. Thread scheduling for multi-core platforms. In *Proceedings of 11th Workshop on Hot Topics in Operating Systems (HotOS'07)*, 2007.

[36] A. Prayati, C. Wong, P. Marchal, F. Catthoor, H. de Man, N. Cossement, R. Lauwereins, D. Verkest, and A. Birbas. Task concurrency management experiment for power-efficient speed-up of embedded mpeg4 im1 player. In *Proceedings of International Conference on Parallel Processing Workshops (ICPPW'00)*, pages 453–460, 2000.

[37] D. S. Johnson. *Near-optimal bin packing algorithms*. Massachusetts Institute of Technology, 1973.

[38] D. de Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *Journal of Embedded Systems*, 2(3-4):196–208, 2006.

[39] Y. Liu, X. Zhang, H. Li, and D. Qian. Allocating tasks in multi-core processor based parallel systems. In *Proceedings of Network and Parallel Computing Workshops, in conjunction with IFIP'07*, pages 748–753, 2007.

[40] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 321–329, 2005.

[41] G. Lipari and E. Bini. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In *Proceedings of 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, 2010.

[42] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[43] F. Nemati and T. Nolte. Resource sharing among real-time components under multiprocessor clustered scheduling. *Real-Time Systems (under revision)*.

[44] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 95–103, 2004.

[45] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of 23th IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[46] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, 2000.

[47] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 99–110, 2005.

[48] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, 2001.

[49] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[50] D. Faggioli, G. Lipari, and T. Cucinotta. The Multiprocessor Bandwidth Inheritance Protocol. In *Proceedings of 22th Euromicro Conference on Real-time Systems (ECRTS'10)*, pages 90–99, 2010.

[51] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. Is semi-partitioned scheduling practical? In *Proceedings of 23rd Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 125–135, 2011.

[52] N.C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start. Technical report, 1991.

[53] R. I. Davis and A. Burns. Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 398–409, 2009.

[54] D. Li and X. Sun. Nonlinear integer programming. Springer, 2006.

II

Included Papers

Chapter 7

Paper A: Partitioning Real-Time Systems on Multiprocessors with Shared Resources

Farhang Nemati, Thomas Nolte and Moris Behnam
In 14th International Conference On Principles Of Distributed Systems (OPODIS'10), pages 253-269, December, 2010.

Abstract

In this paper we propose a blocking-aware partitioning algorithm which allocates a task set on a multiprocessor (multi-core) platform in a way that the overall amount of blocking times of tasks are decreased. The algorithm reduces the total utilization which, in turn, has the potential to decrease the total number of required processors (cores). In this paper we evaluate our algorithm and compare it with an existing similar algorithm. The comparison criteria includes both number of schedulable systems as well as processor reduction performance.

7.1 Introduction

Two main approaches for scheduling real-time systems on multiprocessors exist; global and partitioned scheduling [1, 2, 3, 4]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor. A single global queue is used for storing jobs. A job can be preempted on a processor and resumed on another processor, i.e., migration of tasks among processors is permitted. Under a partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) and EDF. Each processor is associated with a separate ready queue for scheduling task jobs.

Partitioned scheduling protocols have been used more often and are supported (with fixed priority scheduling) widely by commercial real-time operating systems [5], inherent in their simplicity, efficiency and predictability. Besides, the well studied uniprocessor scheduling and synchronization methods can be reused for multiprocessors with fewer changes (or no changes). However, partitioning (allocating tasks to processors) is known to be a bin-packing problem which is a NP-hard problem in the strong sense; hence finding an optimal solution in polynomial time is not realistic in the general case. Thus, to take advantage of the performance offered by multi-cores, scheduling protocols should be coordinated with appropriate partitioning algorithms. Heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been developed to find a near-optimal partitioning [1, 3]. However, the scheduling protocols and existing partitioning algorithms for multiprocessors (multi-cores) mostly assume independent tasks while in real applications, tasks often share resources.

We have developed a heuristic partitioning algorithm [6], under which our system assumptions include presence of mutually exclusive shared resources. The heuristic partitions a system (task set) on an identical shared memory single-chip multiprocessor platform. The objective of the algorithm is to decrease blocking overheads by assigning tasks to appropriate processors (partitions). This consequently increases the schedulability of the system and may reduce the number of processors. Our heuristic identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, and extends the best-fit decreasing (BFD) bin-packing algorithm with blocking time parameters. In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. The Multiprocessor Priority Ceiling Protocol (MPCP) which was proposed by Rajkumar in [7], for many years, has been a stan-

dard multiprocessor synchronization protocol under fixed priority partitioned scheduling. Thus, both our algorithm and an existing similar algorithm proposed in [5] assume that MPCP is used for lock-based synchronization. We have investigated MPCP in more details in [6]. Our algorith, however, can be easily extended to other synchronization protocols under partitioned scheduling policies. The algorithm proposed in [5] is named the Synchronization-Aware Partitioning Algorithm (SPA), and our algorithm is named the Blocking-Aware Partitioning Algorithm (BPA). From now on we refer them as SPA and BPA respectively.

7.1.1 Contributions

The contributions of this paper are threefold:
(1) We propose a blocking-aware heuristic algorithm to allocate tasks onto the processors of a single chip multiprocessor (multi-core) platform. The algorithm extends a bin-packing algorithm with synchronization parameters.
(2) We implement our algorithm together with the best known existing similar heuristic [5]. The implementation is modular in which any new partitioned scheduling and synchronization protocol as well as any new partitioning heuristic can easily be inserted.
(3) We evaluate our algorithm together with the existing heuristic and compare the two approaches to each other as well as to an blocking-agnostic bin-packing partitioning algorithm, used as reference. The blocking-agnostic algorithm, in the context of this paper, refers to a bin-packing algorithm that does not consider blocking parameters to increase the performance of partitioning, although blocking times are included in the schedulability test.

The rest of the paper is as follows: we present the task and platform model in Section 7.2. We explain the existing algorithm (SPA) and present our partitioning algorithms (BPA) in Section 7.3. In Section 7.4 the experimental results of both algorithms are presented and the results are compared to each other as well as to the blocking-agnostic algorithm.

7.1.2 Related Work

A significant amount of work has been done in the domain of task allocation on multiprocessors and distributed systems. The emerging of multi-core architectures has increased the interest in the multiprocessor methods. However, in this paper we present the most related works to our approach.

Tindell et al. [8] describe a method called *simulated annealing* for partitioning a task set on a distributed system. The simulated annealing technique is not a heuristic solution but a global optimization method which is used to find a near-optimal solution. The important factor in simulated annealing is that it includes jumps to new solutions to be able to get a better one. The simulated annealing techniques do not include heuristics and it is usually difficult to find a good or even any feasible partitioning [9].

The *Slack Method* presented in [9] is a partitioning heuristic in which the first step is to divide the tasks into sets of communicating tasks (precedence constraint). The size of each set then is reduced based on the concept of *task slack* which is the delay a task can tolerate without missing its deadline. The second step is to map the sets of tasks onto the processors in a way to reduce the communication among processors.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [10]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes. However, their partitioning method assumes independent tasks.

Baruah and Fisher have presented a bin-packing partitioning algorithm (first-fit decreasing (FFD) algorithm) in [11] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm, however, assumes independent tasks. On the other hand their algorithm has been developed under the EDF scheduling protocol while most existing real-time systems use fixed priority scheduling policies. The focus of our proposed heuristic, in this paper, is fixed priority scheduling protocols, although it can easily be extended to other policies.

Of great relevance to our work presented in this paper is the work presented by Lakshmanan et al. in [5]. In the paper they investigate and analyze two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. They have developed a blocking-aware task allocation algorithm (an extension to BFD) and evaluated it under both execution control policies.

In their partitioning algorithm, the tasks that directly or indirectly share resources are put into what they call bundles (in this paper we call them macro-tasks) and each bundle is tried to be allocated onto a processor. The bundles

that cannot fit into any existing processors are ordered by their cost, which is the blocking overhead that they introduce into the system. Then the bundle with minimum cost is broken and the algorithm is run from the beginning. However, their algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its size (utilization). Furthermore, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle is considered which could help to allocate tasks from a broken bundle to appropriate processors to decrease the blocking times. However, their experimental results show that a blocking-aware bin-packing algorithm for suspend-based execution control policy does not have significant benefits compared to a blocking-agnostic bin-packing algorithm. Firstly, for the comparison, they have only focused on the processor reduction issue; they suppose that the algorithm is better if it reduces the number of processors. They have not considered the worst case as it could be the case that an algorithm fails to schedule a task set. In our experimental evaluation, besides processor reduction, we have considered this issue as well. If an algorithm can schedule some task sets while others fail, we consider it as a benefit. Secondly, in their experiments they have not investigated the effect of some parameters such as the different number of resources, variation in the number and length of critical sections of tasks. By considering these parameters, our experimental results show that in most cases our blocking-aware algorithm has significantly better results than blocking-agnostic algorithms. However, according to our experimental results, their heuristic performs slightly better than the blocking-agnostic algorithm, and our algorithm performs significantly better than both.

In the context of multiprocessor synchronization, Rajkumar et al. for the first time proposed a synchronization protocol in [12] which later [7] was called Distributed Priority Ceiling Protocol (DPCP). DPCP extends PCP to distributed systems and it can be used with shared memory multiprocessors. However, a major motivation of increasing interest in the multiprocessor methods is the emerging of multi-core platforms for which DPCP is not an appropriate synchronization protocol. Rajkumar in [7] presented MPCP, which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Considering that MPCP has been a standard multiprocessor synchronization protocol, our partitioning algorithm attempts to decrease blocking times under MPCP and consequently decrease worst case response times which in turn may reduce the number of needed processors. Gai et al. [13, 14] present MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors. The shared resources are classified as either (i) local resources that

are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. In MSRP, tasks synchronize local resources using SRP [2], and access to global resources is guaranteed a bounded blocking time. Lopez et al. [15] present an implementation of SRP under P-EDF. Devi et al. [16] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is soft real-time systems.

Block et al. [17] present Flexible Multiprocessor Locking Protocol (FMLP), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [18]. However, although in a longer version of [17] [1], the blocking times have been calculated, but to our knowledge there is no concrete schedulability test for FMLP under global scheduling protocols. However, Brandenburg and Anderson in [19] have extended partitioned FMLP to fixed priority scheduling policy and derived a schedulability test for it. In a later work [20], the same authors have compared DPCP, MPCP and FMLP. However, as the partitioned scheduling approaches suffer from bin-packing problem, we believe to achieve a better and fair comparison of the approaches, they should be coordinated with task allocation algorithms.

Recently, Easwaran and Andersson have proposed a synchronization protocol [21] under global fixed priority scheduling protocol. In this paper, for the first time, the authors have derived schedulability analysis of the priority inheritance protocol under global scheduling algorithms.

7.2 Task and Platform Model

In this paper we assume a task set that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ where T_i denotes the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks share a set of resources, R, which are protected using semaphores. The set of critical sections, in which task τ_i requests resources in R is denoted by $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i in which the task locks resource $R_q \in R$. Critical sections of tasks should be sequential or properly nested. The deadline

[1] available at http://www.cs.unc.edu/~anderson/papers/rtcsa07along.pdf

of each job is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

We also assume that the multiprocessor (multi-core) platform is composed of identical, unit-capacity processors (cores) with shared memory. The task set is partitioned into partitions $\{P_1, \ldots, P_m\}$, and each partition is allocated onto one processor (core), thus m represent the minimum number of processors needed.

7.3 The Blocking Aware Partitioning Algorithms

7.3.1 Blocking-Aware Partitioning Algorithm (BPA)

In this section we propose a partitioning algorithm that groups tasks into partitions so that each partition can be allocated and scheduled on one processor. The objective of the algorithm is to decrease the overall blocking times of tasks. This generally increases the schedulability of a task set which may reduce the number of required partitions (processors).

Considering the blocking factors of tasks under MPCP, tasks with more and longer global critical sections lead to more blocking times. This is also shown by experiments presented in [14]. Our goal is to (i) decrease the number of global critical sections by assigning the tasks sharing resources to the same partition as far as possible, (ii) decrease the ratio and time of holding global resources by assigning the tasks that request the resources more often and hold them longer to the same partition as long as possible.

In our previous work [22] we have presented a partitioning framework in which tasks are grouped together based on task preferences and constraints. The framework partitions tasks based on a cost function which is derived from task preferences and constraints. The framework attempts to allocate the tasks that directly or indirectly share resources onto the same processor. Tasks that directly or indirectly share resources are called *macrotasks*, e.g., if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask. However, there are cases that a macrotask cannot fit in one processor (i.e., assuming that the tasks in the macrotask are the only tasks allocated on a processor, still it can not be scheduled by the processor). In this case tasks belonging to the same macrotask can be allocated to different partitions (processors).

The goal of the framework presented in [22] is to put the tasks into appropriate partitions so that the costs are minimized. The framework may have

different partitioning strategies, e.g., increasing cache hits, decreasing blocking times, etc. The strategy of partitioning may differ, depending on the nature of a system, and result in different partitions. The framework is a general partitioning approach without deeply focusing on any specific strategy and thus we have not presented any evaluation except one example. Obviously, for different partitioning strategies (e.g., increasing cache hits) the guiding heuristics as well as the implementation of the algorithm will be completely different. In current work, however, we specifically focus on a partitioning strategy for decreasing remote blocking overheads of tasks which leads to increasing the schedulability of a task set and possibly will reduce the number of processors required for scheduling the task set. We derive heuristics to specifically guide the partitioning algorithm to reduce the remote blocking times. We have also performed detailed experimental evaluation according to different resource sharing parameters.

We have developed a blocking-aware algorithm that is an extension to the BFD algorithm. In a blocking-agnostic BFD algorithm, bins (processors) are ordered in non-increasing order of their utilization and tasks are ordered in non-increasing order of their size (utilization). The algorithm attempts to allocate the task from the top of the ordered task set onto the first processor that fits it (i.e., the first processor on which the task can be allocated while all processors are schedulable), beginning from the top of the ordered processor list. If none of the processors can fit the task, a new processor is added to the processor list. At each step the schedulability of all processors should be tested, because allocating a task to a processor can increase the remote blocking time of tasks previously allocated to other processors and may make the other processors unschedulable. This means, it is possible that some of the previous processors become unschedulable even if a task is allocated to a new processor, which makes the algorithm fail.

The Algorithm: The algorithm performs partitioning of a task set in two rounds and the result will be the output of the round with better partitioning results. However, the algorithm performs a few common steps before starting to perform the rounds. Each round allocates tasks to the processors (partitions) in a different strategy. When a BFD algorithm allocates an object (task) to a bin (processor), it usually puts the object in a bin that fits it better, and it does not consider the unallocated objects that will be allocated after the current object. The rationale behind the two rounds is that the heuristic tries to consider both past and future by looking at tasks allocated in the past and those that are not allocated yet. In the first round the algorithm considers the tasks that are not allocated to any processor yet; and tries to take as many as possible of

the best related tasks (based on remote blocking parameters) with the current task. On the other hand, in the second round it considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. In the second round, the algorithm performs more like the usual bin packing algorithms (i.e., tries to find the best bin for the current object), although it considers the remote blocking parameters while allocating a task to a processor. Any time the algorithm performs schedulability test, for more precise schedulability analysis, it always performs response time analysis [23].
The common steps of the algorithm before the two rounds are performed are as follow:
1. Each task is assigned a weight. The weight of each task, besides its utilization, should depend on parameters that lead to potential remote blocking time caused by other tasks:

$$w_i = u_i + \lceil (\sum_{\rho_i < \rho_k} \mathrm{NC}_{i,k} \beta_{i,k} \lceil \frac{T_i}{T_k} \rceil + \mathrm{NC}_i \max_{\rho_i \geq \rho_k} \beta_{i,k}) / T_i \rceil \qquad (7.1)$$

where, $\mathrm{NC}_{i,k}$ is the number of critical sections of task τ_k in which it shares a resource with τ_i, among these critical sections $\beta_{i,k}$ is the longest one, and NC_i is the total number of critical sections of τ_i.

Considering the remote blocking terms of MPCP [6], the rationale behind the definition of weight is that the tasks that can be punished more by remote blocking become heavier. Thus, they can be allocated earlier and attract as many as possible of the tasks with which they share resources.

2. Macrotasks are generated, i.e., the tasks that directly or indirectly share resources are put into the same macrotask. A macrotask has two alternatives; it can either be broken or unbroken. If a macrotask cannot fit in one processor, (i.e., it is not possible to schedule the macrotask on a single processor even if there is no any other tasks), it is set as broken, otherwise it is denoted as unbroken. Please observe that the test of fitting a macrotask in a single processor (to set it as broken or unbroken) is only done at the beginning. Later on at any time the algorithm tests fitting an unbroken macrotask in a processor, the macrotask may co-exist with other tasks and/or macrotasks on the same processor.

If a macrotask is unbroken, the partitioning algorithm always allocates all tasks in the macrotask to the same partition (processor). This means that all tasks in the macrotask will share resources locally relieving tasks from remote blocking. However, tasks within a broken macrotask will be distributed into more than one partition. Similar to tasks, a weight is assigned to each unbro-

ken macrotask, which equals to the sum of the utilizations (not weights) of its tasks. This is because all the tasks within an unbroken macrotask will always be allocated on the same processor and the tasks will not suffer from any remote blocking, hence there is no need to consider blocking parameters in the weight of an unbroken macrotask.

3. The unbroken macrotasks together with the tasks that do not belong to any unbroken macrotasks are ordered in a single list in non-increasing order of their weights. We denote this list the *mixed list*.

The strategy of allocation of tasks in both rounds depends on attraction between tasks. The attraction function of task τ_k to a task τ_i is defined based on the potential remote blocking overhead that task τ_k can introduce to task τ_i if they are allocated onto different processors. We represent the attraction of task τ_k to task τ_i as $v_{i,k}$ which is defined as follows:

$$v_{i,k} = \begin{cases} \mathrm{NC}_{i,k}\beta_{i,k}\lceil\frac{T_i}{T_k}\rceil & \rho_i < \rho_k; \\ \mathrm{NC}_i\beta_{i,k} & \rho_i \geq \rho_k \end{cases} \tag{7.2}$$

The rationale of the attraction function is to allocate the tasks that may remotely block a task, τ_i, to the same processor as of τ_i (in order of the amount of remote blocking overhead) as far as possible. Please notice, the definition of weight (Equation 7.1) and attraction function (Equation 7.2) are heuristics that guide the algorithm under MPCP. However, these functions may differ under other synchronization protocols, e.g., MSRP and partitioned FMLP, which have different remote blocking terms.

There can be the case in which all tasks sharing resources end up in one macrotask. In this case if the macrotask can fit in one processor, there is no need to use MPCP or any other multiprocessor synchronization protocol, because there will not be any global resources in the system. On the other hand, if the macrotask does not fit in one processor (i.e., should be broken) the algorithm attempts, by using weight (Equation 7.1) and attraction (Equation 7.2) functions to put attracted tasks on the same processor as far as possible which leads to reducing the remote blocking overhead.

Now we present the continuation of the algorithm in two rounds:

First Round: After the common steps the following steps are repeated within the first round until all tasks are allocated to processors (partitions):
1. All processors are ordered in their non-increasing order of utilization.
2. The object at the top of the mixed list is picked. **(i)** If the object is a task, τ_i,

and it does not belong to a broken macrotask (τ_i does not share any resource) τ_i will be allocated onto the first processor that fits it (all tasks on the processor are still schedulable), beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit τ_i a new processor is added to the list and τ_i is allocated onto it. **(ii)** If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits all of them. If none of the processors can fit the macrotask, it (all its tasks) will be allocated onto a new processor.**(iii)** If the object is a task, τ_i, that belongs to a broken macrotask, the algorithm orders the tasks (those that are not allocated yet) within the macrotask in non-increasing order of attraction to τ_i based on equation 7.2. We call this list the *attraction list* of τ_i. Task τ_i itself will be on the top of its attraction list. The best processor for allocation is selected, which is the processor that fits the most tasks from the attraction list, beginning from the top of the list. As many as possible of the tasks from the attraction list are then allocated to the processor. If none of the existing processors can fit any of the tasks, a new processor is added and as many tasks as possible from the attraction list are allocated to the processor. However, if the new processor cannot fit any task from the attraction list, i.e., at least one of the processors become unschedulable, the first round fails and the algorithm moves to the second round and restarts.

Second Round: The following steps are repeated until all tasks are allocated to processors:

1. The object at the top of the mixed list is picked. **(i)** If the object is a task and it does not belong to a broken macrotask, this step is performed the same way as in the first round. **(ii)** If the object is an unbroken macrotask, in this the algorithm performs the same way as in the first round. **(iii)** If the object is a task, τ_i, that belongs to a broken macrotask, the processors are put in a ordered list, denoted as *Plist*. However the processors are put in *Plist* in two steps. First, the processors that include some tasks from τ_i's macrotask are added to *Plist* in non-increasing order of processors' attraction to τ_i (according to equation 7.2), i.e., the processor which has the greatest sum of attractions of its tasks to the picked task (τ_i) is the most attracted processor to τ_i and is added to *Plist* first. Second, the processors that do not contain any task from τ_i's macrotask are added to *Plist* in non-increasing order of their utilization. After the two steps, the processors which contain at least one task from τ_i's macrotask will be located at the top of the ordered list, *Plist*, followed by the processors not containing any task from τ_i's macro task. The rationale behind this is that the algorithm first attempts to allocate τ_i on a processor containing some tasks from τ_i's macro task and if not succeeded then it tries other pro-

cessors. The picked task (τ_i) will be allocated onto the first processor from the processor list (*Plist*) that will fit τ_i. Task τ_i will be allocated to a new processor if none of the existing ones can fit it. And the second round of the algorithm fails if allocating the task to the new processor makes some of the processors unschedulable.

If both rounds fail to schedule a task set the algorithm fails. If one of the rounds fails the result will be the output of the other one. If both rounds succeed to schedule the task set, the one with fewer partitions (processors) will be the output of the algorithm.

7.3.2 Synchronization-Aware Partitioning Algorithm (SPA)

We have implemented the best known existing partitioning algorithm proposed in [5] in our experimental evaluation framework. The implementation of the algorithm required details of the algorithm which were not presented in [5], hence, in this section we present the algorithm in more details.

1. First, the macrotasks are generated. In [5], macrotasks are denoted as bundles. A number of processors (enough processors that fit the total utilization of the task set) are added.

2. The macrotasks together with other tasks are ordered in a list in non-increasing order of their utilization. The algorithm attempts to allocate each macrotask (i.e., allocate all tasks within the macrotask) onto a processor. Without adding any new processor, all macrotasks and tasks that fit are allocated onto the processors. The macrotasks that can not fit are put aside. After any allocation, the processors are ordered in their non-increasing order of utilization.

3. The remaining macrotasks are ordered in the order of the cost of breaking them. The cost of breaking a macrotask is defined based on the estimated cost (blocking overhead) introduced into the tasks by transforming a local resource into a global resource (i.e., the tasks sharing the resource are allocated to different processors). The estimated cost of transforming a local resource R_q into a global resource is calculated as follows:

$$\mathrm{Cost}(R_q) = \text{Global Overhead} - \text{Local Discount} \tag{7.3}$$

The Global Overhead is calculated as follows:

$$\text{Global Overhead} = \max(|Cs_q|) / \min_{\forall \tau_i}\{\rho_i\} \tag{7.4}$$

where $\max(|Cs_q|)$ is the length of longest critical section accessing R_q.

The Local Discount is defined as follows:

$$\text{Local Discount} = \max_{\forall \tau_i \text{ accessing } R_q} (\max(|Cs_{i,q}|)/\rho_i) \tag{7.5}$$

where $\max(|Cs_{i,q}|)$ is the length of longest critical section of τ_i accessing R_q.

The cost of breaking any macrotask, mTask_k, is calculated as the summation of blocking overhead caused by transforming its accessed resources into global resources.

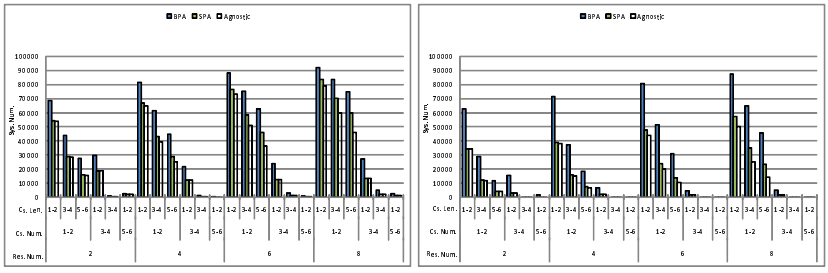
$$\mathrm{Cost}(\mathrm{mTask}_k) = \sum_{\forall R_q \text{ accessed by } \mathrm{mTask}_k} \mathrm{Cost}(R_q) \tag{7.6}$$

4. The macrotask with minimum breaking cost is picked and is broken in two pieces such that the size of one piece is as close as the largest utilization available among processors. This means, tasks within the selected macrotask are ordered in decreasing order of their size (utilization) and the tasks from the ordered list are added to the processor with the largest available utilization as far as possible. In this way, the macrotask has been broken in two pieces; (i) the one including the tasks allocated to the processor and (ii) the tasks that could not fit in the processor. If the fitting is not possible a new processor is added and the whole algorithm is repeated again.

Firstly, as one can see, the SPA algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its utilization, i.e. the tasks are ordered in order of their utilization only. However, our algorithm assigns a weight (Equation 7.1) which besides the utilization includes the blocking terms as well. Secondly, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle (macrotask) is considered which could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times. In our heuristic, we have defined an attraction function (Equation 7.2), which attempts to allocate the most attracted tasks from the current task's broken macrotask, on a processor. As the experimental evaluation in Section 7.4 shows, considering these issues can improve the partitioning significantly.

7.4 Experimental Evaluation and Comparison of Algorithms

In this section we present our experimental results of our blocking-aware bin-packing algorithm (BPA) together with the blocking-aware algorithm recently proposed in [5] (SPA), as well as the reference blocking-agnostic algorithm. For a number of systems (task sets), we have compared the performance of the algorithms in two different aspects; (1) Given a number of systems, the total number of systems that each of the algorithms can schedule, (2) The processor reduction aspect of algorithms.



(a) Workload: 3 processors, 3 tasks per processor

(b) Workload: 3 processors, 6 tasks per processor

(c) Workload: 3 processors, 9 tasks per processor

(d) Workload: 6 processors, 6 tasks per processor

Figure 7.1: Total number of task sets each algorithm schedules.

7.4.1 Experiment Setup

We generated systems (task sets) for different workloads; we denote workload as a defined number of fully utilized processors, e.g., the workload equal to 3 fully utilized processors means the summation of utilizations of all tasks in the system equals to 3. Please notice that the definition of the workload as a number of processors is only to show the total utilization of the task set and it is not the same as the number of required processors (which may be more than the workload) to schedule the task set. Given a workload, the full capacity of each processor (utilization of 1) is randomly divided among a defined number of tasks. Usually for generating systems, utilization and periods are randomly assigned to tasks, and worst case execution times of tasks are calculated based on them. However, in our system generation, the worst case execution times (WCET) of tasks are randomly assigned and the period of each task is calculated based on its utilization and WCET. The reason is that we had to restrict that the WCET of a task not to be less than the total length of its critical sections. Since we have limited the maximum number of critical sections to 6 and the maximum length of any critical section to 6 time units, hence the WCET of each task is greater than 36 (6×6) time units. The WCET of each task was randomly chosen between 36 and 150 time units. The system generation was based on different settings; the input parameters for settings are as follows:

1. Workload (3, 4, 6, or 8 fully utilized processors).

2. The number of tasks per processor (3, 6 or 9 tasks per processor), e.g., 3 tasks per processor means that the utilization of one processor (utilization $= 1$) is randomly distributed among 3 tasks.

3. The number of resources (2, 4, 6, or 8). For each alternative, the resource accessed by each critical section is randomly chosen among the resources, e.g, given the alternative with 2 resources (R_1 and R_2), the resource accessed by any critical section is randomly chosen from $\{R_1, R_2\}$.

4. The range of the number of critical sections per task (1 to 2, 3 to 4 or 5 to 6 critical sections per task). For an alternative (e.g., 1 to 2 critical sections per task), the number of critical sections of any task τ_i is randomly chosen from $\{1, 2\}$.

5. The range of length of critical sections (1 to 2, 3 to 4, or 5 to 6). The length of each critical section is chosen the same way as the number of critical sections per task.

For each setting, we generated 100.000 systems, and combining the param-

eters of settings, i.e., (workloads)×(tasks per processor)×(resources)×(critical sections per task)×(critical section lengths)$= 4 \times 3 \times 4 \times 3 \times 3 = 432$ different settings, total number of systems generated for the experiment were $43.200.000$.

With the generated systems we were able to evaluate the partitioning algorithms with respect to different factors, i.e., various workloads (number of fully utilized processors), number of tasks per processor, number of shared resources, number of critical sections per task, and length of critical sections.

7.4.2 Results

In this section we present the evaluation results of our proposed blocking-aware algorithm (BPA), an existing blocking-aware algorithm [5] (SPA) and the blocking-agnostic algorithm.

The first aspect of comparison of the results from the algorithms is, given a number of systems, the total number of systems each algorithm successfully schedules (Figure 7.1). Figures 7.1(a), 7.1(b) and 7.1(c) represent the results for 3, 6 and 9 tasks per processor respectively. The vertical axis shows the total number of systems that the algorithms could schedule successfully. The horizontal axis shows three factors in three different lines; the bottom line shows the number of shared resources within systems (Res. Num.), the second line shows the number of critical sections per task (Cs. Num.), and the top line represents the length of critical sections within each task (Cs. Len.), e.g., Res. Num.=4, Cs. Num.=1-2, and Cs. Len.=1-2 represents the systems that share 4 resources, the number of critical sections per each task are between 1 and 2, and the length of these critical sections are between 1 and 2 time units. For some settings the number of schedulable systems were too few to be shown on the graphs, thus we omitted these settings from the graphs, e.g., The results for the combination of the number of critical sections = 3-4 and the length of critical sections = 5-6 are not shown in Figure 7.1.

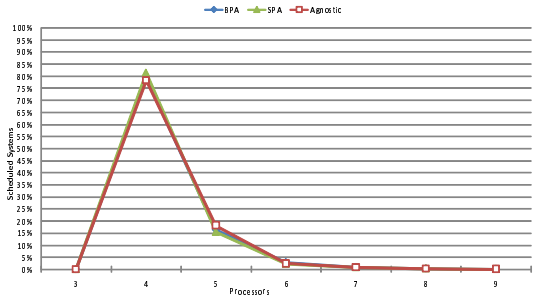
As depicted in Figure 7.1, considering the total number of systems that each algorithm succeeds to schedule, our blocking-aware algorithm (BPA) performs better (can schedule more systems) compared to the SPA and the blocking-agnostic algorithm. However the SPA performs better than the blocking-agnostic algorithm. As shown in the figure, by increasing the number of resources, the number of successfully scheduled systems in all algorithms is increased. The reason for this behavior is that with fewer resources, more tasks share the same resource introducing more blocking overheads which leads to fewer schedulable systems. However, it is illustrated that the blocking-aware algo-

rithms perform better as the number of resources is increased. It is also shown that increasing the number and/or the length of critical sections generally reduces the number of schedulable systems significantly. The reason is that more and longer critical sections introduce greater blocking overhead into the tasks making fewer systems schedulable.

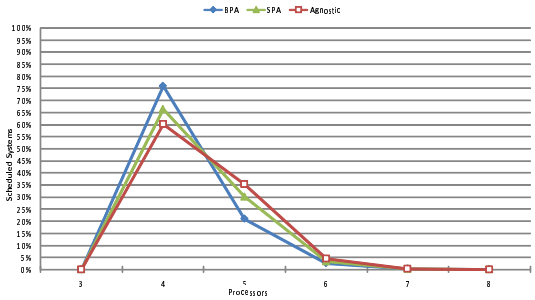
As the number of tasks per processor is increased from 3 (Figures 7.1(a)) to 6 (Figures 7.1(b)) and to 9 (Figures 7.1(c)), the BPA performs significantly better (i.e., schedules significantly more systems) than the SPA and blocking-agnostic bin-packing. However, as one can see, the SPA does not perform significantly better than the blocking-agnostic algorithm as the number of tasks per processor are increased. Increasing the number of tasks per processor lead to smaller tasks (tasks with smaller u_i). The BPA allocates tasks from a broken macrotask based on Equations 7.1 and 7.2, which are functions of the blocking parameters (the number and length of critical sections) as well as the size of the tasks. On the other hand, with the smaller size of tasks, the blocking parameters have a bigger role in these functions, hence more dependent tasks are allocated to the same processor. This leads to less blocking overhead and increased schedulability, hence more systems are scheduled by BPA as the tasks per processor are increased. On the other hand, in SPA, allocation of tasks from a broken macrotask is only based on their utilization, and this does not necessarily allocates highly dependent tasks to the same processor.

As the workload (the number of fully utilized processors) is increased, although the BPA still performs better than the SPA and the blocking-agnostic algorithm, generally the number of schedulable systems by all algorithms is significantly reduced (Figure 7.1(d)). The reason for this behavior is that the number of tasks within systems are relatively many (36 tasks per each system in Figure 7.1(d)) and the workload is high (6 fully utilized processors), and all the tasks within systems share resources. On the other hand, the MPCP is pessimistic. This introduces a lot of interdependencies among tasks and consequently a huge amount of blocking overheads, making fewer systems schedulable. In practice in big systems with many tasks, not all of the tasks share resources, which leads to fewer interdependencies among tasks and less blocking times. However, we continued the experiment with higher workload in the same way as the other experiments (that all tasks share resources) to be able to compare the results with the previous results. We believe that realistic systems, even with high workload and many tasks can benefit from our partitioning algorithm to increase the performance.

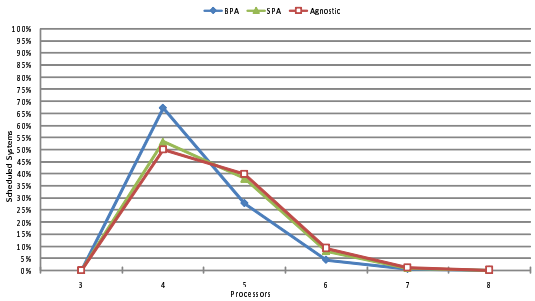
The second aspect for comparison of performance of the algorithms is the processor reduction aspect. To show this, for each algorithm, we ordered the



(a) 3 tasks per processor



(b) 6 tasks per processor



(c) 9 tasks per processor

Figure 7.2: Percentage of systems each algorithm schedules, ordered by required number of processors.

total schedulable systems in order of the number of required processors. Figure 7.2 illustrates the results for the workload of 3 fully packed processors and different number of tasks (3, 6 and 9) per processor. For each algorithm, the schedulable systems by each number of processors are shown as percentage of the total scheduled systems by that algorithm. As the results show, for 3 tasks per processor all three algorithms perform almost the same (Figure 7.2(a)), i.e., each algorithm schedules around 80% of its schedulable systems by 4 processors, 15% to 18% by 5 processors and less than 3% by 6 processors, etc. The reason is that the tasks are large (the utilization of a processor is distributed among 3 task), thus the blocking-aware algorithms do not have much possibility to increase the performance. However as the number of tasks per processor is increased (Figures 7.2(b) and 7.2(c) for 6 and 9 tasks per processor respectively), the blocking-aware algorithms, generally, perform better in processor reduction aspect. Especially the BPA, performs significantly better than the the SPA and the blocking-agnostic algorithm. This means that BPA reduces the required number of processors compared to SPA and the blocking-agnostic algorithm, e.g., as shown in Figure 7.2(c), 68% and 28% of the systems scheduled by BPA require 4 and 5 processors respectively, while 54% and 37% of systems scheduled by SPA can be scheduled by 4 and 5 processors respectively. This means a bigger part (68%) of systems scheduled by BPA require only 4 processors while with SPA this number is smaller (54%).

7.5 Conclusion

In this paper we have proposed a heuristic blocking-aware algorithm, for identical unit-capacity multiprocessor systems, which extends a bin-packing algorithm with synchronization parameters. The algorithm allocates a task set onto the processors of a single-chip multiprocessor (multi-core) with shared memory. The objective of the algorithm is to decrease blocking times of tasks by means of allocating the tasks that directly or indirectly share resources onto appropriate processors. This generally increases schedulability of a task set and may lead to fewer required processors compared to blocking-agnostic bin-packing algorithms. We have also presented and implemented an existing similar blocking-aware algorithm originally proposed in [5].

Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, a standard synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. Another reason to implement our algorithm under MPCP was to be able to

compare our approach to the existing similar approach [5] which has also been developed under MPCP. However, our approach is not limited to MPCP and it can easily be extended to other synchronization protocols such as MSRP and partitioned FMLP.

Our experimental results confirm that our algorithm mostly performs significantly better than the blocking-agnostic as well as the existing heuristic with respect to the number of schedulable systems and the number of required processors. However, given a NP-hard problem, a bin-packing algorithm may not achieve the optimal solution, i.e, there can exist systems that only one of the algorithms can schedule. Thus using a combination of heuristics improves the results with respect to the total number of schedulable systems and processor reduction.

A future work will be extending our partitioning algorithm to other synchronization protocols, e.g., MSRP and FMLP for partitioned scheduling. A very interesting future work is to apply our approach to different synchronization protocols and investigate the effect of bin-packing on those protocols and compare the improvement in their performance. Another interesting future work is to apply our approach to real systems and study the performance gained by the algorithm on these systems. In the domain of multiprocessor scheduling and synchronization our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Acknowledgments

The authors wish to thank Karthik Lakshmanan for fruitful discussions, helping out in improving the quality of this paper.

Bibliography

[1] T. Baker. A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report, 2005.

[2] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[3] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[4] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[5] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[6] F. Nemati, T. Nolte, and M. Behnam. Blocking-aware partitioning for multiprocessors. Technical report, Mälardalen Real-Time research Centre (MRTC), Mälardalen University, March 2010. Available at http://www.mrtc.mdh.se/publications/2137.pdf.

[7] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[8] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: An NP-hard problem made easy. *Journal of Real-Time Systems*, 4(2):145–165, 1992.

[9] P. Altenbernd and H. Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Journal of Real-Time Systems*, 15(2):103–130, 1998.

[10] D. de Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *Journal of Embedded Systems*, 2(3-4):196–208, 2006.

[11] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 321–329, 2005.

[12] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of 9th IEEE Real-Time Systems Symposium (RTSS'88)*, 1988.

[13] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[14] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[15] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[16] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of 18th Euromicro Conference on Real-time Systems (ECRTS'06)*, pages 75–84, 2006.

[17] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[18] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 342–353, 2008.

[19] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[20] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP , D-PCP , and FMLP on LITMUS. In *Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 105–124, 2008.

[21] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[22] F. Nemati, M. Behnam, and T. Nolte. Efficiently migrating real-time systems to multi-cores. In *Proceedings of 14th IEEE Conference on Emerging Techonologies and Factory (ETFA'09)*, 2009.

[23] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In *Principles of Real-Time Systems*, pages 225–248. Prentice Hall, 1994.

Chapter 8

Paper B: Independently-developed Real-Time Systems on Multi-cores with Shared Resources

Farhang Nemati, Moris Behnam and Thomas Nolte
In 23rd Euromicro Conference on Real-Time Systems (ECRTS'11), pages 251-261, July, 2011.

Abstract

In this paper we propose a synchronization protocol for resource sharing among independently-developed real-time systems on multi-core platforms. The systems may use different scheduling policies and they may have their own local priority settings. Each system is allocated on a dedicated processor (core).

In the proposed synchronization protocol, each system is abstracted by an interface which abstracts the information needed for supporting global resources. The protocol facilitates the composability of various real-time systems with different scheduling and priority settings on a multi-core platform.

We have performed experimental evaluations and compared the performance of our proposed protocol (MSOS) against the two existing synchronization protocols MPCP and FMLP. The results show that the new synchronization protocol enables composability without any significant loss of performance. In fact, in most cases the new protocol performs better than at least one of the other two synchronization protocols. Hence, we believe that the proposed protocol is a viable solution for synchronization among independently-developed real-time systems executing on a multi-core platform.

8.1 Introduction

The availability of multi-core platforms has attracted a lot of attention in multiprocessor embedded software analysis and runtime policies, protocols and techniques. As the multi-core platforms are to be the defacto processors, the industry must cope with a potential migration towards multi-core platforms.

An important issue for industry when it comes to migration to multi-cores is the *existing* systems. When migrating to multi-cores it should be possible that several of these systems co-execute on a shared multi-core platform. The (often independently-developed) systems may have been developed with different techniques, e.g., several real-time systems that will co-execute on a multi-core may have different scheduling policies. However, when the systems co-execute on the same multi-core platform they may share resources that require mutual exclusive access. Two challenges to overcome when migrating existing systems to multi-cores are how to migrate the independently-developed systems with minor changes, and how to abstract systems sufficiently, such that the developer of one system does not need to be aware of particular techniques used in other systems.

On the other hand, looking at industrial systems, to speed up their development, it is not uncommon that large and complex systems are divided into several semi-independent subsystems each of which is developed independently. The subsystems which may share resources will eventually be integrated and coexist on the same platform. This issue has got attention and has been studied in the uniprocessor domain [1, 2, 3]. However, new techniques are sought for scheduling semi-independent subsystems on multi-cores.

Looking at current state-of-the-art, two main approaches for scheduling real-time systems on multiprocessors (multi-cores) exist; global and partitioned scheduling [4, 5, 6]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor, i.e., migration of tasks among processors is permitted. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF). Partitioned scheduling policies have been used more often in industry and are they supported widely by commercial real-time operating systems [7], inherent in their simplicity, efficiency and predictability.

The work presented in this paper is inspired by our initial ideas presented in [8], where we focus on the partitioned scheduling policy and synchronization protocols. Allocation (partitioning) of independently-developed systems on a

multi-core architecture may have the following alternatives: (i) one processor includes only one system, (ii) one processor may contain several systems, (iii) a system may be distributed over more than one processor.

In this paper, we focus on the first alternative in which each a system is allocated on a dedicated processor (core). For the second alternative, the well studied techniques for integrating independently-developed systems on uniprocessors can be used, e.g., the methods presented in [9] and [1]. These techniques usually abstract the timing requirements of the internal tasks of each system and using this each system is abstracted as one (artificial) task, hence from outside of the containing processor there will be one system (task set) on the processor. Thus by reusing uniprocessor techniques in this area the second alternative becomes similar to the first alternative. However, extension to the third alternative remains as a future work.

8.1.1 Contributions

The contributions of this paper are as follows:

- We propose a *synchronization protocol* for resource sharing among independently-developed real-time systems (open real-time systems) on a multi-core platform, each of which is allocated on a dedicated core. We have named the protocol as Multiprocessors Synchronization protocol for real-time Open Systems (MSOS).

- We derive an *interface-based schedulability condition* for MSOS. The interface abstracts the global resource sharing of a system in one processor through a set of requirements that should be satisfied to guarantee the schedulability of the system in the processor. A global resource is a resource that is shared across processors. A requirement is a function of resource maximum wait times of global resources (i.e., the worst-case time that a processor may wait for a global resource to be available) which should not exceed a certain value. Thus, the requirements in the interface only depend on the maximum wait times of global resources. Hence we do not need any information from other processors, e.g., scheduling protocol or priority setting policy on other processors, in calculating the interface of a processor.

- We have *evaluated the performance* of MSOS by means of experimental evaluation. In the experiments we compared MSOS against MPCP and FMLP. The obtained results show that the composability offered by

MSOS does not introduce any significant loss of performance and in most cases it even performs better than at least one of the two other protocols. Thus we believe MSOS can be an appropriate synchronization protocol for handling resource sharing among independently-developed systems on a multi-core platform.

8.1.2 Related Work

In the context of independently-developed real-time systems in a shared open environment on uniprocessors, a considerable amount of work has been done. A non-exhaustive list of works in this domain includes [10, 11, 12, 13, 14, 9, 15]. Hierarchical scheduling has been studied and developed as a solution for these systems.

Hierarchical scheduling techniques have also been developed for multiprocessors (multi-cores) [16, 17]. However, the systems (called clusters in the mentioned papers) are assumed to be independent and do not allow for sharing of mutually exclusive resources.

In the context of the synchronization protocols, PCP (Priority Ceiling Protocol) [18] and SRP (Stack-based Resource allocation Protocol) [19] are two of the best known methods for synchronization in uniprocessor systems.

For multiprocessor synchronization, Rajkumar et al. proposed a synchronization protocol [20] which later was called Distributed Priority Ceiling Protocol (DPCP) [21]. DPCP extends PCP to distributed systems and it can be used with shared memory multiprocessors. Rajkumar presented MPCP [21], which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Lakshmanan et al. [7] investigated and analyzed two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. However, MPCP can be used for one single system whose tasks are distributed on different processors. Furthermore for schedulability analysis of each processor, detailed information of tasks allocated on other processors (e.g., priority, the number of global critical section, etc) may be required. Under MSOS the schedulability test of a system on a processor is represented as requirements in its interface which can be obtained without any information from other systems (even before these systems are developed) which will be allocated on other processor.

Gai et al. [22, 23] presented MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors and is an extension of SRP to multiprocessors. Lopez et al. [24] presented

an implementation of SRP under P-EDF. Devi et al. [25] presented a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [26] presented Flexible Multiprocessor Locking Protocol (FMLP) which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [27]. Brandenburg and Anderson in [28] have extended partitioned FMLP to the fixed priority scheduling policy and derived a schedulability test for it. In a later work [29], the same authors compared DPCP, MPCP and FMLP.

Easwaran and Andersson proposed a synchronization protocol [30] under the global fixed priority scheduling protocol. In this paper, the authors have derived schedulability analysis of the Priority Inheritance Protocol (PIP) under global scheduling algorithms.

Recently, Brandenburg and Anderson [31] presented a new suspension-based locking protocol, called O(m) Locking Protocol (OMLP), which has variations for both global and partitioned scheduling. The OMLP (both variations) is *asymptotically optimal*, which means that the total blocking for any task set is a constant factor of blocking that cannot be avoided for some task sets (in the worst case). An asymptotically optimal locking protocol however does not mean it can perform better than non-asymptotically optimal protocols. On the other hand, OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended jobs are assumed to occupy processors and thus blocking is counted as demand. To test the schedulability, the worst-case execution times of tasks are inflated with blocking times. This means that blocking time of any task is introduced to all lower priority tasks. In this paper we focus on *suspension-aware* locking synchronization in which suspended jobs are not assumed to occupy processors. In addition Brandenburg and Anderson have also proposed an asymptotically optimal suspension-aware protocol in [31], called Simple Partitioned FIFO Locking Protocol (SPFP) (under partitioned scheduling), however it uses one single global FIFO queue in which all requests to all global resources are enqueued. However, the drawback of using one single FIFO queue is that it prevents parallelization in accessing resources.

In all the aforementioned existing synchronization protocols (under partitioned scheduling) on multi-cores it is assumed that the tasks of a system are distributed among processors and all processors use the same scheduling policy, e.g., EDF or RM. Furthermore, in the schedulability analysis of the existing

protocols (e.g., MPCP and FMLP) a processor needs timing attributes of tasks allocated on other processors that share resources with its tasks. MSOS, however, allows each system in a processor to use its own scheduling policy and it abstracts the timing requirements regarding global resources shared by the system in its interface, hence, it is not required to reveal its task attributes to other processors which it shares resources with. Recently, in industry, co-existing of several separated systems on a multi-core platform (called virtualization) has been considered to reduce the hardware costs [32]. MSOS seems to be a natural fit for synchronization under virtualization of real-time systems on multi-cores.

8.2 Task and Platform Model

In this paper, we assume that the multiprocessor (multi-core) platform is composed of identical, unit-capacity processors (cores) with shared memory. Each processor contains a different task set (system). The scheduling techniques used on each processor may differ from other processors, e.g., one processor can be scheduled by fixed priority scheduling (e.g., RM) while another processor is scheduled by dynamic priority scheduling (e.g., EDF), which means the priority of tasks are local to each processor. However, for the sake of presentation clarity, in this paper we focus on schedulability analysis of processors with fixed priority scheduling. A task set allocated on a processor, P_k, is denoted by τ_{P_k} and consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. The tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. A task, τ_h, has a higher priority than another task, τ_l, if $\rho_h > \rho_l$. For the sake of simplifying presentation we assume that each task has a unique priority. The tasks on processor P_k share a set of resources, R_{P_k}, which are protected using semaphores. The set of shared resources (R_{P_k}) consists of two subsets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks on the same processor while a global resource is shared by tasks on more than one processor. The sets of local and global resources accessed by tasks on processor P_k are denoted by $R^L_{P_k}$ and $R^G_{P_k}$ respectively. The set of critical sections, in which task τ_i requests resources in R_{P_k} is denoted by $\{Cs_{i,p,q}\}$, where $Cs_{i,q,p}$ is the worst-case execution time of the p^{th} critical section of task τ_i in which the task locks resource $R_q \in R_{P_k}$. We denote $C_{i,q}$ as the worst-case execution time of the longest critical section in which τ_i requests R_q. In this paper, we focus on non-nested critical sections (the common case). A job of task τ_i, is specified by J_i.

8.3 The Multiprocessors Synchronization Protocol for Real-time Open Systems (MSOS)

8.3.1 Assumptions and terminology

We assume that systems are already allocated on processors and that each processor may use a different scheduling policy. The tasks within a system allocated on a processor do not need any information about the tasks within other systems allocated on other processors, neither do they need to be aware of the scheduling policies on other processors, when performing schedulability analysis of the system.

Definition 1: *Resource Hold Time* of a global resource R_q by task τ_i on processor P_k is denoted by $\mathrm{RHT}_{q,k,i}$ and is the maximum duration of time the global resource R_q can be locked by τ_i. In other words, $\mathrm{RHT}_{q,k,i}$ is the the maximum time interval starting from the time instant τ_i locks R_q and ending at the time instant τ_i releases R_q, which includes the longest critical section in which τ_i accesses R_q as well as the possible interference form other tasks accessing global resources other than R_q. Consequently, the resource hold time of a global resource, R_q, by processor P_k (i.e., the maximum duration of time R_q is locked by any task on P_k) denoted by $\mathrm{RHT}_{q,k}$, is as follows:

$$\mathrm{RHT}_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{\mathrm{RHT}_{q,k,i}\} \tag{8.1}$$

where $\tau_{q,k}$ is the set of tasks on processor P_k sharing R_q.

The concept of resource hold times for composing multiple independently-developed real-time applications on uniprocessors has been studied previously [33, 34], however, on a multi-core (multiprocessor) platform we compute resource hold times for global resources in a different way (Section 8.4.1).

Definition 2: *Maximum Resource Wait Time* for a global resource R_q on processor P_k, denoted as $\mathrm{RWT}_{q,k}$, is the worst-case time that any task, τ_i, within P_k may wait for R_q (R_q is held by other processors) each time τ_i requests R_q. Processors waiting for a global resources are enqueued in a corresponding FIFO queue (Section 8.3.2), hence the worst case occurs when all tasks within other processors have requested R_q before τ_i.

Definition 3: A processor, P_k, is abstracted and represented by an *interface* $I_k(Q_k, Z_k)$. In the interface, Q_k represents a set of l requirements where l is the number of tasks on P_k that request at least one global resource, i.e., each requirement is extracted from a task requesting one or more global re-

sources (Section 8.5). For a processor, P_k, to be schedulable all requirements in Q_k should be satisfied. A requirement, $r_s \in Q_k$, is an expression of the maximum resource wait times of one or more global resources, e.g., $r_1 \equiv \mathrm{RWT}_{1,k} + \mathrm{RWT}_{3,k} \leq 10$ indicates that the maximum waiting time for both global resources R_1 and R_3 should not exceed 10 time units. The requirements (Q_k) of each processor is extracted from the schedulability analysis of the processor independently. Z_k in the interface is a set; $Z_k = \{\ldots, Z_{q,k}, \ldots\}$, where $Z_{q,k}$ is the Maximum Processor Locking Time ($MPLT$) which represents the maximum duration of time that any task τ_x on any other processor P_l $(l \neq k)$ may be blocked by (tasks from) P_k each time τ_x requests R_q. I.e., whenever a task, τ_x, on a processor, P_l issues a request to a global resource, R_q, the maximum (collective) time that τ_x can be blocked on resource R_q by tasks on P_k, $(k \neq l)$ is indicated by $Z_{q,k}$.

8.3.2 General Description of MSOS

The MSOS manages intra-processor and inter-processor global resource requests. Each global resource is associated with a global queue in which processors requesting the resource are enqueued. The processors are granted the resource in FIFO manner. For the global queue, FIFO fits well because prioritizing the systems on processors may not be the case, since during the development of a system, the priority of other systems may not be known. Within a processor the tasks requesting the global resource are enqueued in a local queue. We have studied and developed both priority-based and FIFO-based queues for handling intra-processor global resource requests.

Figure 8.1 shows an overview of the protocol. When the resource becomes available to the processor at the head of the global queue the eligible task (e.g., at the top of queue if FIFO is used) from the local queue within the processor can hold the resource.

Considering that a processor, P_l, can block another processor, P_k, on a global resource, R_q, up to $Z_{q,l}$ time units each time (any task within) P_k requests the resource, the worst-case waiting time ($\mathrm{RWT}_{q,k}$) for P_k to wait until R_q becomes available is bounded by the sum of all $MPLT$'s of other processors on R_q:

$$\mathrm{RWT}_{q,k} = \sum_{P_l \neq P_k} Z_{q,l} \tag{8.2}$$

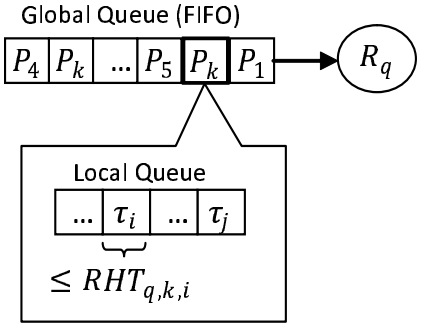


Figure 8.1: An overview of MSOS. The processor at the head of a global queue receives the corresponding global resource and within the processor the eligible task in the local queue is granted to access the resource.

8.3.3 MSOS Rules

The MSOS rules are as follows:

Rule 1: Access to local resources is controlled by a uniprocessor synchronization protocol, e.g. PCP or SRP.

Rule 2: When a task, τ_i, within a processor, P_k, requests a global resource, R_q, the priority of τ_i is increased immediately to $\rho_i + \rho^{max}(P_k)$, where $\rho^{max}(P_k) = \max\{\rho_i | \tau_i \in P_k\}$. This means a task, τ_i, which is granted to access a global resource can only be delayed or preempted by higher priority tasks executing within a *global critical section* (gcs), in which they accesses a global resource, e.g., when a higher priority task, τ_x, which is blocked on a global resource R_l ($\neq R_q$) is granted to access R_l, it resumes and preempts τ_i while τ_i is accessing R_q. This bounds blocking times on a global resource as a function of only global critical sections. The concept that the blocking time on global resources should only depend on the duration of global critical sections is one of the principles in the existing multiprocessor synchronization protocols, e.g., MPCP, MSRP [21, 23].

Rule 3: When a task, τ_i, within a processor, P_k, requests a global resource, R_q, if R_q is not locked (i.e., both local and global queues are empty), τ_i accesses R_q. If R_q is locked, a placeholder for P_k is located in the global FIFO queue of R_q and τ_i is located in the local queue of R_q and then τ_i suspends itself.

Rule 4: When global resource R_q becomes available to processor P_k the eligible task within the local queue of R_q is resumed and granted the access to

R_q. Depending on the type of local queue, the eligible task would be the one at the top of the local queue if FIFO is used and if the local queue is a prioritized queue the eligible task would be the highest priority task blocked on R_q. Note that using FIFO local queues, a task τ_i will always access a global resource R_q when the corresponding placeholder in the global queue (which is added by τ_i) is at the top of R_q's global queue. However, for a prioritized queue, it may not be the case because when a higher priority task is released, it will locate a placeholder in the global FIFO, but it may use the earlier placeholders added by lower priority tasks. This means, the lower priority task may use a later placeholder added by the higher priority tasks from P_k (Figure 8.2 shows an example of such case).

Rule 5: When a task, τ_i, on processor P_k releases a global resource, R_q, the placeholder of P_k from the top of the global FIFO queue will be removed and the resource becomes available to the processor whose placeholder is at the top of R_q's global queue.

8.4 Schedulability Analysis

8.4.1 Computing Resource Hold Times

We now describe how to compute the global resource hold time by a task and consequently by a processor.

Lemma 1. *On a processor, P_k, any task, τ_i, that is granted to access a global resource, R_q, can be interfered (either delayed at the beginning or preempted) by at most one gcs per each higher priority task, τ_j (on P_k) in which τ_j accesses a global resource other than R_q.*

Proof. For τ_i, that is granted the access to a global resource, to be interfered by two gcs's (and more) of a higher priority task, τ_j (from the same processor), τ_j needs to enter a non-critical section before entering the second gcs. On the other hand τ_i, which has been granted the access to a global resource, has a priority higher than any task that is not accessing a global resource (Rule 3). Considering that τ_i (granted to access a global resource) can only be preempted by other tasks within gcs's, τ_j will be preempted after exiting the first gcs and will not have any chance to enter the second gcs as long as τ_i has not exited its gcs. □

Based on Lemma 1, the maximum interference to any gcs of task τ_i in which it accesses a global resource R_q, from the higher priority tasks located

on the same processor, P_k, executing within their gcs's is denoted as $H_{i,q,k}$ and is computed as follows:

$$H_{i,q,k} = \sum_{\substack{\rho_i < \rho_j \\ \wedge\ R_l \in R^G_{P_k},\ l \neq q}} Cs_{j,l}$$

Consequently the resource hold time of global resource R_q by task τ_i is computed as follows:

$$\mathrm{RHT}_{q,k,i} = Cs_{i,q} + H_{i,q,k} \tag{8.3}$$

8.4.2 Blocking Times under MSOS

In this section we describe the possible situations that a task τ_i can be blocked by other tasks on the same processor as well as by other processors. Each processor may contain a different system and may have a different scheduling policy. Thus the worst case blocking overhead from other processors on a global resource, R_q, introduced to any task, τ_i (each time τ_i requests R_q), within a processor, P_k, is abstracted by $\mathrm{RWT}_{q,k}$ (Definition 1). As shown in Equation 8.2, $\mathrm{RWT}_{q,k}$ depends on the $MPLT$'s of other processors on R_q. The value of $MPLT$ of a processor on each global resource is included in the interface. However, depending on the type of the local queues i.e., FIFO or prioritized, the $MPLT$ on a global resource is calculated differently:

FIFO-based local queues In this case queueing on global resources are handled by FIFO queues. The maximum blocking time on a global resource, R_q, that tasks from P_k can introduce to any task, τ_x, located in a different processor each time τ_x requests R_q will happen when all tasks within P_k, sharing R_q, request R_q earlier than τ_x. Note that at any time instant there will be at most one placeholder per requesting task for P_k in the R_q's global FIFO queue because each task can add at most one placeholder and it cannot add another placeholder before releasing the previous one (i.e., a task cannot be in two critical sections at the same time). On the other hand the longest time that R_q can be locked by any task, τ_i, is $\mathrm{RHT}_{q,k,i}$ time units. Thus the $MPLT$ of P_k on R_q is calculated as follows:

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} \mathrm{RHT}_{q,k,i} \tag{8.4}$$

Priority-based local queues In this case queueing on global resources within processors is handled by prioritized queues; when a global resource, R_q, becomes available to a processor, P_k, the highest priority task, τ_h, is eligible to access R_q. Since FIFO is used for the global queue, similarly to the FIFO-based local queuing, the maximum blocking time that tasks from P_k on a global resource, R_q, can introduce to any task, τ_x, from a different processor each time τ_x requests R_q, will happen when all tasks within P_k, sharing R_q, request R_q earlier than τ_x. This means that the number of P_k's placeholders in R_q's global FIFO is equal to the number of the tasks within P_k sharing R_q. However, a higher priority task that requests R_q may use all these placeholders (as explained in Rule 4) Thus the the upper bound for $MPLT$ of P_k on R_q is calculated as follows:

$$Z_{q,k} = |\tau_{q,k}| \max_{\tau_i \in \tau_{q,k}} \{\mathrm{RHT}_{q,k,i}\} \tag{8.5}$$

where $|\tau_{q,k}|$ is the number of tasks in processor P_k sharing R_q.

Combining Equations 8.5 and 8.1, the result becomes as follows:

$$Z_{q,k} = |\tau_{q,k}| \, \mathrm{RHT}_{q,k} \tag{8.6}$$

The possible blocking terms that a task τ_i on a processor P_k may experience are as follows:

Local blocking due to local resources

Suppose n_i^G is the number of gcs's of τ_i. Each time τ_i is blocked on a global resource and suspended, a lower priority task τ_j may arrive and lock a local resource and may block τ_i when it resumes and after it releases the global resource. This scenario can happen up to n_i^G times. In addition, according to PCP (and SRP), task τ_i can be blocked on a local resource by at most one critical section of a lower priority task which has arrived before τ_i. On the other hand, τ_j can release at most $\lceil T_i/T_j \rceil$ jobs before the current job of τ_i finishes and each job can block τ_i's current job at most $n_j^L(\tau_i)$ times where $n_j^L(\tau_i)$ is the number of the critical sections in which τ_j requests local resources with ceiling higher than the priority of τ_i. This means τ_i can be blocked at most $\min\{n_i^G + 1, \sum_{\rho_j \leq \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\}$ times by τ_j, on local resources. Thus, the upper bound blocking time on local resources (denoted by $B_{i,1}$) is calculated as follows:

$$B_{i,1} = \min\{n_i^G + 1, \sum_{\rho_j < \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\} \max_{\substack{\rho_j < \rho_i \\ \wedge\ R_l \in R_{P_k}^L \\ \wedge\ \rho_i \leq ceil(R_l)}} \{Cs_{j,l}\} \quad (8.7)$$

where $ceil(R_l) = \max\{\rho_i |\ \tau_i\ \in\ \tau_{l,k}\}$.

Local blocking due to global resources

Before τ_i arrives or each time it suspends on a global resource, a lower priority task τ_j may access a global resource (enters a gcs) and preempt τ_i in its non-gcs sections after it arrives or resumes. Since τ_i can suspend on global resources up to n_i^G times, this type of preemption can occur at most $n_i^G + 1$ times (the additional preemption can happen by τ_j arriving and entering a gcs before τ_i arrives). On the other hand and similar to the case of local resources described above, τ_j can release at most $\lceil T_i/T_j \rceil$ jobs before the current job of τ_i finishes and each job can preempt τ_i's current job at most n_j^G times. Hence preemption from τ_j can happen at most $\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\}$ times and thus the upper bound blocking time of this type, denoted by $B_{i,2}$ introduced by lower priority tasks is calculated as follows:

$$B_{i,2} = \sum_{\substack{\rho_j < \rho_i \\ \wedge\ \{\tau_i, \tau_j\} \subseteq \tau_{P_k}}} \left(\min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R_{P_k}^G} \{Cs_{j,q}\} \right) \quad (8.8)$$

Equation 8.8 contains all the possible interference introduced to task τ_i from all gcs's of lower priority tasks including gcs's of tasks in which they share a global resource with τ_i.

Note that in both Equations 8.7 and 8.8, for simplicity we assume that the maximum blocking time (max function) will be introduced to τ_i each time it is blocked by a lower priority task. This may make the results of these equations pessimistic. More complex analysis can give tighter upper bounds.

Remote Blocking

This type of blocking occurs when task τ_i within processor P_k requests a global resource, R_q. Depending on the type of the local queues (FIFO-based or priority-based), the remote blocking is calculated differently:

Remote blocking with FIFO-based local queues In this case, when τ_i is blocked on a global resource, R_q, it is added to the local FIFO of R_q. In the worst case, all tasks within P_k sharing R_q have requested R_q before τ_i and are already in the local FIFO. However, if the tasks that requested R_q before τ_i have priority lower than that of τ_i then their effect has been included in Equation 8.8. Otherwise if the tasks requesting R_q before τ_i have priority higher than that of τ_i the interference from these tasks to τ_i is considered as the normal preemption.

On the other hand, to compute the maximum remote blocking from other processors we assume that each time τ_i requests R_q, all tasks on other processors sharing R_q have requested R_q before τ_i. Since we use FIFO global queue, each task from a different processor can lock R_q at most once before τ_i accesses R_q, this means τ_i can be blocked up to $\mathrm{RWT}_{q,k}$ time units.

This scenario can happen each time τ_i requests R_q, i.e. up to $n_{i,q}^G$ times, where $n_{i,q}^G$ is the number of τ_i's global critical sections in which it requests R_q. Thus the remote blocking with FIFO local queues is calculated as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R_{P_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q}^G \, \mathrm{RWT}_{q,k} \tag{8.9}$$

In order to uniform the equation used to calculate the blocking independently on the type of local queue (Section 8.5), we rewrite Equation 8.9 as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R_{P_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} \, \mathrm{RWT}_{q,k} \tag{8.10}$$

where $\alpha_{i,q} = n_{i,q}^G$.

Remote blocking with Priority-based local queues In the case of using priority-based local queues, when processor P_k gets the resource R_q the highest priority task within processor P_k accesses R_q.

To calculate the remote blocking, first we derive an upper bound for the amount of remote blocking each local higher priority task, τ_x, can introduce to task τ_i (a task is called as a local task to τ_i if it is allocated on the same processor as τ_i). Figure 8.2 illustrates (in three stages) how τ_x may introduce remote blocking into τ_i. In the first stage, τ_i requests R_q and a placeholder

for processor P_k is added to the end of global FIFO queue of R_q. τ_i would wait up to $\mathrm{RWT}_{q,k}$ time units to be eligible to access R_q if there was not any other task within P_k requesting R_q. However, as shown in the second stage, just before P_k gets R_q, task τ_x also requests R_q and another placeholder for P_k is added to the global queue. As shown in the third stage, when it becomes P_k's turn to get R_q, τ_x will access it (because it has a higher priority than τ_i) and τ_i's request is postponed to the second placeholder. This makes τ_i to wait additional $\mathrm{RWT}_{q,k}$ time units.

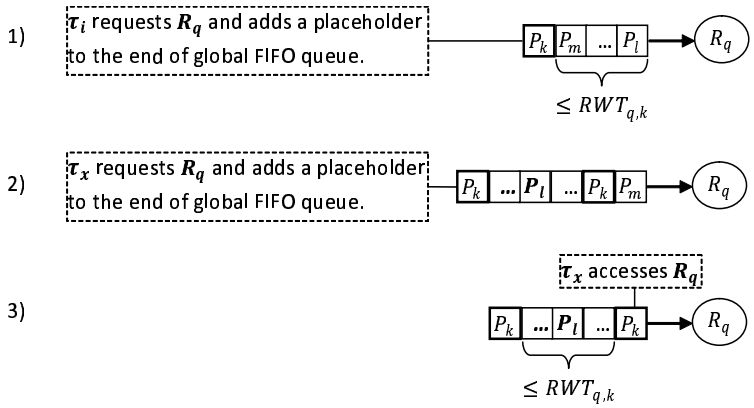


Figure 8.2: Priority-based local queue.

Thus, each gcs of τ_x in which τ_x requests R_q adds up to $\mathrm{RWT}_{q,k}$ time units blocking to τ_i. On the other hand, the higher priority task τ_x may request R_q up to $\lceil T_i/T_x \rceil \, n^G_{x,q}$ times before τ_i finishes. Each time τ_x may hold R_q up to $\mathrm{RHT}_{q,k,x}$ time units, however, as all tasks contributing to $\mathrm{RHT}_{q,k,x}$ have higher priority than τ_i, the time during which τ_x holds R_q is considered as normal preemption and not blocking. Hence, the remote blocking of τ_i on R_q introduced by higher priority task τ_x, denoted by $\mathrm{RB}_i(R_q, \tau_x)$ is calculated as follows:

$$\mathrm{RB}_i(R_q, \tau_x) = \lceil T_i/T_x \rceil \, n^G_{x,q} \, \mathrm{RWT}_{q,k} \tag{8.11}$$

where $\tau_i \in \tau_{q,k}$.

The total blocking time of τ_i on R_q introduced by all higher priority tasks sharing R_q (denoted by $\mathrm{RB}_i(R_q)$) is calculated as follows:

$$\mathrm{RB}_i(R_q) = \sum_{\substack{\rho_i < \rho_x \\ \wedge\ \{\tau_x, \tau_i\} \subseteq \tau_{q,k}}} \mathrm{RB}_i(R_q, \tau_x) \tag{8.12}$$

In addition to the remote blocking on R_q that τ_i incurs because of higher priority tasks, each gcs of τ_i may wait up to $\mathrm{RWT}_{q,k}$ time units to be granted to access R_q, i.e., τ_i may incur this blocking even if no higher priority task (on τ_i's processor) requests R_q while τ_i is waiting for R_q. The upper bound for this blocking time is $n^G_{i,q}\,\mathrm{RWT}_{q,k}$ time units. Finally, the total remote blocking time of τ_i is as follows:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} (\mathrm{RB}_i(R_q) \,+\, n^G_{i,q}\,\mathrm{RWT}_{q,k}) \tag{8.13}$$

and by replacing Equations 8.11 and 8.12:

$$B_{i,3} = \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \tag{8.14}$$

where $\alpha_{i,q} = \sum_{\substack{\rho_i < \rho_x \\ \wedge\ \{\tau_x, \tau_i\} \subseteq \tau_{q,k}}} (\lceil T_i/T_x \rceil\, n^G_{x,q}) \,+\, n^G_{i,q}$

Looking at the remote blocking for FIFO-based and priority-based local queues (Equations 8.9 and 8.13 respectively), it is shown that for the priority-based queues $B_{i,3}$ is always greater than that for FIFO-based queues. This means, using local FIFO queues (combined with global FIFO queues) always gives lower upper bounds for remote blocking time compared to using priority-based local queues. If the remote blocking is low (i.e., maximum resource wait times are small), it may seem that using local FIFO queues, whenever a higher priority task, τ_h, requests a global resource, R_q, τ_h can be delayed by all lower priority tasks that have requested the resource before τ_h which is not the case when using priority-based local queues. However, since the priority of these lower priority tasks requesting R_q is boosted (is higher than the base priority of τ_h) they will delay the execution of τ_h when τ_h is in its non-critical sections anyway, thus from the analysis point of view using priority-based local queues does not benefit higher priority tasks even though the remote blocking is very low.

8.4.3 Total Blocking Time

The total blocking time of τ_i is the summation of the three blocking terms:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} \tag{8.15}$$

Equations 8.10 and 8.14 show that $B_{i,3}$ is a function of maximum resource wait times (e.g., $\mathrm{RWT}_{q,k}$) of the global resources. Consequently B_i will also be a function of maximum resource wait times of global resources. Considering that $B_{i,1}$ and $B_{i,2}$ are constant numbers (i.e., they only depend on internal parameters), we can rewrite Equation 8.15 as follows:

$$B_i = \gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} \mathrm{RWT}_{q,k} \tag{8.16}$$

where $\gamma_i = B_{i,1} + B_{i,2}$.

8.5 Extracting the Requirements in the Interface

In this section we describe how to extract the requirements Q_k in the interface of a processor P_k from the schedulability analysis.

Each requirement in Q_k specifies a criteria on maximum resource wait times (Definition 2) of one or more global resources. We will show how to evaluate the requirement of each task τ_i accessing global shared resources.

Starting from the schedulability condition of τ_i, the maximum value of blocking time $mtbt_i$ that τ_i can tolerate without missing its deadline can be evaluated as follows.

τ_i is schedulable, using the fixed priority scheduling policy and executed in a single processor, if

$$0 < \exists t \leq T_i \ \ \mathtt{rbf_{FP}}(i,t) \leq t, \tag{8.17}$$

where $\mathtt{rbf_{FP}}(i,t)$ denotes *request bound function* of τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t, and is computed as follows:

$$\mathtt{rbf_{FP}}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j) \tag{8.18}$$

By substituting B_i by $mtbt_i$ in Equations 8.17 and 8.18, we can compute $mtbt_i$ as follows:

$$mtbt_i = \max_{0<t\leq T_i}(t - (C_i + \sum_{\rho_i<\rho_j}(\lceil t/T_j \rceil C_j))) \tag{8.19}$$

Note that it is not required to test all possible values of t in Equation 8.19, and only a bounded number of values of t that change $\texttt{rbf}_{\texttt{FP}}(i,t)$ should be considered (see [35] for more details).

Equation 8.16 shows that the total blocking time of task τ_i is a function of maximum resource wait times of the global resources accessed by tasks on P_k. With the achieved $mtbt_i$ and Equation 8.16 we extract a requirement:

$$\gamma_i + \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \leq mtbt_i \tag{8.20}$$

and

$$r_i \equiv \sum_{\substack{R_q \in R^G_{P_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q}\mathrm{RWT}_{q,k} \leq mtbt_i - \gamma_i \tag{8.21}$$

Note that, Equation 8.15 can be used to evaluate B_i for both the tasks that share global resources and the tasks that do not share any global resource. For a task that does not share global resources, $B_{i,3} = 0$ and Equations 8.7 and 8.8 can be used to evaluate $B_{i,1}$ and $B_{i,2}$ assigning $n_i^G = 0$. Since the remote blocking does not affect the schedulability of the tasks that do not share global resources, the schedulability test for these tasks can be done during the requirement extraction phase.

The schedulability of each processor is tested by its requirements. A processor P_k is schedulable if all the requirements in Q_k are satisfied and assuming that all tasks in P_k that do not share global resources are schedulable. To test the requirements in Q_k we need maximum resource wait times (e.g., $RWT_{q,k}$) of global resources accessed by tasks within P_k which are calculated using Equation 8.2.

8.6 Experimental Evaluation

In this section we present our experimental results of the synchronization protocol (MSOS) together with MPCP and a variant of partitioned FMLP (called

long FMLP). In fact, under FMLP global resources are divided into two groups; long resources and short resources, where tasks blocked on long resources suspend while tasks blocked on short resources spin (busy-wait). Dividing of global resources into long and short resources is user-defied and there is no method to efficiently decide which resources should be long or short. In a comparison of FMLP to other synchronization protocols (e.g., MPCP) in [29], FMLP is divided into two variants, i.e., long FMLP (all global resources are long) and short FMLP (all global resources are short). In this paper we consider only long FMLP. Note that MSOS can easily be extended to support spin blocking on global resources as well. However, since under spin blocking, a task blocked on a global resource executes non-preemptively, the worst case blocking times practically would be the same as short FMLP as well as the spin-based variant of MPCP [7].

As shown in Section 8.4 priority-based local queues will always introduce more blocking overheads than FIFO-based local queues and hence FIFO-based local queues will always perform better. Therefore, we have only evaluated the MSOS with FIFO-based local queues.

We have developed MSOS as a synchronization protocol for independently-developed systems on multi-cores and thus we have abstracted overhead introduced to a processor from other resources due to resource sharing. However, to evaluate our protocol and investigate the performance loss due to the abstractions, we have performed experimental evaluations and compared the performance of MSOS to MPCP and FMLP.

It turned out that MSOS, that compared to MPCP and FMLP additionally supports independently-developed systems, does not come with any significant reduction in performance due to composability. In fact MSOS, depending on the settings of the systems under analysis, may even perform better than either one or both of the alternative synchronization protocols. Hence, besides offering the possibility of composability of independently-developed systems, MSOS can be used as a regular synchronization protocol for one single system distributed over processors as it offers relatively a simple schedulability analysis method compared to FMLP and MPCP. The schedulability analysis is simple in the sense that the local schedulability analysis for each processor is performed only once and in the case of changing a system allocated on a processor or introducing a new system (on a new processor), the schedulability analysis of other processors does not to be redone. The only test to be performed is to check that the requirements of all processors are still valid which is much simpler than performing the whole schedulability analysis for every processor.

8.6.1 Experiment Setup

To determine the performance of MSOS compared to other synchronization protocols, we tested the schedulability for each protocol using randomly generated systems. For schedulability analysis of MPCP and FMLP we used the methods described in [21] and [28] respectively to perform the analysis. The tasks within each system allocated on each processor were generated based on parameters as follows. The utilization of each task was randomly chosen between 0.01 and 0.1, and its period was randomly chosen between $10ms$ and $100ms$. The execution time of each task was calculated based on its utilization and period. For each system (processor), tasks were generated until the utilization of the system reached a cap or a maximum number of 40 tasks generated. The utilization cap ranged from $\{0.1,\ 0.2,\ 0,3,\ 0.4,\ 0.5,\ 0.6\}$. In both MPCP and FMLP, calculation of blocking times of each task (allocated on a processor) that shares global resources, depends on the timing attributes of remote tasks (allocated on other processor), e.g., task priorities. For each set of parameters, to achieve a global priority setting, all tasks from all systems were put in a single list and priorities were assigned to them based on Rate Monotonic (RM).

The resource sharing parameters were chosen as follows. The number of resources shared among all tasks (within all systems) was chosen from $\{5, 10, 15, 20\}$. The number of requests each task issued (i.e., the number of critical sections) was randomly chosen from $[0, \mathrm{CsNum}]$ where CsNum ranged from 1 to 6. The length of each critical section was randomly chosen from $[\mathrm{minL}, \mathrm{maxL}]$ which ranged from $\{[5\mu s, 10\mu s], [10\mu s, 20\mu s], [20\mu s, 40\mu s],\ [40\mu s, 80\mu s],\ [80\mu s, 160\mu s],\ [160\mu s, 320\mu s]\}$.

For each setting point we generated 1000 samples. For some settings we repeated the experiments 10 times which always yielded the same results, confirming that 1000 samples per each setting can be representative.

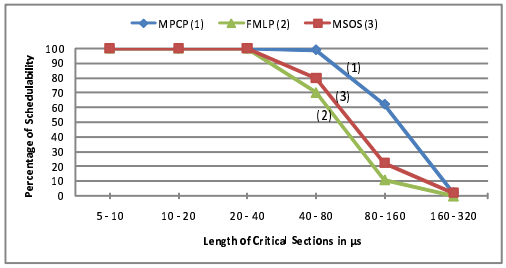
Preemption overhead: Preemption overhead which includes cache state loss as well as context switches, is a platform dependant and may differ in different platforms significantly. However, to determine the performance of different synchronization protocols and considering preemption overhead, besides running our experiments under no preemption overhead we ran them considering various preemption overheads (i.e., $10\mu s$, $20\mu s$, $40\mu s$ and $80\mu s$) as well. In all three protocols, tasks within their non-critical sections suffer from the same preemptions (i.e., from higher priority tasks and from lower priority tasks within gcs's), hence, we assume those overheads are counted for in the worst-case execution times of tasks. The difference is preemptions within gcs's,

from which FMLP does not suffer as tasks within their gcs's execute non-preemptively. Both MSOS and MPCP suffer from preemptions within gcs's. Preemption overhead makes a gcs longer which consequently punishes (blocks more) tasks (from other processors) requesting the same global resource as is accessed in the gcs. Under MPCP, a task within a gcs can be preempted by gcs's from both lower priority and higher priority tasks (for more details see [21]). Under MSOS, on the other hand, a task within a gcs can only be preempted by higher priority tasks within their gcs's.

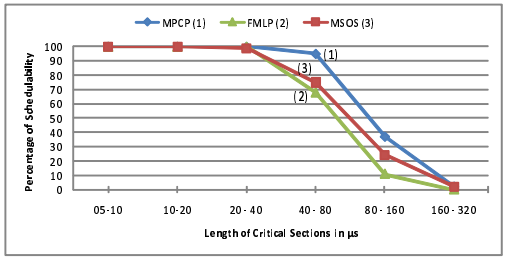
8.6.2 Results

In this section we present the evaluation results. Overall results show that in most cases MSOS performs better than at least one of the other protocols. We have performed experiments according to all different parameters (combining all the parameters, we achieve 2880 different settings), however it is not feasible to present all results in this paper. Thus, we present our observations according to three important factors that affected the protocols differently; the length of critical sections, the number of critical sections, and the preemption overhead.

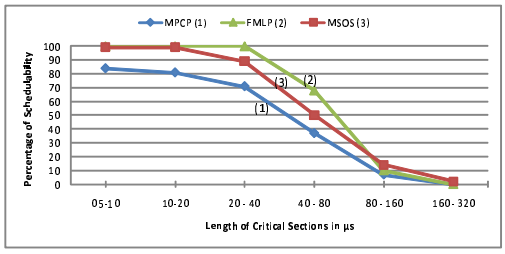
Regarding the length of critical sections, under no preemption overhead, as shown in Figure 8.3(a), for shorter critical sections all three protocols perform the same. However, as the length of critical sections is increased, MSOS mostly performs better than FMLP while MPCP exhibit the best performance. The reason is inherent in the different ways of handling queues for blocked tasks on global resources in each protocol. Under MPCP, tasks (from all processors) blocked on a global resource are enqueued in a priority-based queue while FMLP and MSOS use FIFO-based queues for blocked tasks on global resources. When the critical sections are relatively long, using FIFO queues may lead to starvation of higher priority tasks. This is why MPCP performs better than MSOS and FMLP for longer critical sections. FMLP, uses FIFO's more than MSOS; tasks waiting for a global resource are enqueued in a FIFO, and within a processor the tasks blocked on different global resources are also enqueued in FIFO manner. On the other hand in MSOS, within a processor tasks requesting different global resources are enqueued in a priority-based manner, i.e., a task that becomes eligible to access a global resource can preempt a lower priority task holding another global resource. This leads to less starvation of higher priority tasks requesting global resources specially for longer critical sections. Thus MSOS is affected by FIFO's less than FMLP but more than MPCP (MPCP does not use FIFO's at all).



(a) Preemption overhead=0μs



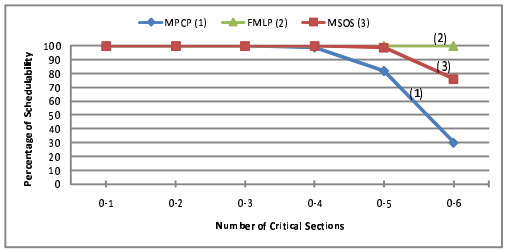
(b) Preemption overhead=10μs



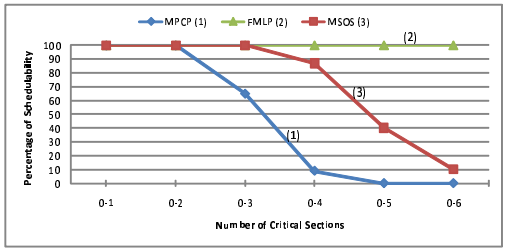
(c) Preemption overhead=40μs

Figure 8.3: Performance of synchronization protocols as the length of critical sections increases. Number of processors=8, utilization cap=0.3, number of resources=10, maximum number of critical sections per task=6.

However, when considering preemption overhead as illustrated in Figures 8.3(b) and 8.3(c), the performance of MSOS and MPCP drops; with $40\mu s$ per preemption overhead (Figure 8.3(c)), FMLP mostly exhibits the best performance and MSOS performs better than MPCP. The rationale behind this is that the global critical sections under MPCP and MSOS suffer from preemp-



(a) Preemption overhead=10μs

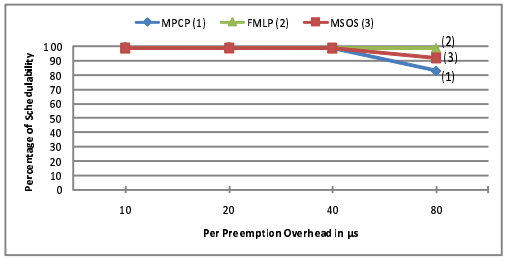


(b) Preemption overhead=40μs

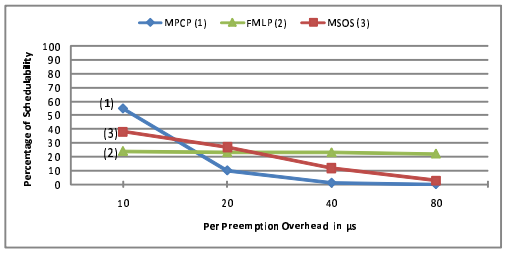
Figure 8.4: Performance of synchronization protocols as the number of critical sections increases. Number of processors=8, utilization cap=0.5, number of resources=10, length of critical sections =10-20 μs.

tion by other global critical sections which introduces preemption overhead into gcs's. This makes gcs's longer which leads to more blocking overhead for remote tasks. As confirmed by experiments, MPCP introduces more preemption overhead and thus leads to a lower schedulability of tasks.

Increasing the number of critical sections when neglecting preemption overhead had similar effect as the length of critical sections on the performance of the protocols, i.e., for lower number of critical sections all three protocols perform almost the same and as the number of critical sections are incremented the performance of FMLP and MSOS drops faster than of MPCP, although MSOS performs better than FMLP. This is also the negative effect of using FIFO's when the number of critical sections becomes higher. However, with presence of preemption overhead (Figure 8.4), when the number of critical sections is increased the performance of MPCP and MSOS drops significantly fast. The reason is that the higher number of critical sections leads to higher number of preemptions within gcs's, hence more preemption overhead within the gcs's.



(a) utilization cap=0.3



(b) utilization cap=0.5

Figure 8.5: Performance of synchronization protocols as the per preemption overhead increases. Number of processors=8, , number of resources=10, maximum number of critical sections=4, length of critical sections =40-80 μs.

This increases the length of gcs's and consequently causes longer blocking times on global resources. MSOS performs better than MPCP since MSOS suffers from less preemptions than MPCP.

Interestingly, in almost all cases MSOS performs better than at least one of the other two protocols. This is because MSOS uses a combination of FIFO-based and priority-based global resource accessing (i.e., FIFO's for global and local queues of global resources, and priority-based accessing different global resources) while FMLP and MPCP use either of them. Consequently, MSOS performs better than FMLP when the preemption overhead is low, and it performs better than MPCP when the preemption overhead is higher. Figure 8.5 shows the performance of the three protocols regarding different values of per preemption overhead.

In our experiments, we also explored the performance of the protocols regarding other parameters, e.g., various number of processors and various utilization cap per each processor. Similar to the aforementioned results, under

no preemption overhead, MSOS performs better than FMLP, and MPCP performs better than both as the number of processors and/or the utilization cap is increased. However, by increasing preemption overhead, although MSOS performs better than MPCP, the performance of both MSOS and MPCP drops significantly fast.

8.7 Conclusion

In this paper, we have proposed a synchronization protocol which manages resource sharing among independently-developed systems on a multi-core platform where each system is allocated on a dedicated core.

In our protocol, each system is presented by an interface which abstracts the sharing of global resources in the system. Furthermore, we have derived schedulability analysis under our synchronization protocol. The systems within each processor may use a different scheduling policy and priority setting, however this does not affect the schedulability analysis of a system as these design decisions are abstracted by the interfaces. This offers the possibility of different systems to be developed independently and their schedulability analysis to be performed and abstracted in their interfaces. Hence, the protocol also simplifies migration of legacy real-time systems to multi-core architectures.

In this paper we focused on the common case of non-nested critical sections. However, MSOS can support properly-nested global critical sections by means of grouping global resources whose requests are nested (similar to FMLP). In this case a joint global FIFO queue will be used for all the resources of each group. However, this may introduce very large amount of blocking overhead.

We have performed experimental evaluations of our proposed synchronization protocol, MSOS, by means of comparing its performance against two existing synchronization protocols, i.e., MPCP and FMLP. The results show that in most cases MSOS performs better than at least one of other two protocols. Therefore, we believe that MSOS is a viable protocol for independently-developed systems on a multi-core platform.

In the future we plan to implement MSOS under real-time operating systems (RTOS) and investigate its performance. Another interesting future work is to study the multiprocessor hierarchical scheduling protocols for independent/semi-independent systems with presence of shared resources.

Bibliography

[1] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *Proceedings of 7th ACM & IEEE International conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[2] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[3] N. Fisher, M. Bertogna, and S. Baruah. The Design of an EDF-Scheduled Resource-Sharing Open Environment. In *Proceedings of 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, 2007.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[5] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[6] T. Baker. A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report, 2005.

[7] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[8] F. Nemati, M. Behnam, and T. Nolte. Sharing resources among independently-developed systems on multi-cores. *ACM SIGBED Review*, 8(1), 2011.

[9] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[10] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 95–103, 2004.

[11] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of 23th IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[12] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, 2000.

[13] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 99–110, 2005.

[14] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, 2001.

[15] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[16] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of 19th Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[17] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of 20th Euromicro Conference on Real-time Systems (ECRTS'08)*, pages 181–190, 2008.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[19] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[20] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of 9th IEEE Real-Time Systems Symposium (RTSS'88)*, 1988.

[21] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[22] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[23] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[24] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[25] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of 18th Euromicro Conference on Real-time Systems (ECRTS'06)*, pages 75–84, 2006.

[26] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[27] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 342–353, 2008.

[28] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[29] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP , D-PCP , and FMLP on LITMUS. In *Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 105–124, 2008.

[30] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[31] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31st IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[32] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper*.

[33] N. Fisher, M. Bertogna, and S. Baruah. Resource-Locking Durations in EDF-Scheduled Systems. In *Proceedings of 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, 2007.

[34] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of 21st IEEE Parallel and Distributed Processing Symposium (IPDPS'07) Workshops*, pages 1–8, 2007.

[35] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic schedulability. In *Proceedings of 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

Chapter 9

Paper C: Resource Sharing among Prioritized Real-Time Applications on Multiprocessors

Farhang Nemati and Thomas Nolte
MRTC report ISSN 1404-3041 ISRN MDH-MRTC-265/2012-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2012 (submitted to conference).

Abstract

MSOS (Multiprocessors Synchronization protocol for real-time Open Systems) is a synchronization protocol for handling resource sharing among independently developed real-time applications (components) on multi-core platforms. MSOS does not consider any priority setting among applications. To handle resource sharing based on the priority of applications, in this paper we propose a new protocol that allows for resource sharing among prioritized real-time applications on a multi-core platform. We propose an optimal priority assignment algorithm which assigns unique priorities to the applications based on information in their interfaces. We have performed experimental evaluations to compare the proposed protocol (called MSOS-Priority) to the existing MSOS as well as to the current state of the art locking protocols under multiprocessor partitioned scheduling, i.e., MPCP, MSRP, FMLP and OMLP. The evaluations show that MSOS-Priority mostly performs significantly better than alternative approaches.

9.1 Introduction

The emergence of multi-core platforms and the fact that they are to be the defacto processors has attracted a lot of interest in the research community regarding multiprocessor software analysis and runtime policies, protocols and techniques.

The industry can benefit from multi-core platforms as these platforms facilitate hardware consolidation by co-executing multiple real-time applications on a shared multi-core platform. The applications may have been developed assuming the existence of various techniques, e.g., relying on a particular scheduling policy. The applications may share mutually exclusive resources. On the other hand, in industry, large and complex systems are commonly divided into several subsystems (components) which are developed in parallel and in isolation. The subsystems will eventually be integrated and co-execute on a multi-core platform.

Two main approaches for scheduling real-time systems on multi-cores exist; global and partitioned scheduling [1, 2]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor, i.e., migration of tasks among processors is permitted. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF). In this paper we focus on partitioned scheduling where tasks of each application are allocated on a dedicated processor.

In our previous work [3] we proposed a synchronization protocol for handling resource sharing among independently-developed real-time applications on multi-core platforms called Multiprocessor Synchronization protocol for real-time Open Systems (MSOS). In an open system, applications can enter and exit during run-time. The schedulability analysis of each application is performed in isolation and its demand for global resources is summarized in a set of requirements which can be used for the global scheduling when co-executing with other applications. Validating these requirements is much easier than performing the whole schedulability analysis. Thus, a run-time admission control program would perform much better when introducing a new application or changing an existing one. The protocol assumes that each real-time application is allocated on a dedicated core. Furthermore, MSOS assumes that the applications have no assigned priority and thus access to shared resources is granted in FIFO manner. However, to increase schedulability of real-time systems priority assignment is a common solution. One of the objectives of

this paper is to extend MSOS to be applicable to prioritized applications when accessing mutually exclusive resources.

9.1.1 Contributions

The main contributions of this paper are as follows:
(1) We extend MSOS such that it supports resource sharing among prioritized real-time applications allocated on a shared multi-core platform. For a given real-time application we derive an interface which includes parametric requirements. To distinguish between the two, i,e., the existing MSOS and the new MSOS, we refer them as MSOS-FIFO and MSOS-Priority respectively.
(2) We propose an optimal priority assignment algorithm which assigns unique priorities to the applications based on the information specified in their interfaces regarding shared resources.
(3) We have performed several experiments to evaluate the performance of MSOS-Priority against MSOS-FIFO as well as the state of the art locking protocol for partitioned scheduling, i.e., MPCP, MSRP, FMLP, and OMLP. To further explore the correlation of performance of the protocols to different parameters, e.g., number of processors, number of critical sections, length of critical sections, we have used a statistical method called Principal Component Analysis (PCA)[4] which is used to explore patterns in data with multiple dimensions (variables).

9.1.2 Related Work

In this section we present a non-exhaustive set of most related synchronization protocols for managing access to mutually exclusive resources on multiprocessors. We specially focus on protocols under partitioned scheduling algorithms.

The existing synchronization protocols can be categorized as *suspend-based* and *spin-based* protocols. In suspend-based protocols a task requesting a resource that is shared across processors suspends if the resource is locked by another task. In spin-based protocols a task requesting a locked resource keeps the processor and performs spin-lock (busy wait).

MPCP: Rajkumar presented MPCP (Multiprocessor Priority Ceiling Protocol) [5] for shared memory multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS (Fixed Priority Scheduling). MPCP is a suspend-based protocol where tasks waiting for a *global* resource suspend. A global resource is a resource shared among tasks

across processors. Lakshmanan et al. [6] extended a spin-based alternative of MPCP.

MSRP: Gai et al. [7] presented MSRP (Multiprocessor Stack-based Resource allocation Protocol), which is a spin-based synchronization protocol. Under MSRP, tasks blocked on a global resource perform busy wait. A task inside a global critical section (gcs) executes non-preemptively.

FMLP: Block et al. [8] presented FMLP (Flexible Multiprocessor Locking Protocol) which is a synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. Brandenburg and Anderson in [9] extended partitioned FMLP to the fixed priority scheduling policy. Under partitioned FMLP global resources are categorized into long and short resources. Tasks blocked on long resources suspend while tasks blocked on short resources perform busy wait. In an evaluation of partitioned FMLP [10], the authors differentiate between long FMLP and short FMLP where all global resources are only long and only short respectively. Thus, long FMLP and short FMLP are suspend-based and spin-based synchronization protocols respectively. In both alternatives the tasks accessing a global resource execute non-preemptively.

OMLP: Brandenburg and Anderson [11] proposed a new suspend-based locking protocol, called OMLP (O(m) Locking Protocol). OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended tasks are assumed to occupy processors and thus blocking is counted as demand. In difference with OMLP, other suspend-based protocols, are suspend-aware where suspended tasks are not assumed to occupy their processors. OMLP is *asymptotically optimal*, which means that the total blocking for any task set is a constant factor of blocking that cannot be avoided for some task sets in the worst case. An asymptotically optimal locking protocol however does not mean it can perform better than non-asymptotically optimal protocols. Our experimental evaluations confirm this fact (Section 9.8). Under OMLP, a task accessing a global resource cannot be preempted by any task until it releases the resource.

MSOS: Recently we presented MSOS [3] which is a suspend-based synchronization protocol for handling resource sharing among real-time applications in an open system on multi-core platforms. MSOS-FIFO assumes that the applications are not assigned any priority and thus applications waiting for a global resource are enqueued in an associated global FIFO-based queue. In this paper we present an alternative of MSOS, called MSOS-Priority to be applicable to prioritized applications when accessing mutually exclusive resources.

In the context of priority assignment, Audsley's Optimal Priority Assign-

ment (OPA) [12] for priority assignment in uniprocessors is the most related and similar to our priority assignment algorithm. Davis and Burns [13] have shown that OPA can be extended to fixed priority multiprocessor global scheduling if the schedulability of a task does not dependent on priority ordering among higher priority or among lower priority tasks. Our proposed algorithm is a generalization of OPA which can be applicable to assigning priorities to applications based on their requirements. Our algorithm can perform more efficiently than OPA since the schedulability test used by our algorithm is much simpler than that used in [13]. On the other hand, as we will show later in this paper (Section 9.6), although our algorithm has the same complexity as OPA, in some cases our algorithm will perform less schedulability tests than OPA.

9.2 Task and Platform Model

We assume that the multi-core platform is composed of identical, unit-capacity processors with shared memory. Each core contains a different real-time application $A_k(\rho A_k, I_k)$ where ρA_k is the priority of application A_k. Application A_k is represented by an interface I_k which abstracts the information regarding shared resources. Applications may use different scheduling policies. In this paper we focus on schedulability analysis of fixed priority scheduling. From scheduling point of view our approach can be classified as partitioned scheduling where each application can be seen as a partition (a set of tasks) allocated on one processor.

An application consists of a task set denoted by τ_{A_k} which consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time (period) between two successive jobs of task τ_i with worst-case execution time C_i and ρ_i as its priority. For the sake of simplicity we assume the tasks have implicit deadlines, i.e., the relative deadline of any job of τ_i is equal to T_i. However, with minor changes in the analysis the assumption of explicit deadlines can also be valid. A task, τ_h, has a higher priority than another task, τ_l, if $\rho_h > \rho_l$. For the sake of simplicity we also assume that each task as well as each application has a unique priority. The tasks in application A_k share a set of resources, R_{A_k}, which are protected using semaphores. The set of shared resources R_{A_k} consists of two subsets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks in the same application while a global resource is shared by tasks from more than one application. The sets of local and global resources accessed by tasks in application A_k are denoted by $R^L_{A_k}$ and $R^G_{A_k}$ respectively. The set of critical

sections, in which task τ_i requests resources in R_{A_k} is denoted by $\{Cs_{i,q,p}\}$, where $Cs_{i,q,p}$ is the worst-case execution time of the p^{th} critical section of task τ_i in which the task locks resource R_q. We denote $Cs_{i,q}$ as the worst-case execution time of the longest critical section in which τ_i requests R_q. In the context of requesting resources, when it is said that a task τ_i is granted access to a resource R_q it means that R_q is available to τ_i, however it does not necessarily mean that τ_i has started using R_q unless we concretely state that τ_i is accessing R_q which means that τ_i has entered its critical section. Furthermore, when we state that access to R_q is granted to τ_i it implies that R_q is locked by τ_i. In this paper, we focus on non-nested critical sections. A job of task τ_i, is specified by J_i.

9.3 The MSOS-FIFO for Non-prioritized Real-Time Applications

In this section we briefly present an overview of our synchronization protocol MSOS-FIFO [3] which originally was developed for non-prioritized real-time applications.

9.3.1 Definitions

Resource Hold Time (RHT)

The RHT of a global resource R_q by task τ_i in application A_k denoted by $\mathrm{RHT}_{q,k,i}$, is the maximum duration of time the global resource R_q can be locked by τ_i, i.e., $\mathrm{RHT}_{q,k,i}$ is the maximum time interval starting from the time instant τ_i locks R_q and ending at the time instant τ_i releases R_q. Thus, the resource hold time of a global resource, R_q, by application A_k denoted by $\mathrm{RHT}_{q,k}$, is as follows:

$$RHT_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{9.1}$$

where $\tau_{q,k}$ is the set of tasks in application A_k sharing R_q.

Maximum Resource Wait Time

For a global resource R_q in application A_k, denoted by $\mathrm{RWT}_{q,k}$, is the worst-case time that any task τ_i within A_k may wait for other applications on R_q whenever τ_i requests R_q. Under MSOS-FIFO, the applications waiting for a

global resource are enqueued in an associated FIFO queue. Hence the worst case occurs when all tasks within other applications have requested R_q before τ_i. As we will see (Section 9.4), this assumption is not valid for the case that the applications are prioritized.

Application Interface

An application, A_k, is represented by an *interface* $I_k(Q_k, Z_k)$ where Q_k represents a set of requirements. An application A_k is schedulable if all the requirements in Q_k are satisfied. A requirement in Q_k is a linear inequality which only depends on the maximum resource wait times of one or more global resources, e.g., $2RWT_{1,k} + 3RWT_{3,k} \leq 18$. The requirements of each application are extracted from its schedulability analysis in isolation. Z_k in the interface represents a set; $Z_k = \{\ldots, Z_{q,k}, \ldots\}$, where $Z_{q,k}$, called Maximum Application Locking Time (MALT), represents the maximum duration of time that any task τ_x in any other application A_l $(l \neq k)$ may be delayed by tasks in A_k whenever τ_x requests R_q.

9.3.2 General Description of MSOS-FIFO

Access to the local resources is handled by a uniprocessor synchronization protocol, e.g., PCP or SRP. Under MSOS-FIFO each global resource is associated with a global FIFO queue in which applications requesting the resource are enqueued. Within an application the tasks requesting the global resource are enqueued in a local queue; either priority-based or FIFO-based queues. Per each request to a global resource in the application a placeholder for the application is added to the global queue of the resource. When the resource becomes available to the application, i.e., a placeholder of the application is at the head of the global FIFO, the eligible task, e.g., at the top of local FIFO queue, within the application is granted access to the resource.

To decrease interference of applications, they have to release the locked global resources as soon as possible. In other words, the length of resource hold times of global resources have to be as short as possible. This means that a task τ_i that is granted access to a global resource R_q, should not be delayed by any other task τ_j, unless τ_j holds another global resource. To achieve this, the priority of any task τ_i within an application A_k requesting a global resource R_q is increased immediately to $\rho_i + \rho^{max}(A_k)$, where $\rho^{max}(A_k) = \max\{\rho_i | \tau_i \in \tau_{A_k}\}$. Boosting the priority of τ_i when it is granted access to a global resource will guarantee that τ_i can only be delayed or pre-

empted by higher priority tasks executing within a gcs. Thus, the RHT of a global resource R_q by a task τ_i is computed as follows:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} \tag{9.2}$$

where $H_{i,q,k} = \sum_{\substack{\forall \tau_j \in \tau_{A_k},\, \rho_i < \rho_j \\ \wedge\ R_l \in R^G_{A_k},\, l \neq q}} Cs_{j,l}$.

An application A_l can delay another application A_k on a global resource R_q up to $Z_{q,l}$ time units whenever any task within A_k requests R_q. The worst-case waiting time $RWT_{q,k}$ of A_k to wait for R_q whenever any of its tasks requests R_q is calculated as follows:

$$RWT_{q,k} = \sum_{A_l \neq A_k} Z_{q,l} \tag{9.3}$$

In [3] we derived the calculation of $Z_{q,k}$ of a global resource R_q for an application A_k, as follows:
for FIFO-based local queues:

$$Z_{q,k} = \sum_{\tau_i \in \tau_{q,k}} RHT_{q,k,i} \tag{9.4}$$

for Priority-based local queues:

$$Z_{q,k} = |\tau_{q,k}| \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}\} \tag{9.5}$$

where $|\tau_{q,k}|$ is the number of tasks in A_k sharing R_q.

9.4 The MSOS-Priority (MSOS for Prioritized Real-Time Applications)

In this section we present MSOS-Priority for real-time applications with different levels of priorities. The general idea is to prioritize the applications on accessing mutually exclusive global resources. To handle accessing the resources the global queues have to be priority-based. When a global resource becomes available, the highest priority application in the associated global queue is eligible to use the resource. Within an application the tasks requesting a global queue are enqueued in either a priority-based or a FIFO-based local queue.

When the highest priority application is granted access to a global resource, the eligible task within the application is granted access to the resource. If multiple requested global resources become available for an application they are accessed in the priority order of their requesting tasks within the application.

A disadvantage argued about spin-based protocols is that the tasks waiting on global resources perform busy wait and hence waste processor time. However, it has been shown [14] that cache-related preemption overhead, depending on the working set size (WSS) of jobs (WSS of job is the amount of memory that the job needs during its execution) can be significantly large. Thus, performing busy wait in spin-based protocols in some cases benefits the schedulability as they decrease preemptions comparing to suspend-based protocols. As our experimental evaluations show, the larger preemption overheads generally decrease the performance of suspend-based protocols significantly. However, as shown by results of our experiments, MSOS-Priority almost always outperforms all other suspend-based protocols. Furthermore, in many cases MSOS-Priority performs better than spin-based protocols even if the preemption overhead is relatively high.

Under MSOS-FIFO, a lower priority task τ_l executing within a gcs can be preempted by another higher priority task τ_h within a gcs if they are accessing different resources. This increases the number of preemptions which adds up the preemption overhead to gcs es and thus making RHT's longer. To avoid this, we modify this rule in MSOS-Priority to reduce preemptions. To achieve this the priority of a task τ_i accessing a global resource R_q has to be boosted enough that no other task, even those that are granted access to other global resources can preempt τ_i.

9.4.1 Request Rules

Rule 1: Whenever a task τ_i in an application A_k is granted access to a global resource R_q the priority of τ_i is boosted to $\rho_i + \rho^{max}(A_k)$. This ensures that if multiple global resources become available to A_k, they are accessed in the order of priorities of tasks requesting them. However, as soon as τ_i accesses R_q, i.e., starts using R_q, its priority is further boosted to 2 $\rho^{max}(A_k)$ to avoid preemption by other higher priority tasks that are granted access to other global resources. This guarantees continued access to a global resource.
Rule 2: If R_q is not locked when τ_i requests it, τ_i is granted access to R_q. If R_q is locked, A_k is added to the global priority-based queue of R_q if A_k is not already in the queueτ_i is also added to the local queue of R_q and suspends.

Rule 3: At the time R_q becomes available to A_k the eligible task within the local queue of R_q is granted access to R_q.
Rule 4: When τ_i releases R_q, if there is no more tasks in A_k requesting R_q, i.e., the local queue of R_q in A_k is empty, A_k will be removed from the global queue, otherwise A_k will remain in the queue. The resource becomes available to the highest priority application in R_q's global queue.

9.5 Schedulability Analysis under MSOS-Priority

In this section we derive the schedulability analysis of MSOS-Priority for prioritized applications. Furthermore we describe extraction of interfaces of such applications.

9.5.1 Computing Resource Hold Times

Similar to Lemma 1 in [3], it can be shown that whenever a task τ_i is granted access to a global resource R_q, it can be delayed by at most one gcs per each higher priority task τ_j where τ_j uses a global resource other than R_q. However, once τ_i starts using R_q, no task can preempt it (Rule 1). This avoids preemptions of a task while executing within a gcs.

On the other hand, once a lower priority task τ_l starts using a global resource R_s before τ_i is granted access to R_q, τ_l will delay τ_i as long as τ_l is using R_s because τ_l cannot be preempted (Rule 1). It is easy to see that τ_i will not anymore be delayed by lower priority tasks that are granted access to global resources other than R_q; whenever τ_i is granted access to a global resource R_q, in the worst-case it can be delayed for duration of the largest gcs among all lower priority tasks in which they share global resources other than R_q.

Thus $RHT_{q,k,i}$ is computed as follows:

$$RHT_{q,k,i} = Cs_{i,q} + H_{i,q,k} + \max_{\substack{\forall \tau_l \in \tau_{A_k},\, \rho_i > \rho_l \\ \wedge\ R_s \in R^G_{A_k},\, s \neq q}} \{Cs_{l,s}\} \tag{9.6}$$

9.5.2 Blocking Times under MSOS-Priority

Under MSOS-Priority, by blocking time we mean delays that any task τ_i may incur from local lower priority tasks and as well as from other applications due to mutually exclusive resources in the system. Local tasks of τ_i are the tasks that are belong to the same application as τ_i.

Similar to MSOS-FIFO, there are three possible blocking terms that a task τ_i may incur. The first and second terms are blocking incurred from the local tasks and are calculated the same way as for MSOS-FIFO [3]. Hence, because of space limitation we skip repeating explanation about how to derive the calculations of the two first blocking terms shown in Equations 9.7 and 9.8 respectively. The third blocking term is the delay incurred from other applications and is calculated in a totally different way from that in MSOS-FIFO. The blocking terms are as follows:

Local blocking due to local resources, denoted by $B_{i,1}$

Is the upper bound for the total blocking time that τ_i incurs from lower priority tasks using local resources and is calculated as follows:

$$B_{i,1} = \min\{n_i^G + 1, \sum_{\rho_j < \rho_i} \lceil T_i/T_j \rceil n_j^L(\tau_i)\} \max_{\substack{\rho_j < \rho_i \\ \wedge\ R_l \in R_{A_k}^L \\ \wedge\ \rho_i \leq ceil(R_l)}} \{Cs_{j,l}\} \tag{9.7}$$

where $ceil(R_l) = \max\{\rho_i |\ \tau_i\ \in\ \tau_{l,k}\}$, n_i^G is the number of gcs es of τ_i, and $n_j^L(\tau_i)$ is the number of the critical sections in which τ_j requests local resources with ceiling higher than the priority of τ_i.

Local blocking due to global resources, denoted by $B_{i,2}$

Is the upper bound for the maximum blocking time that τ_i incurs from lower priority tasks using global resources and can be calculated as follows:

$$B_{i,2} = \sum_{\substack{\rho_j < \rho_i \\ \wedge\ \{\tau_i, \tau_j\} \subseteq \tau_{A_k}}} \min\{n_i^G + 1, \lceil T_i/T_j \rceil n_j^G\} \max_{R_q \in R_{A_k}^G} \{Cs_{j,q}\} \tag{9.8}$$

Equation 9.8 contains all the possible delay introduced to the execution of task τ_i from all gcs es of lower priority tasks including gcs es in which they share a global resource with τ_i. Task τ_i incurs this type of blocking because of priority boosting of lower priority tasks which are granted access to global resources.

Remote blocking, denoted by $B_{i,3}$

An application A_k may introduce different values of remote blocking times to tasks in other applications. We clarify this issue by means of an example:
Example 1: Suppose that a task τ_x in an application A_l requests a global resource R_q which is already locked by a task within application A_k. In this case A_l will be added to the global queue of R_q if the queue does not already contain A_k (Rule 2). If A_k has a lower priority than A_l, after A_k releases R_q it cannot lock R_q anymore as long as A_l is in the global queue, i.e., as long as there are more tasks in A_l requesting R_q. On the other hand if A_k has a higher priority than A_l, before A_l is granted access to R_q, it will be blocked by A_k on R_q as long as A_k is in the global queue, i.e., as long as there are tasks in A_k requesting R_q. In this case the maximum delay that τ_x incurs from A_k during τ_x's period is a function of the maximum number of requests from A_k to R_q during T_i.

Thus the amount of remote blocking introduced by A_k to any task τ_x in any other application A_l depends on: (i) if A_k has a lower or higher priority than A_l, (ii) the period of τ_x.

Lemma 1. *Under MSOS-Priority, whenever any task τ_i in an application A_k requests a global resource R_q, only one lower priority application can block τ_i; this delay is at most $\max_{\rho A_k > \rho A_l}\{RHT_{q,l}\}$ time units.*

Proof. At the time τ_i in A_k requests R_q, if a lower priority application A_l has already locked R_q, it will delay A_k for at most $RHT_{q,l}$ time units. Since access to global resources is granted to applications based on their priorities, after R_q is released by A_l no more lower priority applications will have a chance to access R_q before A_k. □

Whenever any task τ_i in A_k requests a global resource R_q, it may be delayed by multiple jobs of each task within a higher priority application that request R_q. All these jobs requesting R_q will be granted access to R_q before τ_i. The maximum delay that τ_i incurs from these jobs in any time interval t is a function of the maximum number of them executing during t.

Definition 1. *Maximum Application Locking Time (MALT), denoted by $Z_{q,k}(t)$ represents the maximum delay any task τ_x in any lower priority application A_l may incur from tasks in A_k during time interval t, each time τ_x requests resource R_q.*

In order to calculate the total execution of all critical sections of all tasks in application A_k in which they use global resource R_q during time interval t,

we first need to calculate the total execution (workload) of all critical sections of each individual task in A_k in which it requests R_q during t. The maximum number of jobs generated by task τ_j during time interval t equals $\lceil \frac{t}{T_j} \rceil + 1$. On the other hand, whenever a job J_j of τ_j locks R_q it holds R_q for at most $RHT_{q,k,j}$ time units. J_j may lock R_q at most $n^G_{j,q}$ times where $n^G_{j,q}$ is the maximum number of requests to R_q issued by any job of τ_j. Thus, the total workload of all critical sections of τ_j locking R_q during time interval t is denoted by $W_j(t, R_q)$ and is computed as follows:

$$W_j(t, R_q) = (\lceil \tfrac{t}{T_j} \rceil + 1)\ n^G_{j,q}\ RHT_{q,k,j} \tag{9.9}$$

Now we can compute the maximum application locking time $Z_{q,k}(t)$ that is introduced by tasks in A_k to any task sharing global resource R_q in any lower priority application:

$$Z_{q,k}(t) = \sum_{\tau_j \in \tau_{q,k}} W_j(t, R_q) \tag{9.10}$$

Equation 9.10 can be computed in isolation and without requiring any information from other applications because the only variable is t and other parameters, e.g., $RHT_{q,k,j}$, are constants which means they are calculated using only local information. Thus, $Z_{q,k}(t)$ remains as a function of only t.

Definition 2. *Maximum Resource Wait Time (RWT) for a global resource R_q incurred by task τ_i in application A_k, denoted by $\mathrm{RWT}_{q,k,i}(t)$, is the maximum duration of time that τ_i may wait for remote applications on resource R_q during any time interval t.*

A RWT under MSOS-Priority, considering delays from lower priority applications (Lemma 1) and higher priority applications (Equation 9.10), can be calculated as follows:

$$RWT_{q,k,i}(t) = \sum_{\rho A_k < \rho A_l} Z_{q,l}(t) + n^G_{i,q} \max_{\rho A_k > \rho A_l} \{RHT_{q,l}\} \tag{9.11}$$

Under MSOS-FIFO, a RWT for a global resource is a constant value which is the same for any task sharing the resource. However, a RWT under MSOS-Priority is a function of time interval t and may differ for different tasks. The RWT for a global resource R_q of a task τ_i in application A_k during the period of τ_i equals to $RWT_{q,k,i}(T_i)$ which covers all delay introduced from both higher priority and lower priority applications sharing R_q:

$$RWT_{q,k,i} = \sum_{\rho A_k < \rho A_l} Z_{q,l}(T_i) + n^G_{i,q} \max_{\rho A_k > \rho A_l} \{RHT_{q,l}\} \tag{9.12}$$

where $RWT_{q,k,i}(T_i)$ is denoted by $RWT_{q,k,i}$.

Computing Remote Blocking: Equation 9.12 can be used to compute remote blocking $B_{i,3}$ for task τ_i. Based on Lemma 1 the maximum delay introduced by lower priority applications on a global resource R_q to any task requesting R_q is the same for all the tasks. Thus, regardless of the type of the local queues (FIFO-based or priority-based) the second term in the computation of $RWT_{q,k,i}$, shown in Equation 9.12, is the same for all tasks requesting R_q. The first term is also independent of the type of local queues as the total interference from higher priority applications during the period of each task is the same for both types of local queues. Hence, despite of the type of local queues, $B_{i,3}$ can be calculated as follows:

$$B_{i,3} = \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \tag{9.13}$$

9.5.3 Interface

The interface of an application A_k has to contain information regarding global resources which is required for schedulability analysis when the applications co-execute on a multi-core platform. It has to contain the requirements that have to be satisfied for A_k to be schedulable. Furthermore, the interface has to provide information required by other applications sharing resources with A_k.

Looking at Equation 9.12, the calculation of the RWT of a task τ_i, in application A_k, for a global resource R_q, requires MALT's, e.g., $Z_{q,h}(t)$, from higher priority applications as well as RHT's, e.g., $RHT_{q,l}$, from lower priority applications. This means that to be able to calculate the RWT's, the interfaces of the applications have to provide both RHT's and MALT's for global resources they share. Thus the interface of an application A_k is represented by $I_k(Q_k, Z_k, RHT)$ where Q_k represents a set of requirements, Z_k is a set of MALT's and a MALT is a function of time interval t. MALT's in the interface of application A_k are needed for calculating the total delay introduced by A_k to lower priority applications sharing resources with A_k. RHT in the interface is a set of RHT's of global resources shared by application A_k. RHT's are needed for calculating the total delay introduced by A_k to higher priority applications.

Extracting the Requirements

The requirements in the interface of an application under MSOS-Priority are extracted similar to MSOS-FIFO [3]. The difference is that RWT's under

MSOS-Priority may have different value for each task.

Starting from the schedulability condition of τ_i, the maximum value of blocking time B_i^{max} that τ_i can tolerate without missing its deadline can be calculated as follows:

τ_i is schedulable using the fixed priority scheduling policy if the following statement holds:

$$0 < \exists t \leq T_i \ \ \texttt{rbf}_{\texttt{FP}}(i,t) \leq t, \tag{9.14}$$

where $\texttt{rbf}_{\texttt{FP}}(i,t)$ denotes *request bound function* of τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t, and is computed as follows:

$$\texttt{rbf}_{\texttt{FP}}(i,t) = C_i + B_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j) \tag{9.15}$$

By substituting B_i by B_i^{max} in Equations 9.14 and 9.15, B_i^{max} can be calculated as follows:

$$B_i^{max} = \max_{0<t\leq T_i} (t - (C_i + \sum_{\rho_i < \rho_j} (\lceil t/T_j \rceil C_j))) \tag{9.16}$$

The total blocking of task τ_i is the summation of three blocking terms calculated in Section 9.5.2:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} \tag{9.17}$$

Since $B_{i,1}$ and $B_{i,2}$ totally depend on internal factors, i.e., the parameters from the application that τ_i belongs to, they are considered as constant values, i.e., they depend on only internal factors of τ_i's application. Thus, Equation 9.17 can be rewritten as follows:

$$B_i = \gamma_i + \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \tag{9.18}$$

where $\gamma_i = B_{i,1} + B_{i,2}$.

Equation 9.18 shows that the total blocking time of task τ_i is a function of maximum resource wait times of τ_i for the global resources accessed by τ_i. With the achieved B_i^{max} and Equation 9.18 a requirement can be extracted:

$$\gamma_i + \sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \leq B_i^{max} \tag{9.19}$$

or:

$$\sum_{\substack{\forall R_q \in R^G_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} RWT_{q,k,i} \leq B_i^{max} - \gamma_i \tag{9.20}$$

Global Schedulability Test

The schedulability of each application is tested by validating its requirements. Any application A_k is schedulable if all its requirements in Q_k are satisfied. Validating the requirements in Q_k requires maximum resource wait times, e.g., $RWT_{q,k,i}$ of global resources accessed by tasks within A_k which are calculated using Equation 9.12.

One can see that most of the calculations in the scheduling analysis of applications can be performed off-line and in isolation. The global schedulability analysis remains as testing a set of requirements which are in form of linear inequalities. This makes MSOS an appropriate synchronization protocol for open systems on multi-cores where applications can enter during run-time. An admission control program can easily test the schedulability of the system by revalidating the requirements in the interfaces.

As shown in Section 9.5.3, in an application each task sharing global resources produces one requirement, i.e., the number of requirements in the application's interface equals to the number of its tasks sharing global resources. In the worst-case all tasks in all applications share global resources. The global schedulability test requires that all requirements in all applications are validated, thus the complexity of interface-based scheduling is O(m n) where m is the number of applications and n is the number of tasks per application.

9.6 The Optimal Algorithm for Assigning Priorities to Applications

MSOS-Priority has the potential to increase the schedulability if appropriate priorities are assigned to the applications. In this section to assigns unique priorities to the applications we propose an optimal algorithm similar to the algorithm presented by Davis and Burns [13]. The algorithm is based on interface-based scheduling test which only requires information in the interfaces. The algorithm is optimal in the sense that if it fails to assign priorities to applications, any hypothetically optimal algorithm will also fail.

```
1  List remainedAppList ← all applications sharing resources;
2  for each application A in remainedAppList
3      A.priority ← 0;
4  end for
5  while ( remainedAppList is not empty )
6        List SchedulableApps ← { };
7        List NotSchedulableApps ← { };
8        for each application A in remainedAppList
9            if all requirements of A are satisfied
10                 add A to SchedulableApps;
11           else
12                 add A to NotSchedulableApps;
13           end if
14       end for
15       if SchedulableApps is empty
16             return fail;
17       remainedAppList ← NotSchedulableApps;
18       for each application A in remainedAppList
19           A.priority ← A.priority + (number of applications in SchedulableApps);
20       end for
21       incr ← 0;
22       for each application A in SchedulableApps
23           A.priority ← A.priority + incr;
24           incr ← incr + 1;
25       end for
26 end while
27 return succeed;
```

Figure 9.1: The Priority Assignment Algorithm

The pseudo code of the algorithm is shown in Figure 9.1. Initially all applications are assigned lowest priority, i.e., 0 (Line 3). The algorithm tries to, in an iterative way, increase the priority of applications. In each stage it leaves the applications that are schedulable (Line 10) and increases the priority of not schedulable applications (the for-loop in Line 18). The priority of all unschedulable applications is increased by the number of the schedulable applications in the current stage (Line 19). If there are more than one schedulable applications in the current stage, their priorities are increased in a way that each application gets a unique priority; the first application's priority is increased by 0, the second's is increased by 1, the third's is increased by 2,

etc (the for-loop in Line 22). When testing the schedulability of an application A_k, the algorithm assumes that all the applications that have the same priority as A_k are higher priority applications. This assumption helps to test if A_k can tolerate all the remaining applications if they get a higher priority than A_k. Thus, when calculating RWT's based on Equation 9.12 the algorithm changes condition $\rho A_k < \rho A_l$ in the first term to $\rho A_k \leq \rho A_l$.

Figure 9.2 illustrates an example of the algorithm.

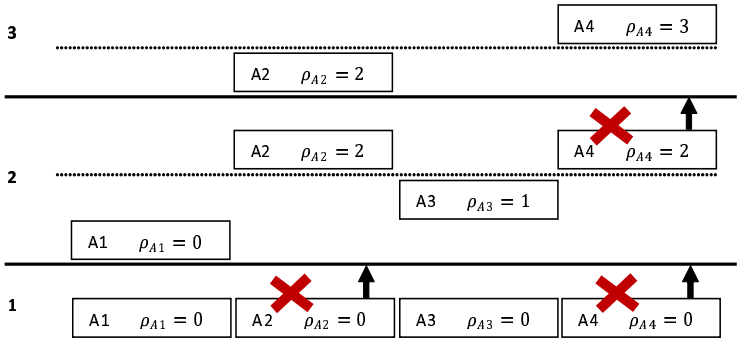


Figure 9.2: Illustrative Example for the Priority Assignment Algorithm

In the example shown in Figure 9.2, there are four applications sharing resources. The algorithm succeeds to assign priorities to them in three stages. First the algorithm gives the lowest priority to them, i.e., $\rho A_i = 0$ for each application. In this stage the algorithm realizes that applications A_1 and A_3 are schedulable but A_2 and A_4 are not schedulable, thus the priority of A_2 and A_4 are increased by 2 which is the number of schedulable applications, i.e., A_1 and A_3. Both A_1 and A_3 are schedulable, hence to assign unique priorities, the algorithm increases the priority of A_1 and A_3 by 0 and 1 respectively. Please notice that increasing the priority of the schedulable applications can be done in any order since their schedulability has been tested assuming that all the other ones have higher priority. Thus the order in which the priorities of these applications are increased will not make any of them unschedulable. In the second stage, only applications A_2 and A_4 are remained. At this stage the algorithm finds that A_4 is not schedulable, hence its priority has to be increased. In the last stage, A_4 also becomes schedulable and since all applications are now schedulable the algorithm succeeds. If at any stage the algorithm cannot find any schedulable application, meaning that none of the remaining applications can tolerate the other ones to have higher priorities, the algorithm fails.

In Audsley's priority assignment algorithm [12] to find a solution (if any) at most $m(m+1)/2$ schedulability tests will be performed where m is the number of tasks to be prioritized. Similarly, in our algorithm to find a solution (if any), in the worst case at each stage only one application is schedulable and is assigned a priority. In the next stage the schedulability of all the remaining applications has to be performed again. In this case after the algorithm is finished, the schedulability test for the applications with priority $m, m-1, \ldots, 2, 1$ has been performed $m, m-1, \ldots, 2, 1$ times respectively, and hence the maximum number of schedulability tests is $m(m+1)/2$ where m is the number of applications to be prioritized.

However, it may happen that at a stage, x number of applications are schedulable where $x > 1$. In this case the priority of all remaining applications (i.e. applications that are unschedulable at the current stage) will be increased by x (Figure 9.1, Line 19 of the algorithm). This means that, the maximum number of schedulability tests for each of the remaining applications would be decreased by x, i.e., the number of stages the algorithm runs is decreased by x. The more similar stages exist the lower the maximum number of schedulability tests will be. As a result the maximum number of stages and consequently the number schedulability tests are decreased. This is not the case in Audsley's OPA; depending on the order of selecting tasks (or applications), it is still possible that $m(m+1)/2$ schedulability tests would be performed, e.g., OPA finds a solution in exactly m stages. E.g., in the illustrative example in Figure 9.2, OPA will assign priorities in 4 stages, and if it selects the applications in order A_4, A_2, A_3, A_1, it will perform 4, 3, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 9 tests will be performed. On the other hand, our algorithm assigns priorities in 3 stages and it performs 3, 2, 1, 1 schedulability tests for A_4, A_2, A_3 and A_1 respectively, and in total 7 tests are performed.

Lemma 2. *The priority assignment algorithm is optimal, i.e., if the algorithm fails to assign unique priorities any hypothetically optimal algorithm will also fail.*

Proof. We assume that the priority assignment algorithm at some stage fails, lets assume that it fails at stage f ($1 \leq f \leq m$ where m is the number of applications), i.e., at stage f the algorithm does not find any schedulable application and thus fails. This means that there is no application in the system that can be schedulable with priority $f-1$. We assume that a hypothetically optimal algorithm succeeds to assign unique priorities to applications. This means that any application A_k is assigned a unique priority where $0 \leq \rho A_k \leq m-1$.

Thus there is a schedulable application that has priority equal to $f - 1$. This is in contradiction with the assumption with which the priority assignment algorithm fails. □

9.7 Schedulability Tests Extended with Preemption Overhead

If the tasks allocated on a processor do not share resources, since any job can preempt at most one job during its execution, it suffices to inflate the worst-case execution time of each task by one preemption overhead [15]. This type of preemption which originates from different priority levels of tasks is common under all synchronization protocols discussed in this paper, hence, we assume that this overhead is already inflated in the worst-case execution times. When tasks share local resources under the control of a uniprocessor synchronization protocol, e.g., SRP, an additional preemption overhead has to be added to the worst-case execution times. We assume that the worst-case execution times are also inflated with this preemption overhead as the synchronization protocols under partitioned scheduling algorithms generally assume reusing a uniprocessor synchronization protocol for handling local resources.

However, when tasks share global resources, depending on the synchronization protocol used, the preemption overhead may not be the same for different protocols.

9.7.1 Local Preemption Overhead

Under a suspend-based protocol, e.g., MSOS-Priority, MPCP, OMLP, whenever a task τ_i requests a global resource if the resource is locked by a task in a remote processor (application), τ_i suspends. While τ_i is suspending, lower priority tasks can execute and request global resources as well. Later on when τ_i is resumed and finishes using the global resource, it can be preempted by those lower priority tasks when they are granted access to their requested global resources. Each lower priority task τ_l can preempt τ_i up to $\lceil T_i/T_l \rceil n_l^G$ times. On the other hand τ_l cannot preempt τ_i more than $n_i^G + 1$ times. Thus, τ_i can be preempted by any lower priority task τ_l at most $\min\{n_i^G + 1, \lceil T_i/T_l \rceil n_l^G\}$ times.

Task τ_i may also experience extra preemptions from higher priority tasks requesting global resources. Whenever a higher priority task τ_h requests a global resource which is locked by remote tasks, it suspends and thus τ_i has

the chance to execute. When τ_h is granted access to the resource it will preempt τ_i. This may happen up to $\lceil T_i/T_h \rceil n_h^G$ times.

Thus, the total number of extra preemptions that a task τ_i may experience from local tasks, because of suspension on global resources, is denoted by $Lpreem_i$ and is calculated as follows:

$$Lpreem_i = \sum_{\rho_l < \rho_i} \min\{n_i^G + 1, \lceil T_i/T_l \rceil n_l^G\} + \sum_{\rho_h > \rho_i} \lceil T_i/T_h \rceil n_h^G \tag{9.21}$$

The preemption overhead in Equation 9.21 is due to suspension of tasks while they are waiting for global resources. Spin-based protocols do not suffer from this preemption overhead at all as they do not let a task suspend while waiting for a global resource.

9.7.2 Remote Preemption Overhead

Besides the preemption overhead a task τ_i, may incur from local tasks, it may incur preemption overhead propagated from tasks on remote processors/applications. Under a synchronization protocol, when a task τ_r is allowed to be preempted while it is using a global resource R_q, i.e., τ_r is within a gcs, the preemption overhead will make the critical section longer which in turn makes remote tasks wait longer for R_q. The more preemptions τ_r can experience within a gcs the more remote preemption overhead it will introduce to the remote tasks. FMLP, OMLP and MSOS-Priority do not let a task using a global resource be preempted, i.e., tasks execute non-preemptively within a gcs, therefore they are free from remote preemption overhead. However, under MPCP and MSOS-FIFO a task within a gcs can be preempted by other tasks within gcs es and thus remote preemption overhead has to be included in their schedulability tests. Under MPCP, a task within a gcs can be preempted by gcs es from both lower priority and higher priority tasks [5]. Under MSOS-FIFO a task within a gcs can only be preempted by higher priority tasks within their gcs es. Under both MPCP and MSOS-FIFO a gcs of a task τ_i in which it accesses a global resource R_q can be preempted by at most one gcs per each task τ_j in which it accesses a global resource other than R_q. This is because the preempting task τ_j will not have chance to execute and enter another gcs before τ_i releases R_q. The reason is that the priority of a task within a gcs is boosted to be higher than any priority of the local tasks.

Under MPCP the priority of a gcs of a task τ_i in which it requests a global resource R_q is boosted to its *remote ceiling* which is the summation of the highest priority of any remote task that may request R_q and the highest priority in the local processor plus one. Thus under MPCP, a gcs can be preempted by any gcs with a higher remote ceiling. Consequently, under MPCP the maximum number of preemptions a gcs of τ_i may incur, equals to the maximum number of tasks containing a gcs with a higher remote ceiling. On the other hand, under MSOS-FIFO a gcs of τ_i in which it requests a global resource R_q can only be preempted by gcs es of higher priority tasks in which they access a resource other than R_q. Thus, under MSOS-FIFO the maximum number of preemptions a gcs of τ_i, in which it access R_q, may incur equals to the maximum number of higher priority tasks with a gcs in which they access any global resource other than R_q.

The length of gcs es has to be inflated by the preemption overhead they may incur. This means gcs es become longer and under MSOS-FIFO it leads to longer RHT's.

9.8 Experimental Evaluation

In this section we present our experimental evaluations for comparison of MSOS-Priority to other synchronization protocols under the fixed priority partitioned scheduling algorithm. We compared the performance of protocols with regard to the schedulability of protocols using response time analysis. We have evaluated suspend-based as well as spin-based protocols. All spin-based synchronization protocols perform the same with regarding to global resources, because in all of them, a task waiting for a global resource performs busy wait. Thus the blocking times in those protocols are the same. We present the results of the spin-based protocols in one group and represent the protocols by SPIN. In this category we put MSRP, FMLP (short resources), as well as a version of MSOS-FIFO in which tasks waiting for global resources perform busy wait. However, the suspend-based protocols, i.e., MSOS-Priority, MPCP, FMLP (long resources), OMLP and MSOS-FIFO perform differently in different situations and thus we present their performance individually.

9.8.1 Experiment Setup

We determined the performance of the protocols based on the schedulability of randomly generated task sets under each protocol. The tasks within each task

set allocated on each processor were generated based on parameters as follows. The utilization of each task was randomly chosen between 0.01 and 0.1, and its period was randomly chosen between $10ms$ and $100ms$. The execution time of each task was calculated based on its utilization and period. For each processor, tasks were generated until the utilization of the tasks reached a cap or a maximum number of 30 tasks were generated. The utilization cap was randomly chosen from $\{0,3,\ 0.4,\ 0.5\}$.

The number of global resources shared among all tasks was 10. The number of critical sections per each task was randomly chosen between 1 and 6. The length of each critical section was randomly chosen between $5\mu s$ and $225\mu s$ with steps of $20\mu s$, i.e., 5, 25, 45, etc.

Preemption overhead: The preemption overhead that we chose was inspired by measurements done by Bastoni et al. in [14] where they measured the cache-related preemption overhead as a function of WSS of tasks. To cover a broad range of overhead, i.e., from very low (or no) per-preemption overhead to very high per-preemption overhead, for each task set the per-preemption overhead was randomly chosen (in μs) from $\{0,\ 20,\ 60,\ 140,\ 300,\ 620,\ 1260,\ 2540\}$. This covers preemption overhead for tasks with very small WSS, e.g., 4 kilobytes, as well as tasks with very large WSS, e.g., around 4 megabytes.

We generated 1 million task sets. In the generated task sets the number of task sets were between 115 and 215 for each setting, where the number of settings was 6336. We repeated the experiments three times and we did not observe any significant difference in the obtained results. This means that 1 million randomly generated samples can be representative for our settings.

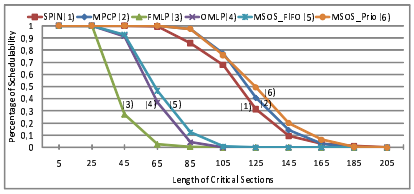
9.8.2 Results

The results of our experiments show that different synchronization protocols can be more sensitive to some factors than others, meaning that depending on different settings some of protocols may perform better.

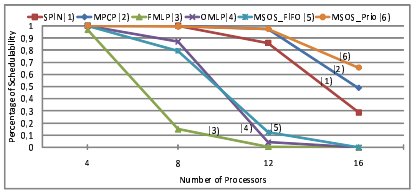
When ignoring preemption overhead, MSOS-Priority, MPCP and SPIN mostly perform significantly better than other protocols. MSOS-Priority performs better than both MPCP and SPIN as the number of processors and (or) the length of critical sections (Figures 9.3(a) and 9.3(b))[1] is increased. However, increasing the utilization cap and (or) the number of critical sections per task punishes MSOS-Priority more than MPCP and SPIN. Figures 9.3(c) and 9.3(d) show the schedulability performance of the protocols against the length

[1] Please notice that in all the figures showing the results of the experiments the lines connecting points are used to illustrate the trends and they do not represent any regression.

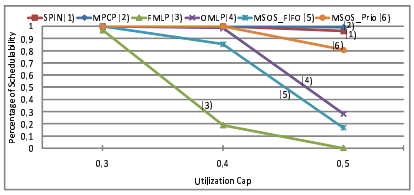
of critical sections and number of processors respectively, when the preemption overhead is ignored. As shown in Figures 9.3(d), OMLP is less sensitive to increasing the number of critical sections as it drops more smoothly compared to the rest of the protocols. For 6 critical sections per task, OMLP performs better than all protocols except MSOS-Priority and SPIN.



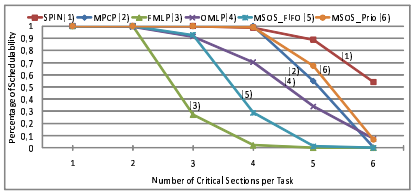
(a) Performance of synchronization protocols against the length of critical sections. Number of processors=12, utilization cap=0.3, number of critical sections per task=3.



(b) Performance of synchronization protocols against the number of processors. Utilization cap=0.3, number of critical sections per task=3, length of critical sections=85 μs.



(c) Performance of synchronization protocols against the utilization cap. Number of processors=8, number of critical sections per task=3, length of critical sections=45 μs.



(d) Performance of synchronization protocols against the number of critical sections per task. Number of processors=12, utilization cap=0.3, length of critical sections=45 μs.

Figure 9.3: Performance of synchronization protocols when the preemption overhead is ignored

As one can expect, the performance of the suspend-based protocols decreases as the preemption overhead is increased (Figure 9.4). Despite of the value of per-preemption overhead, MSOS-Priority almost always outperforms all other suspend-based protocols. Only in some cases where the preemption overhead is ignored, MSOS-Priority performs similar to MPCP.

For lower per-preemption overhead, e.g., less than $140\mu s$, in most cases where the number of processors and (or) the length of critical sections is relatively large MSOS-Priority outperforms spin-based protocols as well (Figure 9.5). However, in this paper we have not considered system dependant overhead, e.g., overhead of queue management. We believe that, similar to

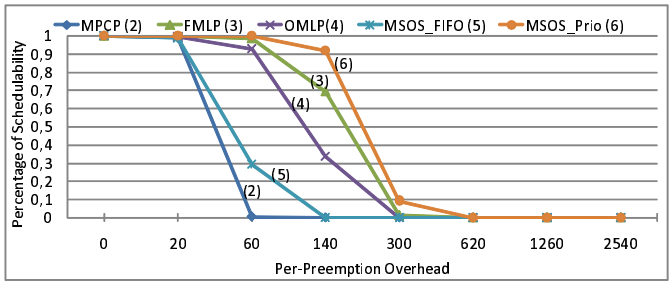


Figure 9.4: Performance of synchronization protocols as the preemption overhead increases. Number of processors=12, utilization cap=0.3, number of critical sections per task=3, length of critical sections=25 μs.

the preemption overhead, the system overhead will favor spin-based protocols significantly, and for relatively large amount of system overhead the suspend-based protocols may hardly (if not at all) outperform spin-based protocols, specially when the lengths of critical sections are relatively short.

Among suspend-based protocols MPCP drops sharply against preemption overhead already from very low per-preemption overhead followed by MSOS-FIFO. The reason that MPCP and MSOS-FIFO are more sensitive to preemption overhead is that they are the only protocols that allow preemption of a task while it is using a global resource, i.e., the task is within a gcs. Hence, only under these two protocols tasks may experience remote preemption overhead which according to the results seems to be expensive.

The local preemption overhead regarding suspension is common for all suspend-based protocols. As shown in Figure 9.4, when the preemption overhead is very low, e.g., 20 μs per-preemption, the suspend-based protocols are affected less. MPCP does not survive as the per-preemption overhead reaches $60\mu s$ and MSOS-FIFO does not survive either as the preemption overhead reaches $140\mu s$. For per-preemption overhead around $300\mu s$ only MSOS-Priority survives and when the per-preemption overhead reaches $620\mu s$ none of the suspend-based protocols survive.

So far we have seen that MSOS-Priority generally outperforms suspend-based protocols and in many cases it even performs better than spin-based protocols. However, it has not been clear how effective the priority assignment algorithm (Section 9.6) is and how much it helps MSOS-Priority protocol to perform better. To investigate the effectiveness of the priority assignment al-

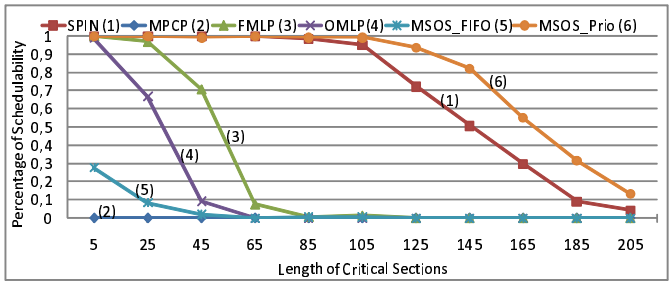


Figure 9.5: MSOS-Priority outperforms spin-based protocols in many cases for lower per-preemption overhead, e.g., as the length of critical sections is increased. Number of processors=16, utilization cap=0.3, number of critical sections per task=2, per-preemption overhead=$140\mu s$.

gorithm we performed experiments in which we compared the performance of MSOS-Priority where the priorities of applications are assigned by the priority assignment algorithm to the performance of MSOS-Priority where the priorities were assigned randomly. The results showed that the priority assignment algorithm increases the schedulability of MSOS-Priority significantly. As shown in Figure 9.6, the priority assignment algorithm boosts the performance of MSOS-Priority significantly specially when the number of applications (processors) is increased. The reason is that larger number of applications gives the priority assignment algorithm more flexibility when it assigns priorities to the applications.

To further illustrate an overview of relationship between the performance of protocols and different parameters, we have used a bilinear modeling method called Principal Component Analysis (PCA) [4]. PCA can be used to visualize and interpret relationships and insights when investigating an output against multiple variables. We have used PCA to observe which parameters and how strong they contribute to the differences among the synchronization protocols. Figure 9.7 illustrates the effect of different parameters on the synchronization protocols using PCA. P, UC, CN, CL, and O denote the number of processors, utilization cap per processor, the number of critical sections per task, length of each critical section and per-preemption overhead respectively. The closer the angle between a parameter and a protocol to 0 or 180 the more correlated the protocol is to the parameter positively or negatively respectively. Besides, the longer the vector of a parameter is the stronger the correlation is.

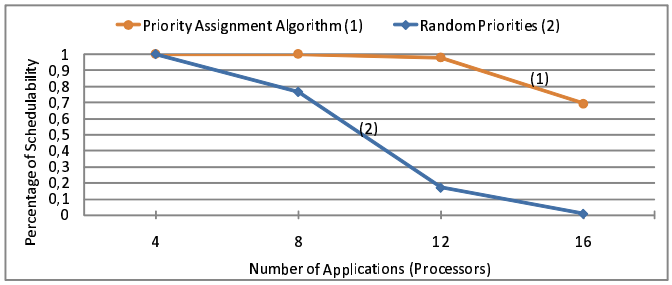


Figure 9.6: Performance of MSOS-Priority where priorities of the applications are assigned by the priority assignment algorithm against its performance where the priorities are assigned randomly. Utilization cap=0.3, number of critical sections per task=3, length of critical sections=85 μs.

An interesting interpretation illustrated in Figure 9.7, is that the suspend-based protocols are most negatively correlated to the preemption overhead, i.e., among other parameters the preemption overhead affects negatively the suspend-based protocols the most. Among suspend-based protocols, MPCP is affected the most followed by MSOS-FIFO. On the other hand the spin-based protocols are mostly affected by the length of the critical sections and the number of processors followed by the number of critical sections and the utilization cap. Briefly speaking, the preemption overhead favors spin-based protocols while the length of critical sections and the number of processors favor the suspend-based protocols.

9.9 Conclusion

In this paper, we have presented a new alternative of our previously presented synchronization protocol MSOS for independently-developed real-time applications on multi-cores [3]. MSOS was originally developed for applications that are not prioritized on accessing shared resources. In this paper we extend MSOS to support prioritized applications. In the new MSOS, called MSOS-Priority, we have extended the notion of maximum resource wait time (RWT) as well as maximum application locking time (MALT) which have to be functions of arbitrary time intervals. Moreover we have proposed an optimal priority assignment algorithm to assign priorities to applications under MSOS-Priority.

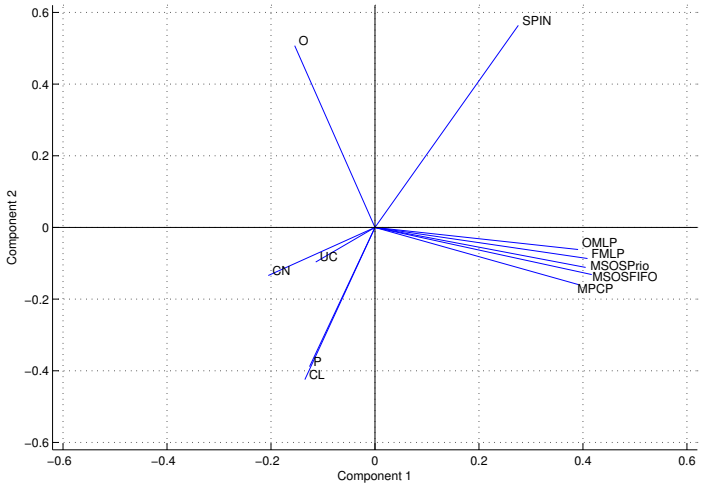


Figure 9.7: Investigate the sensitivity of the synchronization protocols against all factors using PCA.

We have performed experimental evaluation where the results showed that MSOS-Priority when combined with the priority assignment algorithm mostly performs *significantly better* than the existing suspend-based synchronization protocols under partitioned scheduling. In many cases it also outperforms spin-based protocols as well. Beside the good performance of MSOS-Priority, it offers the possibility of using it in open systems on a multi-core platform where an application is allocated on a dedicated core. An admission control program can perform better by using the interface-based global scheduling offered by MSOS-Priority since most of the complex calculations in the scheduling analysis of applications is performed off-line. Finally, MSOS generally offers real-time applications to be developed and analyzed in isolation and in parallel.

The schedulability analysis of MSOS-Priority can be improved by tightening of the calculations of the local blocking terms as well as MALT's can further , e.g., by using actual critical section lengths rather than using multiple of the longest critical sections. As a future can be the work on tightening the blocking terms. All the existing locking protocols mentioned in this paper require shared memory platforms. An interesting future work is to develop

synchronization protocols for real-time applications on multi-cores by means of message passing instead of shared memory synchronization.

Bibliography

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[2] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[3] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of 23th Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 251–261, 2011.

[4] K.H. Esbensen. *Multivariate Data Analysis - in practice (5th Edition)*. CAMO ASA, Oslo, 2010.

[5] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[6] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[7] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[9] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[10] B. B. Brandenburg and J. H. Anderson. A comparison of the M-PCP , D-PCP , and FMLP on LITMUS. In *Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 105–124, 2008.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31st IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[12] N.C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start. Technical report, 1991.

[13] R. I. Davis and A. Burns. Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 398–409, 2009.

[14] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. Is semi-partitioned scheduling practical? In *Proceedings of 23rd Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 125–135, 2011.

[15] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

Chapter 10

Paper D: Resource Sharing among Real-Time Components under Multiprocessor Clustered Scheduling

Farhang Nemati and Thomas Nolte
Journal of Real-Time Systems (under revision).

Abstract

In this paper we generalize our previously proposed synchronization protocol (MSOS) for resource sharing among independently-developed real-time applications (components) on multi-core platforms. Each component is statically allocated on a dedicated subset of processors (cluster) whose tasks are scheduled by its own scheduler. In this paper we focus on multiprocessor global fixed-priority preemptive scheduling algorithms to be used to schedule the tasks of each component on its cluster. Sharing the local resources is handled by the Priority Inheritance Protocol (PIP). For sharing the global resources (shared across components) we have studied the usage of FIFO and Round-Robin queues for access across the components and the usage of FIFO and prioritized queues within components for handling sharing of these resources. We have derived schedulability analysis for the different alternatives and compared their performance by means of experimental evaluations. Finally, we have formulated the integration phase in the form of a nonlinear integer programming problem whose techniques can be used to minimize the total number of processors required to guarantee the schedulability of all components.

10.1 Introduction

The availability of multi-core platforms has attracted a lot of attention in multiprocessor embedded software analysis and runtime policies, protocols and techniques. As the multi-core platforms are to be the defacto processors, the industry must cope with a potential migration towards multi-core platforms. However, the emergence of multi-core architectures introduce challenges in allowing for an efficient and predictable execution of industrial software systems.

For industry, when migrating to multi-core platforms it is important to be possible that several of the existing real-time components (applications) co-execute on a shared multi-core platform. The components may have been developed with different techniques, e.g., with different scheduling policies. However, when the components co-execute on the same multi-core platform they may share resources that require mutually exclusive access.

Looking at industrial systems, to speed up their development, it is not uncommon that large and complex systems are divided into several semi-independent subsystems (components) which are developed in parallel. In order to guarantee temporal correctness of these systems, scheduling techniques are used to enforce predictable execution of subsystems.

Two main approaches for scheduling real-time systems on multiprocessors (multi-cores) exist; global and partitioned scheduling [1, 2]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single scheduler and each task can be executed on any processor. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF). The generalization of global and partitioned scheduling algorithms is called clustered scheduling [3, 4], in which tasks are statically assigned to a subset (cluster) of processors, and within each cluster tasks are scheduled using a global scheduling algorithm.

When the components co-execute on a shared multi-core platform they may share resources that require mutually exclusive access.

Allocation of real-time components on a multi-core architecture may have the following alternatives: (i) one processor includes only one component, (ii) one processor may contain several components, (iii) a component may be allocated on multiple processors. In our previous work [5] we have studied and developed a synchronization protocol for the first alternative which is called Multiprocessors Synchronization protocol for real-time Open Systems

(MSOS). For the second alternative, the techniques developed for uniprocessors can be used, e.g., the methods presented in [6] and [7], by which the second alternative becomes similar to the first alternative. There are actually other alternatives different from those that we mentioned here. The work presented by Lipari and Bini [8] and the framework presented by Shin et al. [9] are examples of alternatives different from what we have mentioned here. In these works a component is allocated on a virtual multiprocessor (cluster) which consists of a set of virtual processors. The virtual processors are allocated on the physical processors (dynamicaly or statically) and components may share physical processors. However, in this paper we have only focused on the cases where the components are allocated on dedicated clusters where they do not share processors. Generalization of MSOS to the alternative (iii), where each component is allocated on a cluster, is the objective of this paper.

10.1.1 Contributions

The contributions of this paper are as follows:

- We develop a synchronization protocol for resource sharing among real-time components on a multi-core platform, where each component is allocated on multiple dedicated processors. We have named the new protocol as Clustered MSOS (C-MSOS).

- Given a real-time component, we derive an *interface-based schedulability condition* for C-MSOS. The interface abstracts the information regarding resource sharing of a component. We show that for schedulability analysis of a component there is no need for detailed information from other components, e.g., scheduling protocol or priority setting policy of other components.

- We formulate the integration of components as a nonlinear integer programming problem for which the algorithms in this domain can be used to minimize the total number of required processors for all components.

10.1.2 Related Work

Clustered scheduling techniques have been developed for multiprocessors (multi-cores) [3, 9]. However, in these works tasks are assumed to be independent and sharing of mutually exclusive resources is not considered.

A non-exhaustive set of existing approaches for handling resource sharing on multiprocessor platforms includes; Distributed Priority Ceiling Protocol (DPCP) [10], Multiprocessor PCP (MPCP) [10], Multiprocessor SRP (MSRP) [11], Flexible Multiprocessor Locking Protocol (FMLP) [12]

Brandenburg and Anderson [13] presented a new locking protocol, called O(m) Locking Protocol (OMLP) and recently [14] the same authors have extended OMLP to clustered scheduling. However, in difference with C-MSOS, OMLP is a *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended jobs in the scheduling analysis are assumed to occupy processors and thus blocking is counted as demand. To test the schedulability, the worst-case execution times of tasks are inflated with blocking times. This means that blocking time of any task is introduced to all lower priority tasks. In this paper we focus on *suspension-aware* locking synchronization in which suspended jobs are not assumed to occupy processors, i.e., C-MSOS is a suspension-aware synchronization protocol.

Easwaran and Andersson proposed a synchronization protocol [15] under global fixed-priority scheduling protocol. In the paper, the authors have derived schedulability analysis of the Priority Inheritance Protocol (PIP) under global scheduling algorithms and proposed a new protocol called P-PCP which is a generalization of PIP. For suspension-aware resource sharing under global scheduling policies, this is the only work that provides a schedulability test, hence in our paper we use their schedulability test and assume that within a component local resources are accessed using PIP.

Faggioli et al. proposed a server-based resource reservation protocol for resource sharing called Multiprocessor BandWidth Inheritance protocol (M-BWI) [16] which can be used for open systems on multiprocessors where hard, soft and non real-time systems may co-execute. M-BWI uses a mixture of spin-based and suspend-based approachs for tasks waiting for resources. The underlying scheduling policy is not required to be known. However, M-BWI assumes that the number of processors are known. The implementation of M-BWI seems to be complex as various states for servers have to be preserved during run-time. Furthermore, under M-BWI tasks have to be aware of each other, e.g., to establish the chain of blocks, which may make it difficult to use M-BWI with black box or legacy components.

In all the aforementioned existing synchronization protocols on multiprocessors, except the work of Faggioli et al. [16], it is assumed that the tasks of one single real-time system are scheduled on a multiprocessor platform. C-MSOS, however, allows a component to use its own scheduling policy and it abstracts the timing requirements regarding global resources shared by the

component in its interface, hence, it is not required to reveal its task attributes to other components which it shares resources with. Besides, C-MSOS offers various queue handling methods for managing resource sharing. However, results observed from our experimental evaluations reveal poor performance for some of the queue handling methods while confirms higher performance for some others methods. Recently, in industry, co-executing of several separated components (systems) on a multi-core platform (called virtualization) has been considered to reduce hardware costs [17]. We believe that C-MSOS is a natural fit for synchronization under virtualization of real-time components on multi-cores where each component is allocated on multiple processors.

Furthermore, C-MSOS is an appropriate locking protocol for open systems. In an open system components can enter the system during run-time, i.e., new components can start executing while some components are already executing. Although the existing resource sharing protocols under partitioned and global scheduling can also be adjusted to work for open systems, however, most of the scheduling analysis of components under C-MSOS is performed offline and the schedulability test of the components is summarized in a set of linear inequalities in their interfaces. Thus, when introducing components during run-time, an admission control program will perform better under C-MSOS, as it only needs to revalidate the requirements rather than performing the whole scheduling analysis.

Usually in the synchronization protocols for multiprocessors the number of processors is assumed to be a known parameter. However, in the scheduling analysis of C-MSOS, we consider the number of processors for a component as a variable which facilitates minimization of the total number of processors in the integration phase. This is done by means of extracting parametric and flexible requirements for each component where they are functions of the number of processors on which the component will be allocated.

10.2 System and Platform Model

We assume that the multiprocessor (multi-core) platform is composed of identical, unit-capacity processors with shared memory. We consider a set of real-time components, i.e., real-time (sub)systems, aimed to be allocated on the multiprocessor platform. A real-time component consists of a set of real-time tasks. A component may also include constituent components, i.e., hierarchical components, however, we focus on components composed of tasks only. Each component is allocated on a dedicated subset of processors, called *cluster*.

Each component has its local scheduler which can be any multiprocessor global scheduling algorithm, e.g., G-EDF. The jobs generated by tasks of a component can migrate among the processors within its cluster, however migration of jobs among clusters is not allowed. In this paper we focus on schedulability analysis for the global fixed-priority preemptive scheduling algorithm.

A component C_k consists of a task set denoted by τ_{C_k} which includes n_k sporadic tasks $\tau_i(T_i, E_i, D_i, \rho_i, \{(Cs_{i,q}, n_{i,q})\})$ where T_i denotes the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time E_i, relative deadline D_i and ρ_i as its unique *base priority*. A task τ_i has a higher priority than another task τ_j if $\rho_i > \rho_j$. The priority of a job of a task may temporarily be raised by a synchronization protocol which is denoted as the *effective priority*. The tasks in component C_k may share a set of mutually exclusive resources R_{C_k} which are protected using semaphores. The set of shared resources R_{C_k} consists of two sets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks within the same component (i.e., intra-component resource sharing) while a global resource is shared by tasks from more than one component (i.e., inter-component resource sharing). The sets of local and global resources accessed by tasks in component C_k are denoted by $R^L_{C_k}$ and $R^G_{C_k}$ respectively. $\{(Cs_{i,q}, n_{i,q})\}$ is a set of tuples, where $Cs_{i,q}$ in tuple $(Cs_{i,q}, n_{i,q})$ denotes the worst case execution time of the longest critical section of task τ_i in which τ_i uses resource R_q and $n_{i,q}$ is the maximum number of critical sections of any job of τ_i in which it uses resource R_q. We also denote $CsT_{i,q}$ as the maximum total amount of time that any job of τ_i uses R_q during its execution.The set of tasks in component C_k sharing R_q is denoted by $\tau_{q,k}$. In this paper, we focus on non-nested critical sections. We also assume constrained-deadline tasks, i.e., $D_i \leq T_i$ for any τ_i. A job of task τ_i is specified by J_i and the utilization factor of τ_i is denoted by u_i where $u_i = \frac{E_i}{T_i}$.

Component C_k will be allocated to a cluster of m_k processors and m_k can be any number in $[m_k^{(min)}, m_k^{(max)}]$ where $m_k^{(min)}$ and $m_k^{(max)}$ are the minimum and maximum number of processors required by C_k respectively. However, the requirements of C_k are more restricted for lower numbers of m_k. On the other hand, the more the value of m_k is increased towards $m_k^{(max)}$, the more its requirements are relaxed and consequently it can tolerate more blocking incurred from other components. Choosing an efficient value for m_k, in the integration phase, depends on the requirements of other components with which C_k shares resources.

10.3 Resource Sharing

To adhere the component-based development, the global resource requirements of each component have to be represented in an *interface* (Definition 3) and the internal details of the component have to be abstracted in the interface. Furthermore, the interface should also provide information about the maximum time duration that each global resource can be held by the component. The tasks within a component should not need any detailed information about the tasks, e.g., deadlines, periods, etc., from other components, neither do they need to be aware of the scheduling algorithms or synchronization protocols in other components.

Definition 1: *Resource Hold Time* (RHT) of a global resource R_q by task τ_i is the maximum duration of time that R_q can be locked by τ_i. The time interval in which τ_i is said to holds R_q is between the time τ_i locks R_q and the time τ_i releases R_q which except the length of critical section in which τ_i uses R_q, it also contains the delay that τ_i may incur from other tasks. The RHT of R_q by τ_i from component C_k, where C_k is allocated on m_k processors, is denoted by $RHT_{q,k,i}(m_k)$. Consequently, the resource hold time of a global resource R_q by component C_k, i.e., the maximum duration of time that R_q can be locked by any task in C_k, denoted by $RHT_{q,k}(m_k)$, is as follows:

$$RHT_{q,k}(m_k) = \max_{\tau_i \in \tau_{q,k}} \{RHT_{q,k,i}(m_k)\} \tag{10.1}$$

The parameter m_k in $RHT_{q,k}(m_k)$ indicates that the resource hold times are dependent on m_k, and hence for different values of m_k the duration of time a task may lock a global resource may change.

The concept of resource hold times for compositional real-time applications on uniprocessors was first studied in [18]. In our previous work [5] we extended this concept to multi-core (multiprocessor) platforms to calculate resource hold times of global resources under multiprocessor partitioned scheduling. In this paper we further extend RHT's to multiprocessor clustered scheduling.

Definition 2: *Maximum Resource Wait Time* (RWT) for a global resource R_q in component C_k, denoted as $RWT_{q,k}$, is the worst-case duration of time that whenever any task τ_i within C_k requests R_q it can be delayed by other components, i.e., R_q is held by tasks from other components.

Definition 3: *Component Interface:* A component C_k is abstracted and represented by an interface denoted by $I_k(Q_k(m_k)\,,\, Z_k(m_k)\,,\, m_k^{(min)}\,,\, m_k^{(max)})$. Please notice that the index of a component, i.e., k, in the specification of the

interface is used to clarify the relationships in the analysis and it does not indicate any order among the components. Furthermore, presence of the index does not mean that the number of the components are known.

Global resource requirements of C_k are encapsulated in the interface by $Q_k(m_k)$ which is a set of resource requirements that have to be satisfied for C_k to be schedulable. Similar to RHT the parameter m_k in $Q_k(m_k)$ also, indicates that the requirements are dependent on m_k and by changing the value of m_k the requirements may become different. For C_k to be schedulable on any m_k processors ($m_k^{(min)} \leq m_k \leq m_k^{(max)}$), all requirements in $Q_k(m_k)$ have to be satisfied. Each requirement $r_l(m_k)$ in $Q_k(m_k)$ which depends on m_k, is represented as a linear inequality which indicates that an expression of the maximum resource wait times of one or more global resources should not exceed a value $g_l(m_k)$ calculated in the interface extraction phase (Section 10.6), e.g., $r_1(m_k) \stackrel{def}{=} 4RWT_{2,k} + 3RWT_{3,k} \leq g_1(m_k)$. Each requirement is extracted from one task requesting at least one global resource (Section 10.6). Thus, the number of requirements equals to the number of tasks in component C_k that may request global resources. A formal definition of the requirements is as follows:

$$Q_k(m_k) = \{r_l(m_k)\} \tag{10.2}$$

where

$$r_l(m_k) \stackrel{def}{=} \sum_{\substack{\forall R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \leq g_l(m_k) \tag{10.3}$$

where $\alpha_{i,q}$ is a constant, i.e., it only depends on internal parameters of C_k (Section 10.6).

The global resource requirements in $Q_k(m_k)$ of a component C_k are extracted from the schedulability analysis of the component in isolation, i.e., to extract the requirements of a component, no information from other possible existing components on the same multi-core platform is required.

$Z_k(m_k)$ in the interface, represents a set $Z_k(m_k) = \{Z_{q,k}(m_k)\}$ where $Z_{q,k}(m_k)$ is the *Maximum Component Locking Time* ($MCLT$). $Z_{q,k}(m_k)$ represents the maximum duration of time that C_k can delay the execution of any task τ_x in any component C_l ($l \neq k$) whenever τ_x requests R_q, i.e., any time any task in C_l requests R_q its execution can be delayed by C_k for at most $Z_{q,k}(m_k)$ time units.

10.4 Locking Protocol for Real-Time Components under Clustered Scheduling

Under our proposed protocol which we call C-MSOS (Clustered MSOS), a component can be allocated on one cluster, thus the tasks within each component have to be scheduled using a global scheduling policy and local resources are to be handled using a locking protocol under global scheduling policies.

We assume that the Priority Inheritance Protocol (PIP) for multiprocessors is used for sharing local resources among tasks of a component. We extend the schedulability analysis presented by Easwaran and Andersson [15] such that it can be compatible with C-MSOS. First we review the characteristics of PIP for multiprocessors.

10.4.1 PIP on Multiprocessors

Assume that a task set is scheduled on a multiprocessor composed of m processors, and that shared resources are handled by PIP. Whenever a job J_i is blocked on a resource which is locked by another job J_j with a lower base priority than J_i, the effective priority of J_j is raised to the base priority of J_i if the effective priority of J_j is not already higher than the base priority of J_i. In this case, J_i is said to be *directly blocked* by J_j if J_i is among the m highest priority jobs [15].

Under PIP, besides direct blocking, a job J_i can also incur blocking from other lower-priority jobs whose effective priorities have been raised above J_i's priority. Furthermore, J_i may incur extra interference from higher priority jobs when they have locked a resource that J_i has requested and J_i is among the m highest priority jobs.

10.4.2 General Description of C-MSOS

Under C-MSOS, sharing local resources is handled by the multiprocessor PIP. Each global resource is associated with a *global queue* in which components requesting the resource are enqueued. Since prioritizing the components may not be possible, the global queues can be implemented in either FIFO or Round-Robin manner. In [5] we only studied FIFO-based global queues. In this paper we study both types. Since the resource queues are also shared among tasks and components it may cause contention. We assume that access to queues is performed in an atomic manner, e.g., the index of a FIFO queue has to be

an atomic variable. In this paper we do not consider the overhead regarding contention on resource queues.

Within a component the jobs requesting a global resource are enqueued in a *local queue*. The local queues can be either FIFO-based or priority-based queues. The blocking time on global resources should only depend on the duration of *global critical sections* (gcs) in which jobs access global resources. This bounds blocking times on global resources as a function of (length and number of) global critical sections only. Thus the priority of jobs accessing global resources should be boosted to be higher than any base priority within the component. The *boosted priority* of any job of task τ_i requesting any global resource equals to $\rho^{max}(C_k) + 1$, where $\rho^{max}(C_k) = \max\{\rho_i | \tau_i \in C_k\}$. If the priorities of multiple jobs are boosted they will be served in a FIFO-based queue. Boosting the priority of a job when it executes within a gcs ensures that it can only be delayed by jobs within gcs es.

10.4.3 C-MSOS Rules

The C-MSOS request rules are as follows:
Rule 1: Access to the local resources is handled by PIP.
Rule 2: When a job J_i within a component C_k requests a global resource R_q the priority of J_i is increased immediately to its boosted priority, i.e., $\rho^{max}(C_k) + 1$.
Rule 3: If global resource R_q is free, access to R_q is granted to J_i. If R_q is locked (by a local job or another component);
(i) if the global queues are FIFO-based a placeholder for C_k is added to the global queue of R_q, and
(ii) if the global queues are Round-Robin-based, C_k's placeholder is added to the global queue.

A placeholder of a component in a global queue indicates that the component has requested the resource. For Round-Robin global queues there will be at most one placeholder per each component in any global queue while a FIFO global queue may contain more than one placeholder for any component sharing the corresponding resource. This means that under Round-Robin global queues, when a component requests a global resource its placeholder is added to the associated queue only if the queue does not already contain the placeholder. However, under FIFO-based global queues, each request from a component to a global resource adds a placeholder to the associated global queue, i.e., for each request from tasks in the component to a resource a placeholder of the component is added to the queue.

Locally, for both types of global queues, J_i is located in the local queue of R_q and suspends.
Rule 4: When a global resource R_q becomes available to component C_k the eligible job is granted access to R_q.
Rule 5: When J_i is granted access to R_q all processors of the component may be busy with other jobs executing global resources other than R_q. If at any time the number of jobs that are granted access to global resources is larger than the number of available processors they will be served in a FIFO-based queue; the jobs that are granted access to global resources and waiting for processors are enqueued in a FIFO queue denoted by $allResourcesQ$. Obviously jobs in $allResourcesQ$ are granted access to different global resources and the queue cannot contain more than one job per each global resource at any time. At the time J_i is granted access to R_q, if all processors are occupied by other jobs accessing other global resources, J_i is added to $allResourcesQ$. As soon as an executing job releases a global resource (it enters a non-critical section) it will be preempted by the job (say J_x) at the top of $allResourcesQ$ (if any), and J_x will hold the global resource it has been granted access to and it will be removed from $allResourcesQ$.
Rule 6: When J_i releases R_q;
(i) in the case of using FIFO global queues, the placeholder of C_k from the top of the global FIFO queue of R_q will be removed and R_q becomes available to the component whose placeholder is now at the top of R_q's global queue,
(ii) in the case of using Round-Robin global queues, R_q becomes available to the next component whose placeholder is in the queue. If the local queue is empty the placeholder of C_k is removed.

10.4.4 Illustrative Example

In this section we show how C-MSOS works by showing an snapshot of an time interval where tasks and components interference among on global resources. In this example, shown in Figure 10.1, we assume that both global and local queues for handling the shared global resources are FIFO-based. The example contains three components, C_1, C_2 and C_3. The behavior of the tasks within components C_2 and C_3 are abstracted away, and the behavior of these two components only their resource requests and usage has been shown. In the component C_1, however, we have shown detailed execution of its tasks to illustrate how C-MSOS handles the shared resources.

We assume that components C_1, C_2 and C_3 are allocated on 2, 2, and 3 processors respectively. Component C_1 uses resources R_1, R_2, R_3 and R_4,

component C_2 uses resources R_1, R_2 and component C_3 uses resources R_1, R_3 and R_4. Component C_1 consists of 4 tasks, τ_1, τ_2, τ_3 and τ_4 where τ_1 and τ_4 have the lowest and highest priorities respectively. τ_1 uses resource R_2, both τ_2 and τ_3 use resources R_1 and R_3, and finally τ_4 uses resources R_3 and R_4.

At time instant 1, τ_2 issues a request to R_1 and it starts using R_1 since it is not locked by any other component of local task. At time instant 1.5 component C_2 issues a request to R_1. since R_1 is already locked by C_1, a placeholder of C_2 is added to the global queue of R_1. At time instant 2 component C_3 also requests R_1 and a placeholder for C_3 is added to the global queue of R_1. C_2 issues another request to R_1 at time instant 2.5 and another placeholder for C_2 is added to R_1's global queue. τ_2 releases R_1 at time instant 2.5. At this time C_2 is granted access to R_1 since its placeholder is at the top of the global queue of R_1. After C_2 releases R_1, C_3, C_2 (again), and C_1 (τ_3) will be granted access to R_1, i.e., in the same order that they have been added to the global FIFO queue of R_1.

τ_4 (from time 2.5 to 5), τ_3 (from time 3 to 5.5), τ_2 (from time 3.5 to 6.5) and τ_1 (from time 3.5 to 6) incur remote blocking on R_4, R_1, R_3 and R_2 respectively.

τ_1 is granted access to R_2 at time 6, however, it cannot access R_2 yet because both processors of C_1 are occupied by τ_3 and τ_4 where they are using global resources, i.e., R_1 and R_4 respectively. Thus, τ_1 is added to $allResourcesQ$ and waits until a processor is free. τ_2 is also added to $allResourcesQ$ at time 6.5 since it is granted access to R_3 and it cannot access R_3. At time instant 7, τ_3 releases its global resource. At this time τ_1 can start using its global resource R_2 since it is at the top of $allResourcesQ$. At time 7.5 τ_4 also releases its global resource and thus τ_2 can start using R_3. τ_3 and τ_4 are preempted by τ_1 (at time 7) and τ_2 (at time 7.5) respectively since the priorities of τ_1 and τ_2 have been boosted to be higher than any priority in C_1.

Component C_3 is blocked on R_3 by τ_2 in C_1, from time 6.5 to time 10.5. A part of the blocking time is due to the critical section of τ_2, i.e., the time interval between 7.5 and 10.5, the other part of the blocking time is due to interference from other tasks, i.e., from τ_3, τ_4 and τ_1, executing in their global critical sections (Equations 10.13 and 10.14). However, since component C_1 is allocated on two processors, the total amount of time tasks other than τ_2 executing in their critical sections where they delay τ_2 between time 6.5 and 7.5 is divided by 2, i.e., accessing τ_2 to R_3 is delayed for 1 time unit.

At time instant 11.5, τ_4 is blocked by τ_3 on R_3 for 0.5 time unit and the request of τ_2 to R_3 is delayed by τ_4 from time instant 12 to 13.5.

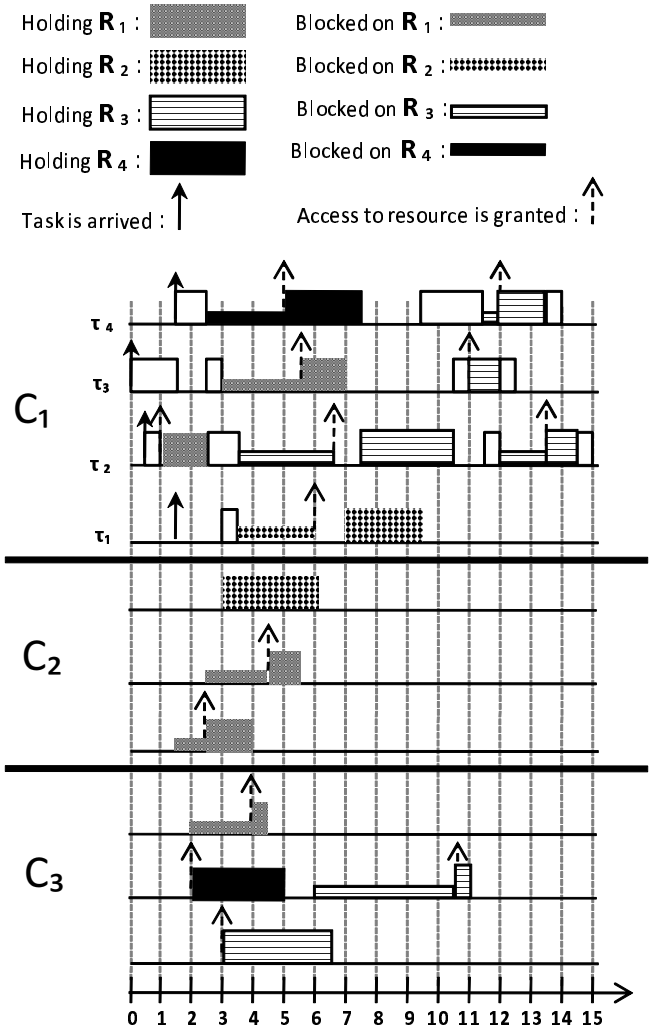


Figure 10.1: Illustrative example

10.5 Schedulability Analysis

In this section we extend the response time analysis for multiprocessor PIP [15] to be applicable to C-MSOS.

10.5.1 Schedulability Analysis of PIP

Easwaran and Andersson [15] have shown that under multiprocessor PIP the response time of any task τ_i denoted by RT_i can be calculated as follows:

$$RT_i = E_i + DB_i + Ihp_i^{(dsr)} + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i \tag{10.4}$$

where

- DB_i is an upper bound for the direct blocking (on local resources) that τ_i incurs,
- $Ihp_i^{(dsr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i lock (local) resources shared by τ_i (*direct shared resources*),
- $Ihp_i^{(osr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i may lock (local) resources not shared by τ_i (*other shared resources*),
- $Ihp_i^{(nsr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i execute in their non-critical sections, i.e., they do not hold any resource (*no shared resource*),
- Ilp_i is an upper bound for the amount of time that tasks with a lower base priority than τ_i execute with a higher effective priority than τ_i.

All the aforementioned factors that contribute to response time of τ_i, except $Ihp_i^{(nsr)}$, are delays inherent in local resources. Thus, for the sake of simplicity we rewrite Equation 10.4 as follows:

$$RT_i = E_i + Ihp_i^{(nsr)} + I^{local}(\tau_i) \tag{10.5}$$

where $I^{local}(\tau_i) = DB_i + Ihp_i^{(dsr)} + Ihp_i^{(osr)} + Ilp_i$.

To upper bound the worst-case interference from any task τ_j to task τ_i in the interval RT_i Easwaran and Andersson presented a worst case execution pattern [15]. In this pattern, during the interval RT_i, the carry-in job of τ_j executes as late as possible and all following jobs execute as early as possible. This pattern was first proposed by Bertogna and Cirinei [19] and later extended by Easwaran and Andersson to maximize the total interference from a *certain*

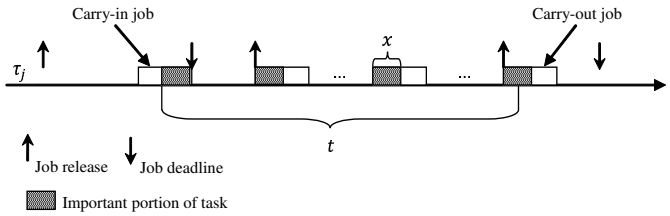


Figure 10.2: Worst-case execution pattern regarding giving importance to a certain portion of execution time.

portion x (e.g., critical sections) of execution time of any job τ_j to τ_i in RT_i. In the extended pattern, x time units of execution of the carry-in job of τ_j appear as late as possible and the x time units of execution time of its all the following jobs appear as early as possible (Fig. 10.2). In this execution pattern Easwaran and Andersson have shown that in any interval t the total execution of specific x time units of jobs of any task τ_j is maximized as follows:

$$W_j(t,x) = x\,N_j(t,x) + \min\{x, t - x + D_j - T_j N_j(t,x)\} \tag{10.6}$$

where $N_j(t,x) = \left\lfloor \frac{t-x+D_j}{T_j} \right\rfloor$.

Based on this worst-case execution pattern $DB_i, Ihp_i^{(dsr)}, Ihp_i^{(osr)}, Ihp_i^{(nsr)}$ and Ilp_i are calculated as follows (we skip repeating the rationales behind these calculations [15] and we only present the final calculations to show an overview of each term and the parameters that the term is dependent on, e.g., which terms are dependent on the number of processors):

$$DB_i = \sum_{\substack{R_q \in R^L_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q} \max_{\substack{\rho_j < \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} \{Cs_{j,q}\} \tag{10.7}$$

$$Ihp_i^{(dsr)} = \sum_{\rho_j > \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R^L_{C_k} \\ \wedge\ \{\tau_i,\tau_j\} \subset \tau_{q,k}}} CsT_{j,q}\Big) \tag{10.8}$$

$$Ihp_i^{(osr)} = \frac{\sum_{\rho_j > \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R_{C_k}^L \\ \wedge\ \tau_j \in \tau_{q,k} \\ \wedge\ \tau_i \notin \tau_{q,k}}} CsT_{j,q}\Big)}{m_k} \tag{10.9}$$

$$Ihp_i^{(nsr)} = \frac{\sum_{\rho_j > \rho_i} W_j\Big(RT_i, E_j - \sum_{\substack{R_q \in R_{C_k}^L \\ \wedge\ \tau_j \in \tau_{q,k}}} CsT_{j,q}\Big)}{m_k} \tag{10.10}$$

$$Ilp_i = \frac{\sum_{\rho_j < \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R_{C_k}^L \\ \wedge\ \tau_j \in \tau_{q,k} \\ \wedge\ \lceil R_q \rceil > \rho_i}} CsT_{j,q}\Big)}{m_k} \tag{10.11}$$

where $\lceil R_q \rceil = \max\{\rho_i | \tau_i \in \tau_{q,k}\}$

Improved Response Times for m_k Highest Priority Tasks

Easwaran and Andersson [15] further improved the computation of the response times for m_k highest priority tasks. The improved response times of m_k highest priorities is calculated as follows:

$$RT_i = \begin{cases} E_i + DB_i + Ihp_i^{(dsr)} & |\tau_H(\tau_i)| < m_k \\ E_i + DB_i + Ihp_i^{(dsr)} & \\ \quad + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i & \text{Otherwise} \end{cases} \tag{10.12}$$

where $|\tau_H(\tau_i)|$ is the number of tasks with priority higher than τ_i.

10.5.2 Schedulability Analysis of C-MSOS

Computing Resource Hold Times

In this section we determine the calculation of resource hold times of tasks and components.

Lemma 1. *Assuming that component C_k is allocated on m_k processors, whenever a task τ_i in C_k is granted access to a global resource R_q, if the number*

of tasks in C_k that share global resources is at most m_k, τ_i is not delayed by any other task, otherwise τ_i can be delayed by other tasks by at most $H_{i,q}(m_k)$ time units where:

$$H_{i,q}(m_k) = \frac{\sum_{\tau_j \neq \tau_i} \left(\max_{\substack{R_l \in R^G_{C_k},\, l \neq q \\ \wedge\ \tau_j \in \tau_{l,k}}} \{Cs_{j,l}\} \right)}{m_k} \tag{10.13}$$

Proof. When task τ_i in C_k is granted access to a global resource R_q, no any other task that does not execute in a global critical section, can delay τ_i because the priority of τ_i is boosted to be higher than the base priority of any other task in C_k (Rule 2). In this case τ_i can only be delayed by other tasks accessing global resources other than R_q. However, if the number of tasks sharing global resources is less than or equal to m_k, these tasks cannot delay τ_i on using R_q because the other tasks can use their granted resources in parallel with τ_i since they are granted access to different resources. However, if the number of these tasks is larger than m_k, at the time R_q becomes available to τ_i, if all processors are occupied by tasks accessing other global resources τ_i will be added to $allResourcesQ$ and in the worst case all other tasks that share other global resources have been granted access to their requested global resources before τ_i, e.g., they have been ahead of τ_i in $allResourcesQ$ (Rule 5). However, since these tasks are granted access to different global resources and they do not compete each other on resources and hence they can use their resources in parallel, and hence the maximum length of the execution of these in using other resources can be divided by m_k, i.e., at any time if the number of those tasks that are ahead of τ_i is less than m_k, they do not interfere with τ_i. For example in Figure 10.1 task τ_1 is granted access to global resource R_2 at time instant 6 but it cannot use the resource since τ_3 and τ_4 are using their resources, however as there are two processors for the component, the delay that τ_1 incurs from them between time 6 and 7 equals to the time τ_3 and τ_4 execute in critical sections divided by 2. Thus an upper bound of the delay that any job of τ_i incurs by other jobs when it is granted access to R_q, denoted by $H_{i,q}$, can be calculated by Equation 10.13. □

τ_i itself will hold R_q for at most $Cs_{i,q}$ time units. Thus the maximum duration of time that τ_i can lock R_q can be calculated as follows:

$$RHT_{q,k,i}(m_k) = Cs_{i,q} + H_{i,q}(m_k) \tag{10.14}$$

As shown in Equation 10.1 the resource hold time of a resource in a component is the longest resource hold time among all tasks sharing the resource.

Looking at Equations 10.13 and 10.14, it is clear that all parameters except m_k are constants, i.e., the parameters are internal parameters of C_k. Thus $RHT_{q,k,i}(m_k)$ and consequently $RHT_{q,k}(m_k)$ is a function of only m_k.

Computing Maximum Resource Wait Times

Each time a component C_k requests a global resource R_q it can be blocked by each component C_l (where $l \neq k$) up to $Z_{q,l}(m_l)$ time units. Thus the worst-case waiting time $RWT_{q,k}$ for C_k to wait until R_q becomes available is bounded by a summation of all $MCLT$ s of other components on R_q:

$$RWT_{q,k} = \sum_{l \neq k} Z_{q,l}(m_l) \tag{10.15}$$

Calculation of $MCLT$ s for components depends on the type of global queues. In the case of using FIFO global queues, whenever a component C_l requests a global resource R_q the worst case happens when all tasks from component C_k sharing R_q have issued requests before C_l, i.e., are already in the global FIFO. On the other hand each task in C_k may hold R_q up to $RHT_{q,k,i}(m_k)$ time units, hence we can calculate $Z_{q,k}(m_k)$ as follows:

$$Z_{q,k}(m_k) = \sum_{\substack{\forall \tau_i,\ \tau_i \in \tau_{q,k} \\ \wedge\ R_q \in R^G_{C_k}}} RHT_{q,k,i}(m_k) \tag{10.16}$$

If the global queues are of type Round-Robin, each component can have at most one placeholder in each global queue, e.g., whenever a global resource R_q is released by a job in component C_k the resource R_q should become available to the next component even if there are jobs waiting for R_q in the local queue of R_q. Thus when C_l is waiting for resource R_q it may wait for component C_k for at most $\max\{RHT_{q,k,i}\}$ time units which by definition (Equation 10.1) equals to $RHT_{q,k}(m_k)$:

$$Z_{q,k}(m_k) = RHT_{q,k}(m_k) \tag{10.17}$$

Response Times under C-MSOS

Under C-MSOS, the response time of any task τ_i in component C_k, further depends on interference with other jobs regarding global resource sharing, i.e.,

interference with other jobs within the same component as well as the other components. Besides the factors mentioned in Section 10.5.1, the execution of any job of τ_i may further be delayed by the following factors, denoted as *global factors*:

- The tasks with base priority lower than τ_i that may lock global resources shared by τ_i. DB_i^G (*direct global blocking*) denotes an upper bound on the amount of time (workload) that these tasks lock global resources shared by τ_i in interval RT_i. In Figure 10.1, the time interval between 11.5 and 12 where τ_4 is delayed by τ_3 is an example of this type of blocking.

- The tasks with higher (base) priority than τ_i that access global resources shared by τ_i. An upper bound of the amount of time that these tasks may delay τ_i during interval RT_i is denoted by $Ihp_i^{(dsgr)}$ (*direct shared global resources*). An example for this type of delay, shown in Figure 10.1, is the time interval between 12 and 13.5 where the request of τ_2 to R_3 is delayed by higher priority task τ_4. The reason that the delay incurred from local tasks is separately calculated by DB_i^G and $Ihp_i^{(dsgr)}$ for lower and higher priority tasks respectively, is that (we will show the computation of these terms later in this section) they are calculated differently.

- The tasks with base priority lower than τ_i that may access any global resource. The tasks holding global resources may delay the execution of any task since their effective priority is boosted to be higher than any task's priority. $Ilp_i^{(gr)}$ (*global resources*) denotes an upper bound on the amount of time that jobs of these tasks execute with boosted priority in interval RT_i. E.g., The execution of τ_4 (between time 7.5 and 9.5) and τ_3 (between time 7 and 10.5) in their non-critical sections are delayed by lower priority tasks τ_1 and τ_2 when they are accessing global resources (R_2 and R_3 respectively).

- The components other than C_k whose tasks may lock global resources shared by τ_i. RB_i (*remote blocking*) is an upper bound on the amount of time that tasks in those components lock global resources shared by τ_i during interval RT_i. E.g., as shown in Figure 10.1, tasks τ_4 (between time 2.5 and 5), τ_3 (between time 3 and 5.5), τ_2 (between time 3.5 and 6.5) and τ_1 (between time 3.5 and 6), incur remote blocking introduced by components C_2 and C_3.

Considering these interferences under C-MSOS, the response time of task τ_i is calculated as follows:

$$\begin{aligned} RT_i = \quad & E_i + Ihp_i^{(nsr)} + I^{local}(\tau_i) \\ & + DB_i^G + Ihp_i^{(dsgr)} + Ilp_i^{(gr)} + RB_i \end{aligned} \tag{10.18}$$

Computing the global factors: We now compute the global factors that may delay the execution of any job of task τ_i in component C_k. We use the dispatch pattern in Fig. 10.2 and the definition of workload in Equation 10.6 to calculate these factors.
(i) Computing DB_i^G:

- *For FIFO-based local queues:* Whenever τ_i requests a global resource R_q, it may happen (in the worst case) that all the lower-priority jobs in C_k which share R_q have requested it before τ_i and thus are located in the local FIFO queue ahead of τ_i. Hence, the maximum delay that each request of τ_i to R_q may incur from these lower priority tasks is the summation of the longest gcses of these tasks in which they use R_q. On the other hand τ_i may request R_q for at most $n_{i,q}$ times. In the worst case, τ_i may incur this type of blocking on each global resource that it shares with local lower priority tasks. Thus, DB_i^G can be calculated as follows:

$$DB_i^G = \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \Big(n_{i,q} \sum_{\substack{\rho_j < \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} Cs_{j,q} \Big) \tag{10.19}$$

- *For priority-based local queues:*

  Lemma 2. *A request of task τ_i to a global resource R_q can be delayed by the local lower priority tasks sharing R_q for the duration of at most one global critical section among all those tasks in which they use R_q.*

  Proof. Whenever τ_i requests global resource R_q, it may happen that a lower priority task τ_l in C_k has already locked R_q. However, after τ_l releases R_q no other lower priority task that has requested R_q can be granted access to R_q before τ_i. The reason is that the local tasks requesting R_q wait in a prioritized queue. Thus, the delay each request of τ_i to R_q may incur from local lower priority tasks sharing R_q can be for the duration of at most one global critical section among all these tasks. □

Considering Lemma 2 and the fact that τ_i can request each global resource R_q up to $n_{i,q}$ times, DB_i^G can be calculated as follows:

$$DB_i^G = \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q} \max_{\substack{\rho_j < \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} \{Cs_{j,q}\} \tag{10.20}$$

(ii) Computing $Ihp_i^{(dsgr)}$:

- *For FIFO-based local queues:* The calculation of this term is similar to the calculation term DB_i^G when local FIFO queues are used with the difference that $Ihp_i^{(dsgr)}$ is the upper bound for the delay incurred from higher priority tasks. In addition to lower priority tasks, in the worst case all higher priority tasks that have requested R_q before τ_i will be located ahead of τ_i in the local FIFO queue each time τ_i requests R_q. Thus $Ihp_i^{(dsgr)}$ is calculated as follows:

$$Ihp_i^{(dsgr)} = \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \Big(n_{i,q} \sum_{\substack{\rho_j > \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} Cs_{j,q}\Big) \tag{10.21}$$

- *For priority-based local queues:* In the case of prioritizing local tasks on accessing global resources, task τ_i can repeatedly be delayed by each higher priority task τ_j in C_k. Any job of a higher priority task τ_j, say J_j, can delay τ_i on accessing each global resource R_q that they share, up to $CsT_{j,q}$. Thus, the maximum delay that τ_i incurs from J_j on all global resources that it shares with τ_i equals to the following:

$$\sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \{\tau_i, \tau_j\} \subset \tau_{q,k}}} CsT_{j,q}$$

The upper bound for the total workload of all global critical sections of τ_j, in which it uses global resources with τ_i, during time interval RT_i equals to the following:

$$W_j\Big(RT_i, \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \{\tau_i, \tau_j\} \subset \tau_{q,k}}} CsT_{j,q}\Big)$$

Consequently, $Ihp_i^{(dsgr)}$ which upper bounds the total delay incurred from all the higher priority tasks to τ_i on accessing its global resources during RT_i can be calculated as follows:

$$Ihp_i^{(dsgr)} = \sum_{\rho_j > \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \{\tau_i, \tau_j\} \subset \tau_{q,k}}} CsT_{j,q}\Big) \tag{10.22}$$

(iii) Computing $Ilp_i^{(gr)}$: The priority of a task τ_j accessing a global resource is boosted and is higher than any base priority in the component (Rule 2). Thus, τ_j can delay the execution of any higher priority task τ_i when τ_i is not executing in a global critical section. The maximum time that τ_j may execute in any of its global critical sections during RT_i equals to:

$$W_j\Big(RT_i, \sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_j \in \tau_{q,k}}} CsT_{j,q}\Big)$$

Please notice that this type of delay introduced to τ_i from lower priority tasks executing in their global critical sections is originated from priority boosting and it is not because of competing with τ_i on global resources, i.e., they compete with τ_i on processors. These tasks may include the tasks that share global resources with τ_i. However, the delay from the lower priority tasks when they directly block τ_i on global resources is already calculated in blocking term DB_i^G.

Since the lower priority tasks that contribute to this type of delay to τ_i is not because of competing on any global resource, if there are enough processors they will not delay the execution of τ_i and they can execute in parallel. Thus, the total workload of the global critical sections of these tasks during RT_i has to be divided by m_k and $Ilp_i^{(gr)}$ can be calculated as follows:

$$Ilp_i^{(gr)} = \frac{\sum_{\rho_j < \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_j \in \tau_{q,k}}} CsT_{j,q}\Big)}{m_k} \tag{10.23}$$

Please notice that calculation of DB_i^G, $Ihp_i^{(dsgr)}$ and $Ilp_i^{(gr)}$ does not depend on the type of global queues (i.e., FIFO or Round-Robin), hence those calculations are valid for both types. However, the execution delay introduced

to task τ_i from other components with which τ_i shares global resources is calculated differently depending on the type of global queues as explained in the following:

(iv) Computing RB_i for FIFO global queues:

- *For FIFO-based local queues:* Whenever τ_i requests a global resource R_q, in the worst case all local tasks in C_k as well as all tasks from other components sharing R_q, have requested R_q before τ_i. However, the delays introduced to τ_i from the local tasks regarding shared global resources are included in DB_i^G and $Ihp_i^{(dsgr)}$. On the other hand, any component C_l may block τ_i for at most $Z_{q,l}$ time units any time τ_i requests R_q. This is because for FIFO global queues $Z_{q,l}$ covers the blocking time from all tasks in C_l sharing R_q and $Z_{q,l}$ is the summation of resource hold times of all tasks from component C_l that share R_q (Equation 10.16). Consequently τ_i will wait up to $RWT_{q,k} = \sum_{l \neq k} Z_{q,l}$ time units for all other components sharing R_q each time it requests R_q. Thus for FIFO global queues RB_i is calculated as follows:

$$RB_i = \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q} RWT_{q,k} \tag{10.24}$$

- *For priority-based local queues:* Whenever task τ_i issues a request to a global resource R_q a placeholder for C_k is added to the global queue of R_q. In the worst case it will take up to $RWT_{q,k}$ time units to become C_k's turn to access R_q. However, it may happen that just before C_k is at the top of the global FIFO, a higher priority task τ_h in C_k requests R_q. Now when it becomes C_k's turn to use R_q, task τ_h will use R_q because tasks in C_k are waiting for R_q in a prioritized queue. Thus, the request of τ_i is postponed to the next placeholder of C_k for R_q. This means that each request to R_q from any higher priority task in C_k will add an extra $RWT_{q,k}$ to the calculation of the delay that the request of τ_i to R_q incurs (a similar case is illustrated by an example in [5]).

  Considering that each gcs of a higher priority task τ_j in which it requests R_q may add an extra $RWT_{q,k}$ to a request of τ_i to R_q, we need to calculate the total number of gcses of all jobs of τ_j during RT_i. Each job of τ_j may have $n_{j,q}$ requests to R_q and the maximum number of jobs of τ_j

that may be generated in time interval RT_i equals to $\lceil \frac{RT_i}{T_j} \rceil + 1$.

Besides that each request from higher priority tasks adds an extra $RWT_{q,k}$ to the delay incurred by each request of τ_i to R_q, the request of τ_i itself may also wait for R_q up to $RWT_{q,k}$ time units. Thus, for the case that the local queues for accessing global resources are priority-based and the global queues are FIFO-based, RB_i can be calculated as follows:

$$RB_i = \sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \Big(n_{i,q} + \sum_{\substack{\rho_j > \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} n_{j,q} (\lceil \frac{RT_i}{T_j} \rceil + 1) \Big) RWT_{q,k} \tag{10.25}$$

(v) Computing RB_i for Round-Robin global queues:

- *For FIFO-based local queues:* When using global Round-Robin queues, in the worst case each request from component C_k to a global resource R_q may wait for one request per each component sharing R_q. On the other hand every time task τ_i in C_k requests R_q, in the worst case all local tasks have requested R_q before τ_i and are ahead of τ_i in the local FIFO queue associated with R_q. Thus, in the worst case every local request ahead of τ_i's request as well as τ_i's own request to R_q have to wait for all other components up to $RWT_{q,k} = \sum_{l \neq k} Z_{q,l}$ time units. The maximum number of requests in the local queue of R_q, denoted by $|\tau_{q,k}|$, is the total number of tasks in C_k sharing R_q. Each of the local tasks ahead of τ_i in the local FIFO queue of R_q will delay τ_i when they lock and use R_q. However, the delay that the requests of τ_i to R_q incur from local tasks when they use R_q are considered in calculations of DB^G_i and $Ihp^{(dsgr)}_i$. Thus, we can calculate RB_i for Round-Robin global queues where local queues are FIFO-based, as follows:

$$RB_i = \sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q} |\tau_{q,k}| RWT_{q,k} \tag{10.26}$$

- *For priority-based local queues:* This case is similar to the case of using priority-based local queues with FIFO-based global queues. This means that any request of τ_i to a global resource R_q may incur $RWT_{q,k}$ time units per each request to R_q from higher priority tasks. Furthermore, each request of τ_i to R_q, itself may wait up to $RWT_{q,k}$ time units

for other components accessing R_q. Thus, RB_i for the case of using Round-Robin-based global queues with priority-based local queues for accessing global resources can also be calculated by Equation 10.25.

Looking at Equations 10.24 and 10.26 it may seem that the value of remote blocking (RB_i) under Round-Robin global queues is always greater than that under FIFO global queues because under Round-Robin queues, the maximum resource wait time for each global resource is multiplied by the number of tasks sharing the global resource (e.g., $|\tau_{q,k}|$). This is not true, because maximum resource wait times are calculated differently depending on the type of global queues; comparing Equations 10.16 and 10.17 shows that under Round-Robin global queues maximum component locking times and consequently maximum resource wait times (Equation 10.15) are smaller than that under FIFO global queues. Thus depending on different factors, e.g., the number of tasks sharing a global resource, it is possible that the remote blocking under either type of global queues is larger than that under the other one.

10.5.3 Improved Calculation of Response Times under C-MSOS

Easwaran and Andersson [15] have shown that under PIP, the response time of any task τ_i among the m_k highest priority tasks only depends on E_i, DB_i and $Ihp_i^{(dsr)}$ which are the worst-case execution time of τ_i and the factors with regarded to the local resources that τ_i shares. These factors represent sequential executions and they do not depend on the number of processors available to C_k. However, as shown by Easwaran and Andersson [15] the other factors (i.e., $Ihp_i^{(nsr)}$, $Ihp_i^{(osr)}$ and Ilp_i) are affected by the number of processors and they do not affect response time of τ_i if τ_i is among the m_k highest priority tasks. In this section we present how the calculation of response times for *some* of the m_k highest priority tasks under C-MSOS can be improved.

Under C-MSOS, besides the mentioned sequential factors, the factors DB_i^G, $Ihp_i^{(dsgr)}$ and RB_i regarding the global resources accessed by τ_i are also sequential. This means that when τ_i is waiting for a locked global resource, the waiting cannot be reduced even if there is a free processor in C_k. Thus the factors DB_i^G, $Ihp_i^{(dsgr)}$ and RB_i contribute to the response time of τ_i even if τ_i is among the m_k highest priority tasks in C_k, i.e., $|\tau_H(\tau_i)| < m_k$. However, τ_i can execute in parallel with other tasks accessing global resources as long as τ_i has not requested these resources and the total number of tasks with higher

(base or boosted) priority than τ_i is less than m_k. Hence, $Ilp_i^{(gr)}$ will not affect RT_i if there are enough processors.

Thus, if the summation of the number of tasks with a higher base priority than τ_i, and the number of lower priority tasks which share any global resources is less than m_k, the execution of τ_i will never be delayed except the times it is waiting for a locked resource. Thus we can rewrite the response time calculation in Equation 10.18 for task τ_i as follows, where $|\tau_L^G(\tau_i)|$ denotes the number of tasks with a lower priority than τ_i that share any global resources:

if $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$:

$$RT_i = E_i + DB_i + Ihp_i^{(dsr)} + DB_i^G + Ihp_i^{(dsgr)} + RB_i \tag{10.27}$$

otherwise

$$\begin{aligned} RT_i = \quad & E_i + DB_i + Ihp_i^{(dsr)} + DB_i^G + Ihp_i^{(dsgr)} + RB_i \\ & + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i + Ilp_i^{(gr)} \end{aligned} \tag{10.28}$$

10.6 Extracting Interfaces

A component C_k is abstracted by its interface I_k, which consists of four elements; $Q_k(m_k)$, $Z_k(m_k)$, $m_k^{(min)}$ and $m_k^{(max)}$ (Definition 3). In Section 10.5.2 we have shown how to calculate the elements of $Z_k(m_k)$ for FIFO and Round-Robin global queues (Equations 10.16 and 10.17 respectively). In this section we determine how to extract the requirements in $Q_k(m_k)$ as well as $m_k^{(min)}$ and $m_k^{(max)}$ by means of a schedulability test of C-MSOS.

10.6.1 Deriving Requirements

As shown in Equation 10.3, a requirement in $Q_k(m_k)$ specifies that a linear expression whose variables are the maximum resource wait times of one or more global resources should not exceed a value which is a function of m_k, e.g., $g_l(m_k)$. Each requirement is derived from the schedulability analysis of one task that shares any global resources, i.e., each task sharing global resources produces one requirement.

To guarantee schedulability of a component C_k on m_k processors, for any task τ_i in C_k, condition $RT_i \leq D_i$ has to be satisfied. Looking at the calculation of response times under C-MSOS (Section 10.5.2), the response time of

tasks that do not share any global resources is only dependent on the local factors, i.e., for task τ_i the only factor in RT_i that needs information from other components (other than τ_i's component) is the remote blocking factor RB_i. If τ_i does not share any global resources then $RB_i = 0$, because it will not be blocked on any global resource by remote components. Thus, the response time of such task can be calculated without any requirement on external factors. On the other hand, if τ_i shares global resources it may incur remote blocking. However, the amount of remote blocking that τ_i can tolerate is limited and it should not exceed a value that makes τ_i to miss its deadline.

The maximum acceptable response time of τ_i denoted by $RT_i^{(max)}$, is when it equals to its deadline, i.e., $RT_i^{(max)} = D_i$. During interval $RT_i^{(max)}$ (or D_i) the delay introduced by local factors and global factors except remote blocking RB_i is constant which means that they can be calculated without any requirement on external factors from other components. The remaining slack (if any) can be taken as the maximum tolerable remote blocking. Thus the maximum remote blocking that τ_i can tolerate, denoted by $RB_i^{(max)}$, is calculated as follows:

$$RB_i^{(max)} = RT_i^{(max)} - internal_i(m_k)$$

where

$$\begin{aligned} internal_i(m_k) = \quad & E_i + DB_i + Ihp_i^{(dsr)} + DB_i^G + Ihp_i^{(dsgr)} \\ & + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i + Ilp_i^{(gr)} \end{aligned}$$

by replacing $RT_i^{(max)}$ with D_i:

$$RB_i^{(max)} = D_i - internal_i(m_k) \tag{10.29}$$

The terms in $internal_i(m_k)$ can intuitively be calculated using their corresponding equations in Section 10.5 by replacing RT_i with D_i where it is applicable.

Looking at the calculation of RB_i in Equations 10.24 and 10.26 for FIFO and Round-Robin global queues respectively we can rewrite the calculation of RB_i as follows:

$$RB_i = \sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \tag{10.30}$$

where $\alpha_{i,q} = n_{i,q}$ for FIFO and $\alpha_{i,q} = n_{i,q}|\tau_{q,k}|$ for Round-Robin queues respectively. In both cases, $\alpha_{i,q}$ is a constant, i.e., it depends only on the local factors.

Considering $RB_i \leq RB_i^{(max)}$ and by combining Equations 10.29 and 10.30 we can derive the following requirement:

$$\sum_{\substack{R_q \in R_{C_k}^G \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \leq D_i - internal_i(m_k) \tag{10.31}$$

The requirement derived in Equation 10.31 adheres the definition of a requirement in Equation 10.3 where $g_l(m_k) = D_i - internal_i(m_k)$.

The discussion in Section 10.5.3 for improvement of response times can also be applied here to improve (reduce) $internal_i(m_k)$ and consequently improve (relax) the requirement in Equation 10.31 for some of the tasks among the m_k highest priority tasks sharing global resources:

If $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$:

$$internal_i(m_k) = E_i + DB_i + Ihp_i^{(dsr)} + DB_i^G + Ihp_i^{(dsgr)} \tag{10.32}$$

otherwise

$$\begin{aligned} internal_i(m_k) = \quad & E_i + DB_i + Ihp_i^{(dsr)} + DB_i^G + Ihp_i^{(dsgr)} \\ & + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i + Ilp_i^{(gr)} \end{aligned} \tag{10.33}$$

Lemma 3. *In a component C_k, the maximum response time of a task τ_i sharing global resources cannot further be decreased by increasing m_k if $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$, i.e., the requirement extracted from τ_i (Equation 10.31) cannot be further relaxed.*

Proof. As it can be seen in Equation 10.32, if condition $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$ holds, $internal_i(m_k)$ cannot further be reduced even if m_k is increased since none of its terms are dependant on the number of processors that C_k is allocated on, i.e., m_k. Thus the maximum response time of τ_i (Equation 10.29) cannot be decreased any more which leads to that the requirement extracted from τ_i (Equation 10.31) cannot further be relaxed by increasing m_k. □

10.6.2 Determine Minimum and Maximum Required Processors

In this section we describe a method to determine $m_k^{(min)}$ and $m_k^{(max)}$ for component C_k in its interface.

$m_k^{(min)}$ is the minimum number of processors required by C_k such that it is schedulable. Obviously $\lceil U_k \rceil \leq m_k^{(min)}$ where $\lceil U_k \rceil = \sum_{\tau \in \tau_{C_k}} u_i$.

Theorem 1. *Under C-MSOS, the minimum number of required processors for component C_k to be schedulable, can be achieved by setting $RB_i = 0$ for any task τ_i sharing global resources, and is calculated as follows:*

$$m_k^{(min)} = \min_{\substack{m_x \geq \lceil U_k \rceil \\ \wedge\, C_k \textit{ is schedulable on } m_x}} \{m_x\} \tag{10.34}$$

Proof. Setting $RB_i = 0$ means that τ_i does not need to tolerate any remote blocking, i.e., in the best case its component is not co-executing with any component that shares its global resources. On the other hand looking at the calculations of the rest of the terms contributing to τ_i's worst-case response time RT_i in Section 10.5, all the terms that are dependent on m_k, e.g., Ilp_i, $Ilp_i^{(gr)}$, monotonically decrease when m_k increases, and consequently RT_i monotonically decreases by increasing m_k. This means that increasing m_k favors response times. Thus, starting from $\lceil U_k \rceil$ the first m_k on which C_k is schedulable will be $m_k^{(min)}$. □

$m_k^{(max)}$ is the maximum number of processors required for C_k to be schedulable, i.e., further increasing the number of processors for C_k does not favor the schedulability of any component. In a component C_k the tasks that do not share any global resources do not benefit (from the schedulability point of view) from increasing the number of processors from $m_k^{(min)}$ since these tasks are already schedulable on $m_k^{(min)}$ processors. However, as shown in Equation 10.31, for any task τ_i sharing global resources, the requirement extracted from τ_i will be relaxed by increasing m_k, i.e., τ_i can tolerate more remote blocking (from other components) which benefits other components sharing global resources with τ_i. Thus $m_k^{(max)}$ is the maximum number of processors where at least one requirement in $Q_k(m_k)$ is relaxed, i.e., allocating C_k on m_h where $m_h > m_k^{(max)}$ does not further relax any requirement hence no component will benefit from C_k being allocated on m_h compared to the case where C_k is allocated on $m_k^{(max)}$.

Theorem 2. *Under C-MSOS,* $m_k^{(max)} = |\tau_H(\tau_{min})| + 1$*, where* τ_{min} *is the task with minimum priority among all tasks sharing any global resources.*

Proof. As shown by Lemma 3, the requirement extracted from a task τ_i sharing any global resources, cannot further be relaxed by increasing m_k, if $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$. When increasing m_k, as soon as all tasks sharing global resources are among the m_k highest priority tasks, condition $|\tau_H(\tau_i)|+|\tau_L^G(\tau_i)| < m_k$ will hold for all of them. This is because if a task τ_i (that shares any global resources) is among the m_k highest priority tasks any task in $\tau_H(\tau_i)$ will also be among them. Furthermore since all tasks sharing global resources are among the m_k highest priority tasks any task in $\tau_L^G(\tau_i)$ as well as τ_i itself are also among them, thus $|\tau_H(\tau_i)| + |\tau_L^G(\tau_i)| < m_k$ holds for all these tasks. Hence, by definition $m_k = m_k^{(max)}$. On the other hand, as m_k is increased, the last task sharing global resources that becomes one of the m_k highest priority tasks will be τ_{min}. As soon as τ_{min} belongs to m_k highest priority tasks, these m_k tasks will only consist of all tasks in $\tau_H(\tau_{min})$ and τ_{min} itself. Thus $m_k^{(max)} = |\tau_H(\tau_{min})| + 1$. □

10.7 Minimizing the Number of Required Processors for all Components

In this section we will show that using the information in the interfaces of components the integration of all the real-time components on a multiprocessor platform can be formulated as a Nonlinear Integer Programming (NIP) problem [20]. Formulating the integration phase as a NIP problem facilitates using the techniques in this domain [20] for minimizing the total number of required processors by all components.

A typical model of a NIP problem is represented as follows:

For n number of integer variables $x_1, \cdots, x_n$, there is an objective function $f: R^n \longrightarrow R$ to be minimized (or maximized):

$$Minimize \;\; f(x_1, \cdots, x_n) \tag{10.35}$$

With a set of (nonlinear) inequality constraints g and a set of (nonlinear) equality constraints h formed as follows:

$$\begin{array}{l} g_i(x_1, \cdots, x_n) \leq b_i, \;\; i = 1, \cdots, i = p \\ h_j(x_1, \cdots, x_n) = c_j, \;\; j = 1, \cdots, i = q \end{array} \tag{10.36}$$

If the objective function f or some of the constraints g_i are nonlinear, the problem is a NIP problem. An optimal solution $(\bar{x}_1, \cdots, \bar{x}_n)$ is a solution for which all constraints hold and the output of the objective function is minimized.

Our goal is to minimize the number of total required processors by all components in the integration phase. Thus, assuming that there are n components to be integrated on a multiprocessor platform, the objective function will be formed as follows:

$$Minimize \;\; f(m_1, \cdots, m_n) = m_1 + \cdots + m_n \tag{10.37}$$

where m_i is the number of processors that C_i will eventually be allocated on.

We rewrite the model of a requirement r_i (Equation 10.31) in the requirement set $Q_k(m_k)$ of a component C_k:

$$\sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \leq D_i - internal_i(m_k)$$

It can be shown that by replacing the terms of $internal_i(m_k)$ with their calculations from the corresponding equations in Section 10.5, it can be simplified as follows:

$$internal_i(m_k) = \beta_i + \frac{\delta_i}{m_k} \tag{10.38}$$

where β_i and δ_i are constant numbers which means that they only depend on the internal parameters of C_k.

Thus we can rewrite requirement r_i as follows:

$$\sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} RWT_{q,k} \leq d_i - \frac{\delta_i}{m_k} \tag{10.39}$$

where $d_i = D_i - \beta_i$.

As shown in Equation 10.15, $RWT_{q,k}$ is the summation of $Z_{q,s}(m_s)$'s ($s \neq k$) and $Z_{q,s}(m_s)$'s in turn as shown in Equation 10.16 (we consider FIFO global queues without loss of generality) depend on RHT's. Furthermore, similar to the simplification of $internal_i(m_k)$ in Equation 10.38, the calculation of $RHT_{q,s,i}(m_s)$ can be simplified as follows:

$$RHT_{q,s,i}(m_s) = \sigma_i + \frac{\gamma_i}{m_s} \tag{10.40}$$

where $\sigma_i = Cs_{i,q}$ and $\gamma_i = \sum_{\tau_j \neq \tau_i} \Big(\max_{\substack{R_l \in R^G_{C_s},\, l \neq q \\ \wedge\ \tau_j \in \tau_{l,s}}} \{Cs_{j,l}\} \Big)$ which are also constants.

Thus by combining Equations 10.15, 10.16, and 10.40, we can rewrite Equation 10.39 for FIFO queues as follows (a similar equation can be achieved for Round-Robin queues by combining Equations 10.15, 10.17, and 10.40):

$$\sum_{\substack{R_q \in R^G_{C_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} \alpha_{i,q} \sum_{l \neq k} \sum_{\substack{\forall \tau_i,\ \tau_i \in \tau_{q,l} \\ \wedge\ R_q \in R^G_{C_l}}} (\sigma_i + \frac{\gamma_i}{m_l}) \leq d_i - \frac{\delta_i}{m_k} \tag{10.41}$$

Finally, it is easy to see that Equation 10.41 can be rewritten in the following form:

$$\sum \frac{c_l}{m_l} \leq b_i \tag{10.42}$$

The requirement in Equation 10.42 is a nonlinear inequality constraint which adheres to the form of constraint for a NIP problem (Equation 10.36). Thus every requirement in $Q_k(m_k)$ of every component C_k will generate a nonlinear inequality constraint. Furthermore, every component C_k generates the inequality constraint $m_k^{(min)} \leq m_k \leq m_k^{(max)}$ which can be divided into two inequalities $m_k \geq m_k^{(min)}$ and $m_k \leq m_k^{(max)}$. Obviously $m_1, \cdots$, and m_k are integers, thus the integration of the real-time components on a multiprocessor platform under C-MSOS can be modeled as a NIP problem.

10.8 Evaluation

In this section we present our evaluations. In the first part we have evaluated C-MSOS and its different alternatives regarding queue handling. In the second part we have studied the practicality of using NIP methods for minimizing the number of processors required by the components.

10.8.1 Simulation-based Evaluation of C-MSOS

We have performed simulation-based evaluation to investigate the performance of C-MSOS for its four different alternatives where global queues of global resources are FIFO-based or Round-Robin-based and the local queues are FIFO-based or Priority-based.

Experimental Setup

To determine the performance of all four alternatives we tested the schedulability of a set of randomly generated components on a multiprocessor platform under each alternative and according to different settings. For each setting, the number of components was varied from 2 to 22, and each component was allocated on 3 or 5 processors (processors per component). The number of components sharing each global resource was chosen between 2 and 12 (components per resource), and the number of tasks per each component sharing a global resource was varied from 2 to 12 (tasks per component per resource). For each component a task set was randomly generated where the utilization of each task was randomly chosen between 0.01 and 0.1, and its period was randomly chosen between $10ms$ and $100ms$, and the execution time of the task was calculated based on its utilization and period. For each component, tasks were generated until the utilization of the component reached a cap or a maximum number of 30 tasks were generated. The utilization cap of a component was set to be the number of processors of the component multiplied by 0.4. A task included up to 4 critical sections, and the total number of shared global resources was 8 or 16. The length of global critical sections ranged from $10\mu s$ to $150\mu s$. For each setting we generated 1000 platforms.

Results

First we performed the experiments for the platforms that consisted of similar components, e.g., all the components sharing the global resources had the same number of tasks sharing each global resource (the number of tasks per component per resource were the same), etc. The performance of C-MSOS for its different alternatives, with regard to the number of components on the platform, the number of components sharing each resource, the number of tasks per component sharing each resource, and the length of critical sections per task, is illustrated in Figs. 10.3, 10.4, 10.5, and 10.6 respectively. In this case (where the components are similar), the overall results show that C-MSOS mostly performs better if the local queues are FIFO-based. When using FIFO-based local queues, C-MSOS performs similar for both FIFO-based and Round-Robin-based global queues. However, using prioritized local queues, C-MSOS mostly performs better by using Round-Robin-based global queues.

Second, we performed experiments where each generated platform consisted of components with different degree of resource sharing. This is closer to reality where components may differ in their settings, e.g., the number of tasks per component sharing a global resource can be different for different

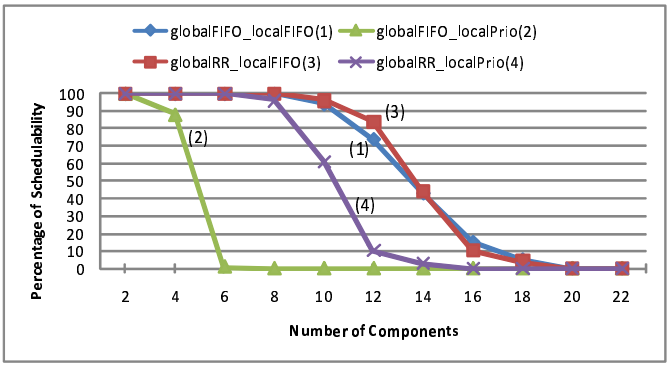


Figure 10.3: Schedulability of C-MSOS by increasing the number of components on the platform. 3 processor per component, 8 global resources each shared by half of the components from which 4 tasks share the resource, up to 4 critical sections per task each with length of 40 μs.

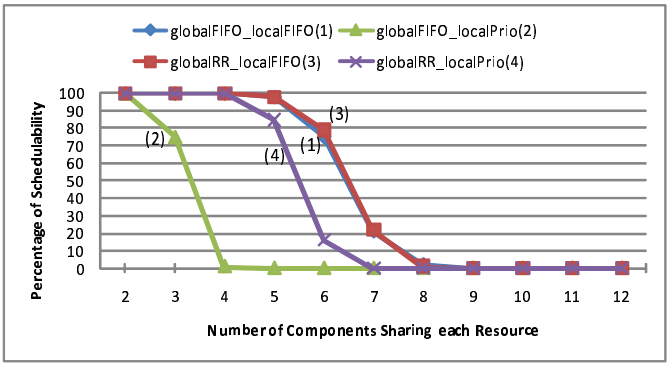


Figure 10.4: Schedulability of C-MSOS by increasing the number of components sharing each resource. 12 component, 3 processor per component, 8 global resources each shared, 4 tasks per component sharing a global resource, up to 4 critical sections per task each with length of 40 μs.

components. Looking at the schedulability analysis in Section 10.5.2, an important factor for which the different alternatives of C-MSOS may perform differently is the degree of resource sharing in each component, e.g., a component may benefit better under an alternative of C-MSOS depending on the

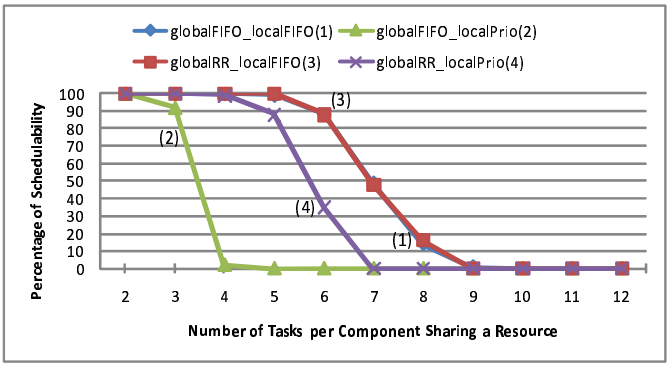


Figure 10.5: Schedulability of C-MSOS by increasing the number of tasks per component per resource. 12 component, 3 processor per component, 8 global resources each shared by 4 components, up to 4 critical sections per task each with length of 40 μs.

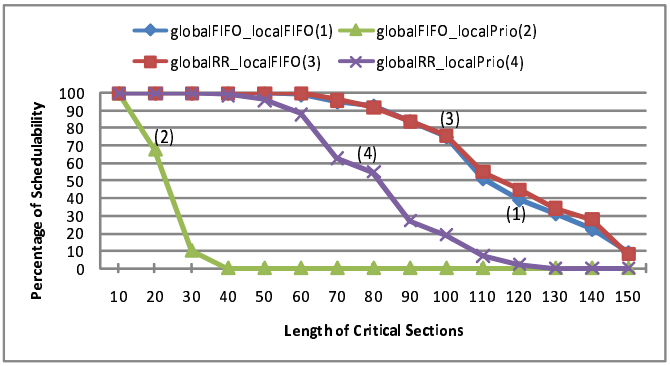


Figure 10.6: Schedulability of C-MSOS by increasing the length of critical sections (in μs). 12 component, 3 processor per component, 8 global resources each shared by 4 components from which 4 tasks share the resource, up to 4 critical sections per task.

number of its tasks that share each global resource. We performed experiments in which each generated platform consisted of components with different number of tasks sharing each global resource. We generated 1000 platforms consisting of 12 components. For each platform we divided its 12 components

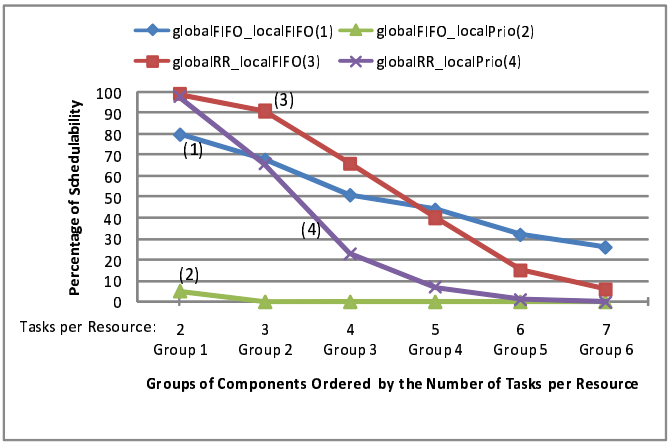


Figure 10.7: Schedulability under C-MSOS for components with different number of tasks per resource. 12 component, 3 processor per component, 8 global resources each shared by 6 components, up to 4 critical sections per task each with length of 80 μs.

into 6 groups (2 components per group); where beginning from the first group to the sixth group they included 2, 3, 4, 5, 6, and 7 tasks sharing each resource respectively. The results in Fig. 10.7 illustrates the average percentage of schedulable components of each group under different alternatives of C-MSOS. As shown in Fig. 10.7, for any type of components the alternative of C-MSOS where the global queues are FIFO-based and the local queues are priority-based (FP) is always outperformed by other alternatives. In fact this alternative (FP) was never better than any other alternatives for any settings. Furthermore, the components that share global resources, and include only 2 tasks per each shared global resource, perform better under the both alternatives that use Round-Robin-based global queues (RF and RP) compared to the alternative where both global and local queues are FIFO-based (FF). The alternative RF (Round-Robin global queues and FIFO local queues) performs better for the components that have less than 5 tasks per each global resource they share while FF (FIFO global queues and FIFO local queues) alternative performs better for the component with 5 and more tasks per each global resource they share. Alternative FF performs better than RP even for components with 3 and more tasks per each global resource the components share. Given

any type of global queues, all types of components benefit more from FIFO-based local queues rather than priority-based queues, i.e., FF and RF always outperform FP and RP respectively.

10.8.2 Practicality of Optimization of the Total Number of Processors Required by Components

Here we study how practical it is to optimize the number of processors needed for all the co-executing components while their schedulability is guaranteed.

Experimental Setup

In this experiment we used the optimization package in MATLAB and its solver for NIP problems. To investigate the duration of time that a relatively large problem takes, we randomly generated 100 platforms where each platform consisted of 20 components. The tasks of each component were randomly generated until either the total utilization of the component reached a cap equal to 1.2 or the number of tasks generated in the component reached 30. The utilization of each task was randomly chosen between 0.01 and 0.1, with its period randomly chosen between $1ms$ and $100ms$. A task included up to 2 critical sections, and the total number of shared global resources was 10. The number of components that shared each resource was 3 where each of the resources was used by 3 tasks in the component. The length of global critical sections was $40\mu s$.

Results

For each of the 100 generated platforms we programmatically formulated the integration of its components as a NIP problem as described in Section 10.7. We measured the time taken for the NIP solver in MATLAB on a normal personal computer to solve each of the formulated NIP problems.

Optimization of the number of required processors for each platform took less than 4 seconds, i.e., the average time taken to solve the problem for a platform was 3.2 seconds.

The measured time durations for optimizing the number of required processors shows that, for a platform with settings similar to the platforms in this experiment, it takes only several seconds. Considering that these settings can cover many complex real-time systems with similar settings, the amount of time taken to optimize the number of processors is practical.

As shown in the first part of the evaluations, i.e., evaluation of C-MSOS in Section 10.8.1, when the number of components of platforms reaches 20, almost none of the platforms can be schedulable by any alternative of C-MSOS. Thus, considering platforms with 20 components and the settings we used in this section can be categorized among the largest platforms that can be schedulable by C-MSOS. In fact, for platforms relatively larger than that, we doubt that any synchronization protocol may ever exist that can improve schedulability of the platforms significantly. This means that a NIP problem formulated from a typical platform with these settings can be categorized as a complex problem, i.e., to investigate the performance of NIP problem solvers with larger number of components and more complex settings is not necessary.

10.9 Summary and Conclusion

We have developed a new locking protocol (C-MSOS) to handle resource sharing among components where each component is statically allocated on multiple dedicated processors (one cluster). We have also assumed that the tasks within each component are scheduled using global fixed-priority preemptive scheduling policy.

In C-MSOS each component is abstracted and represented by an interface which abstracts the information about global resources it shares with other components. Furthermore, the interface includes a set of requirements that should be satisfied for the component to be schedulable when it co-executes with other components on a shared multi-core platform. This offers the possibility to develop different real-time components in parallel and independently and their schedulability analysis can be performed and abstracted in their interfaces. Furthermore, as most of scheduling analysis is performed offline and the resource requirements of components are abstracted as simple linear inequalities, the interface-based scheduling when the components are put together will be much easier. This offers the possibility of introducing the components during run-time in an open system, where an admission control program will perform better as it only needs to revalidate the requirements in the interfaces.

Furthermore, we have shown that the integration phase, where the components are allocated on a multiprocessor platform, can be formulated as a Non-Linear Integer Programming problem. Hence, the existing methods and solvers for NIP problems can be used to obtain the minimum number of processors required by the components where their schedulability is guaranteed.

We have investigated four alternatives for queue handling under C-MSOS;

the global queues associated with global resources may be either FIFO-based or Round-Robin-based, and the local queues associated with the global resources can be either FIFO-based or priority-based.

We have performed simulation-based evaluations for different alternatives of C-MSOS. We have also investigated the practicality of using NIP methods for optimization of the number of processors in the integration phase.

Our evaluation results show that using local FIFO queues to handle global resources almost always outperforms priority-based queues. The priority-based local queues perform better together with Round-Robin-based global queues compared to FIFO-based global queues. Using C-MSOS with priority-based local queues together with FIFO-based global queues turned out to be the worst choice among the four alternatives and performs very poor. In the case of using FIFO-based local queues, FIFO and Round-Robin global queues perform similar. However, they may perform differently depending on the number of tasks in each component using global resources, i.e., FIFO global queues favor the components with fewer tasks using each global resource while Round-Robin global queues perform better for the components with higher number of tasks using global resources.

Using NIP methods the total number of required processors can be minimized. However, when allocating the components on a multiprocessor it is better to allocate each component on the minimum possible number of processors even if there are enough processors available. Allocating a component on fewer processors (if it does not compromise the schedulability of any component) increases the performance, because inside the component tasks are using global scheduling and thus fewer processors means that there will be fewer migrations and consequently less migration overhead.

Regarding the practical size of platforms where C-MSOS protocol can be applicable, as shown by our experimental results, when the number of components in the platforms approaches 20, where each component is allocated on around 3 processors, i.e., the whole platform is allocated on around 60 cores, the platform can hardly be schedulable under C-MSOS. In fact, we believe that no synchronization protocol may survive for problems with this size, confirming that sharing mutually exclusive resources can become a bottleneck on taking advantage of performance offered by multi-cores. Thus, co-executing too many components, i.e., more than 20 components, on a shared multi-core platform where the components interfere relatively highly on global resources may kill any scheduling and synchronization protocol. Considering this limited size of platforms that C-MSOS can be applicable for, if a NIP solver can solve the optimization problem of integration phase for a platform with similar size

in reasonable time, can be considered as acceptable in practice. We measured the average time duration that a NIP solver in MATLAB may take to find a solution for a platform with this size. The average duration of time to find the minimum number of processors required by components of platform consisting of 20 components took a few seconds (less than 4 seconds). This can be acceptable in practice for finding a solution for a platform which is considered as large and complex.

In the future we plan to implement C-MSOS under real-time operating systems (RTOS) and study its performance. We also plan to study legacy real-time components and attempt to extract interfaces for them according to the interface model of C-MSOS. C-MSOS is based on shared memory synchronization, hence an interesting future work is to study resource sharing among real-time components on a multiprocessor platform by means of message passing approaches.

Bibliography

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[2] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[3] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of 19th Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[4] Theodore P. Baker and Sanjoy K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*, 2007.

[5] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of 23th Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 251–261, 2011.

[6] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[7] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *Proceedings of 7th ACM & IEEE International conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[8] G. Lipari and E. Bini. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In *Proceedings of 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 249–258, 2010.

[9] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of 20th Euromicro Conference on Real-time Systems (ECRTS'08)*, pages 181–190, 2008.

[10] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[11] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[12] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[13] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31st IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[14] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of 11th ACM and IEEE International Conference on Embedded Software (EMSOFT'11)*, pages 69–78, 2011.

[15] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[16] D. Faggioli, G. Lipari, and T. Cucinotta. The Multiprocessor Bandwidth Inheritance Protocol. In *Proceedings of 22th Euromicro Conference on Real-time Systems (ECRTS'10)*, pages 90–99, 2010.

[17] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper*.

[18] N. Fisher, M. Bertogna, and S. Baruah. Resource-Locking Durations in EDF-Scheduled Systems. In *Proceedings of 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, 2007.

[19] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'07)*, pages 149–160, 2007.

[20] D. Li and X. Sun. Nonlinear integer programming. Springer, 2006.

Chapter 11

Paper E: Resource Hold Times under Multiprocessor Static-Priority Global Scheduling

Farhang Nemati and Thomas Nolte
In 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pages 197-206, August, 2011.

Abstract

Recently there has been a lot of interest in coexisting of multiple independently-developed real-time applications on a shared open platform. On the other hand, emerging of multi-core platforms and the performance and possibilities they offer has attracted a lot of attention in multiprocessor real-time analysis, protocols and techniques. Co-executing independently-developed real-time applications on a shared multiprocessor system, where each application executes on a dedicated sub set of processors, requires to overcome the problem of handling mutually exclusive shared resources among those applications. To handle resource sharing, it is important to determine the Resource Hold Time (RHT), i.e., the maximum duration of time that an application locks a shared resource.

In this paper, we study resource hold times under multiprocessor static-priority global scheduling. We present how to compute RHT's for each resource in an application. We also show how to decrease the RHT's without compromising the schedulability of the application. We show that decreasing all RHT's for all shared resources is a multiobjective optimization problem and there can exist multiple Pareto-optimal solutions.

11.1 Introduction

The availability of multi-core platforms has attracted much attention in multiprocessor embedded software analysis, runtime techniques, policies, and protocols. As the multi-core architectures are to be the defacto processors within a near future, the industry must cope with a potential migration of existing systems towards multi-core platforms.

An important issue when migrating to multi-cores is to provide the possibility of several of independently-developed real-time applications to co-execute on a shared multi-core platform. In the context of uniprocessors, there has been much interest in developing support for independently-developed real-time applications to be executed in shared *open* environments. A non-exhaustive list of works in this domain includes [1, 2, 3, 4, 5, 6], and hierarchical scheduling has emerged as a common solution for open systems. An open environment requires the co-existence of multiple independently-developed real-time applications on a shared platform. The applications may have been developed using different techniques, e.g., it may be the case that different real-time applications that will coexist on a multi-core have different scheduling policies. Recently, in industry, co-existing of multiple applications on a multi-core platform (using virtualization techniques) has been considered to reduce the overall hardware costs [7].

On the other hand, looking at industrial systems, to speed up their development, it is not uncommon that large and complex systems are divided into multiple semi-independent subsystems, each of which is developed independently. The subsystems which may share resources will eventually be integrated and coexist on the same platform. This issue has got attention and has been studied over the years in the uniprocessor domain [8, 9, 10].

However, when the applications co-execute on the same multi-core platform they may share resources that require mutual exclusive access. An important issue to support resource sharing among independently-developed real-time applications on a shared multiprocessor platform, is to abstract the applications' resource requirements sufficiently such that the internal details of each application are hidden from other applications. To be able to handle resource sharing among such applications, in the abstraction of each application's Resource Hold Times (RHT's) should be specified. The RHT of a shared resource for an application is defined as the longest time interval in which the application may lock the resource. Determining resource hold times for each resource that is shared by an application is the objective of this paper.

Looking at the current state-of-the-art, there exist two major approaches

for scheduling real-time systems on multiprocessors (multi-cores); global and partitioned scheduling [11, 12, 13]. Under global scheduling, e.g., Global Earliest Deadline First (G-EDF), tasks are scheduled by a single (global) scheduler and each task can be scheduled to execute on any processor, i.e., migration of tasks among processors is permitted. Under partitioned scheduling, tasks are statically assigned to processors and tasks within each processor are scheduled by a uniprocessor scheduling protocol, e.g., Rate Monotonic (RM) or Earliest Deadline First (EDF). The generalization of global and partitioned scheduling algorithms is called clustered scheduling [14, 15], in which tasks are statically assigned to a sub set (a cluster) of processors, and within each cluster tasks are scheduled using a global scheduling algorithm.

In our previous work [16] we have studied resource sharing among independently-developed real-time applications where each application is allocated on one dedicated processor (core). However, in this paper we focus on the clustered scheduling policy where a real-time application will be allocated to a cluster (multiple processors) of a multi-core platform. Within an application tasks are scheduled to execute on any processor within the cluster using a global scheduling algorithm, i.e., jobs can migrate among the processors of their application according to the global scheduling policy in use.

11.1.1 Contributions

The concept of Resource Hold Time (RHT) for independently-developed applications on uniprocessor platforms was first presented by Fisher et al. in [17], and later by Bertogna et al. in [18]. In our previous work [16] we extended the notion of RHT to multiprocessors. However, in [16], firstly we assume that each application is fully allocated on one dedicated processor, and secondly within an application the priority of tasks holding a *global resource* (i.e., a global resource is shared among multiple applications) was always boosted to be higher than the priority of any task within the application.

In this paper, we further extend the notion of RHT to the independently-developed real-time applications where each application is allocated on multiple processors (a cluster), and given this setting we present an algorithm for computing resource hold times of the application. It is important that RHT's are as short as possible because shorter RHT's of global resources means that the applications will hold the resources in shorter intervals which leads to decreased interference among the applications sharing the resources. Although, boosting the priority of tasks holding global resources to the highest priority in the application will make the RHT's shorter, however, as we show in this paper

it may make the application unschedulable. Therefore to shorten the RHT's we assume that the priorities of tasks holding global resources are boosted only as long as the application remains schedulable, i.e., boosting the priorities should never compromise the schedulability of the application. Under uniprocessor platforms, it has been shown [17, 18] that it is possible to achieve one single optimal solution, when trying to decrease RHT for an application. However, in this paper we show that this is not the case when the application is scheduled on multiple processors (i.e., systems where tasks in the application are scheduled by a multiprocessor global scheduling policy) and there can in fact exist multiple Pareto-optimal solutions.

11.1.2 Related Work

In the context of independently-developed real-time applications on uniprocessors, a considerable amount of work has been done. A non-exhaustive list of works in this domain includes [1, 2, 3, 4, 5, 6], in which hierarchical scheduling has been studied and developed as a solution. Clustered scheduling techniques have been developed for multiprocessors (multi-cores) [14, 19]. However, the tasks allocated in each cluster are assumed to be independent and therefore those techniques do not allow for sharing of mutually exclusive resources. In the context of locking protocols under multiprocessors, there are several approaches [20, 21, 22, 23, 24, 25, 26, 27]. However, in all the existing synchronization protocols (under partitioned scheduling) on multiprocessors it is assumed that the tasks of a single real-time application can be distributed among all the processors, and all processors use the same scheduling policy.

Recently, Brandenburg and Anderson [28] presented a new locking protocol, called O(m) Locking Protocol (OMLP), which has variations for both global and partitioned scheduling. However, OMLP is an *suspension-oblivious* protocol. Under a suspension-oblivious locking protocol, the suspended jobs are assumed to occupy processors and thus blocking is counted as demand. To test the schedulability of the system, the worst-case execution times of tasks are therefore inflated with blocking times. In this paper we focus on *suspension-aware* locking synchronization in which suspended jobs are not assumed to occupy processors (i.e., no task inflation).

To our knowledge, the approach presented by Easwaran and Andersson [27] is the only work on handling suspension-aware resource sharing under global scheduling which provides a schedulability test. In the paper, the authors have derived a schedulability test for the Priority Inheritance Protocol (PIP) under static-priority global scheduling policy as well as for a new proposed locking

protocol (P-PCP). We use their scheduling analysis to compute resource hold times for global resources. However, in this paper, because of limited space and besides that the analysis of P-PCP is similar to PIP, we only assume using PIP for handling *local resources* (i.e., a local resource is only shared locally by tasks of one application).

The resource hold times for independently-developed applications on a shared uniprocessor platform was for the first time presented by Fisher et al. in [17]. In this work the authors have presented an algorithm to compute the resource hold time for each resource in an application, assuming that the local scheduling policy is EDF and that the resource sharing is handled by SRP (the Stack-based Resource allocation Protocol) [29]. They have further presented an algorithm to reduce the resource hold times without compromising the schedulability of the application. They have shown that one optimal solution can be achieved by using their reduction algorithm. Later, Bertogna et al. [18] extended this work to Static-Priority Scheduling (SPS).

Recently we have proposed a locking protocol [16] for handling resource sharing among independently-developed real-time applications on multiprocessors. In this work each application is represented by an interface which abstracts the resource requirements of the application. Furthermore, the notion of RHT was extended to multiprocessors. However, we assumed that each application should be allocated to one dedicated processor. Besides, to reduce the resource hold times of global resources, the priority of tasks holding a global resource is raised (boosted) to be higher than the highest priority in the application. Boosting the priorities of tasks when they are granted access to global resources has been the case in all existing locking protocols to handle access to global resources under partitioned scheduling protocols. Raising the priority of tasks holding global resources to the highest priority in an application will reduce the RHT but it may make the application unschedulable. In this paper the RHTs are reduced by means of boosting the priorities of tasks holding global resources as far as possible, i.e., as long as the application remains schedulable. This means that the priorities of tasks holding global resources are boosted without compromising the schedulability of the application.

11.2 System and Platform Model

In this paper we assume that a real-time application, denoted by A_k, is allocated on a dedicated cluster consisting of m identical, unit-capacity processors (cores). The real-time application consists of a set of real-time tasks. We

assume that the tasks in application A_k are scheduled using static-priority preemptive global scheduling (G-SPS). The jobs generated by tasks of the application can migrate among the processors within its cluster. However, migration of tasks across the clusters (applications) is not allowed.

Application A_k consists of a task set, $\tau(A_k)$, which includes a set of sporadic tasks, $\tau_i(T_i, E_i, D_i, \rho_i, \{Cs_{i,q,p}\})$ where T_i denotes the minimum inter-arrival time between two successive jobs of task τ_i with worst-case execution time E_i, relative deadline D_i and ρ_i as its unique *base priority*. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. In this paper, we assume constrained-deadline tasks (i.e., $D_i \leq T_i$ for any task τ_i). The priority of a job of a task may temporarily be raised by a synchronization protocol which is denoted as the *effective priority*. A job of task τ_i is specified by J_i.

The tasks in application A_k may share a set of mutually exclusive resources, R_{A_k}, which are protected using semaphores. The set of shared resources R_{A_k} consists of two sets of different types of resources; *local* and *global* resources. A local resource is only shared by tasks within the application while a global resource is shared by tasks from more than one application. The sets of local and global resources accessed by tasks in application A_k are denoted by $R^L_{A_k}$ and $R^G_{A_k}$ respectively. The set of critical sections, in which task τ_i requests resources in R_{A_k} is denoted by $\{Cs_{i,p,q}\}$, where $Cs_{i,q,p}$ is the the worst case execution time of p^{th} critical section of task τ_i in which the task uses resource $R_q \in R_{C_k}$. We define $Cs_{i,q}$ to be the worst case execution time of the longest critical section in which τ_i uses R_q. We also denote $CsT_{i,q}$ as the maximum total amount of time that τ_i uses R_q, i.e., $CsT_{i,q} = \sum Cs_{i,q,p}$. In this paper, we focus on non-nested critical sections.

11.3 Resource Sharing

The schedulability analysis of a real-time application in an open environment requires the global resource requirements of each application to be abstracted in its *interface*. Furthermore, the interface should also provide information about the maximum duration of time that each global resource can be held by the application. This time is denoted the resource hold time.

Definition 1: *Resource Hold Time* of a global resource R_q by task τ_i in application A_k is denoted by $\mathrm{RHT}_{q,k,i}$ and is the maximum duration of time that the global resource R_q can be locked by τ_i. Consequently, the resource hold time of a global resource, R_q, by application A_k (i.e., the maximum duration

of time during which R_q is locked by any task in A_k) is denoted by $\mathrm{RHT}_{q,k}$, and is derived as follows:

$$\mathrm{RHT}_{q,k} = \max_{\tau_i \in \tau_{q,k}} \{\mathrm{RHT}_{q,k,i}\} \tag{11.1}$$

where $\tau_{q,k}$ is the set of tasks in application A_k sharing R_q.

Among other parameters, computation of the resource hold times depend on the way an application handles resource sharing. In this paper we study resource hold times of global resources for a real-time application assuming that the application uses the global static-priority preemptive scheduling policy for scheduling its tasks on processors within its dedicated cluster, together with the Priority Inheritance Protocol (PIP) to access local resources.

Usually in multiprocessor suspension-based locking protocols, under partitioned scheduling, the global resources are handled by priority boosting [21, 30, 28, 16]. This means that the priority of tasks on a processor granted to access a global resource is boosted to be higher than the highest priority of any other task allocated on the processor. The rationale behind this approach is to make the locking times of global resources as short as possible. This rule can be used for independently-developed real-time applications where each application is allocated on a cluster of processors. However, boosting priorities to be higher than any base priority may lead to a situation in which some tasks will miss their deadlines and the application becomes unschedulable. Thus, we assume that the priorities of jobs that are granted access to a global resource R_q are raised (boosted) to a *boost level* without compromising schedulability of the application. We denote the boost level of any resource R_q by $boost_q$, i.e., the priority of any job J_i that is granted access to R_q is immediately raised to $boost_q$. In this paper, for each global resource shared by application A_k, we determine a range of valid values for R_q's boost level. First we review the characteristics of PIP for multiprocessors and its schedulability analysis.

11.4 PIP on Multiprocessors

Assuming that a task set is scheduled on a multiprocessor consisting of m processors, and that shared resources are handled by PIP, we here give an overview of two variants of PIP:

Basic PIP (B-PIP): While a job J_j accesses a resource, R_q, the job's effective priority is raised to the highest priority of any job waiting for R_q if there is any otherwise J_j executes with its base priority.

Immediate PIP (I-PIP): Whenever a job J_j is granted access to a resource, R_q, the job's effective priority is *immediately* raised (boosted) to the highest priority of any task that may request R_q.

The highest priority of any task that may request R_q is denoted by $\lceil R_q \rceil$. Under I-PIP, it can be stated that $\forall\, R_q \in R_{A_k},\ boost_q = \lceil R_q \rceil$.

Under PIP (in both variants, i.e., B-PIP and I-PIP), whenever a job J_i is waiting for a resource which is locked by a lower (base) priority job, J_j, and if J_i is among the m highest priority jobs, J_i is said to be *directly blocked* [27] by J_j.

Besides direct blocking, a job J_i can also incur interference from other lower priority jobs whose effective priorities have been raised at least as high as J_i's priority. Furthermore, J_i may incur extra interference from higher priority jobs when they have locked a resource that J_i has requested and J_i is among the m highest priority jobs.

11.4.1 Schedulability Analysis of B-PIP

In this section we review the schedulability analysis of PIP (B-PIP) as described by Easwaran and Andersson in [27]. Furthermore we extend the analysis to I-PIP by a simple change in their analysis for B-PIP. They have shown that under multiprocessor PIP the response time of any task τ_i denoted by RT_i can be calculated as follows:

$$RT_i = E_i + DB_i + Ihp_i^{(dsr)} + Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i \qquad (11.2)$$

where:

- DB_i upper bounds the direct blocking that τ_i incurs,
- $Ihp_i^{(dsr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i lock resources shared by τ_i (*direct shared resources*),
- $Ihp_i^{(osr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i may lock resources not shared by τ_i (*other shared resources*),

- $Ihp_i^{(nsr)}$ is an upper bound for the amount of time that tasks with a higher base priority than τ_i execute in their non-critical sections, i.e., they do not hold any resource (*no shared resource*),
- Ilp_i upper bounds the amount of time that tasks with a lower base priority than τ_i execute with a higher effective priority than τ_i.

In the same paper [27] the authors have further improved the computation of the response time for m highest priority tasks:

$$RT_i = \begin{cases} E_i + DB_i + Ihp_i^{(dsr)} & |\tau_H(\tau_i)| < m \\ E_i + DB_i + Ihp_i^{(dsr)} & \\ +Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp_i & \text{Otherwise} \end{cases} \tag{11.3}$$

where $|\tau_H(\tau_i)|$ is the number of tasks with priority higher than that of τ_i.

Upper Bound the Workload of a Task

To upper bound the worst-case interference from any task τ_j to task τ_i in any time interval t (e.g., RT_i), Easwaran and Andersson have presented a worst case execution pattern [27]. In this pattern, during the interval t, the carry-in job of τ_j executes as late as possible and all following jobs execute as early as possible. This pattern was first proposed by Bertogna and Cirinei [31] and was later extended by Easwaran and Andersson to maximize the total interference from a *certain portion* x (e.g., critical sections) of execution time of any job τ_j to τ_i in RT_i. In the extended pattern, x time units of execution time of the carry-in job appears as late as possible and the x time units of execution time of all the following jobs (of τ_j in interval RT_i) appear as early as possible (Figure 11.1). In this worst-case execution pattern Easwaran and Andersson have shown [27] that in any interval t the total maximum execution (i.e., workload) of x units of jobs of any task, τ_j is maximized as follows:

$$W_j(t,x) = xN_j(t,x) + \min\{x, t - x + D_j - T_jN_j(t,x)\} \tag{11.4}$$

where $N_j(t,x) = \left\lfloor \frac{t-x+D_j}{T_j} \right\rfloor$

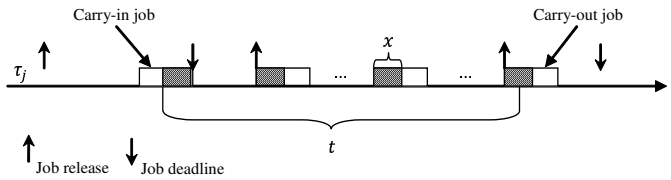


Figure 11.1: Worst-case execution pattern regarding giving importance to a certain portion of execution time

Based on this worst-case execution pattern, Easwaran and Andersson have calculated DB_i, $Ihp_i^{(dsr)}$, $Ihp_i^{(osr)}$, $Ihp_i^{(nsr)}$ and Ilp_i as follows (for details about how the equations are achieved please read [27]):

$$DB_i = \sum_{\substack{R_q \in R_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k}}} n_{i,q} \max_{\substack{\rho_j < \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} \{Cs_{j,q}\} \tag{11.5}$$

where $\tau_{q,k}$ is the set of tasks in application A_k sharing R_q, and $n_{i,q}$ is the number of critical sections of τ_i in which it accesses R_q.

$$Ihp_i^{(dsr)} = \sum_{\rho_j > \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R_{A_k} \\ \wedge\ \{\tau_i, \tau_j\} \subset \tau_{q,k}}} CsT_{j,q}\Big) \tag{11.6}$$

$$Ihp_i^{(osr)} = \frac{\sum_{\rho_j > \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R_{A_k} \\ \wedge\ \tau_j \in \tau_{q,k} \\ \wedge\ \tau_i \notin \tau_{q,k}}} CsT_{j,q}\Big)}{m} \tag{11.7}$$

$$Ihp_i^{(nsr)} = \frac{\sum_{\rho_j > \rho_i} W_j\Big(RT_i, E_j - \sum_{\substack{R_q \in R_{A_k} \\ \wedge\ \tau_j \in \tau_{q,k}}} CsT_{j,q}\Big)}{m} \tag{11.8}$$

$$Ilp_i = \frac{\sum_{\rho_j < \rho_i} W_j\Big(RT_i, \sum_{\substack{R_q \in R_{A_k} \\ \wedge\ \tau_j \in \tau_{q,k} \\ \wedge\ boost_q > \rho_i}} CsT_{j,q}\Big)}{m} \tag{11.9}$$

where $boost_q = \lceil R_q \rceil = \max_{\tau_i \in \tau_{q,k}} \{\rho_i\}$.

11.4.2 Extending Schedulability Analysis to I-PIP

Although Easwaran and Andersson in [27] have derived the aforementioned analysis for B-PIP, however, looking at the schedulability analysis, it is straight forward to draw the conclusion that from a schedulability analysis point of view the terms DB_i, $Ihp_i^{(dsr)}$, $Ihp_i^{(osr)}$ and $Ihp_i^{(nsr)}$ in the response time of task τ_i are the same for both variants (B-PIP and I-PIP).

The only difference is in the calculation of term Ilp_i. The difference is regarding to the tasks with priorities lower than τ_i that share any resource, R_q, with τ_i such that $boost_q = \rho_i$, i.e., τ_i is the highest priority task that may access R_q. Under B-PIP, interference from these tasks when they are accessing R_q, is not considered in Ilp_i because their effective priority is only raised to $boost_q$ if they (direct) block τ_i on R_q. However, under I-PIP the effective priority of these tasks is immediately raised to $boost_q$ whenever they are granted access to R_q, hence τ_i may suffer from their interference even if τ_i executes in non-critical sections. Assuming that tie-breaker is applied in first-come first-served manner, these lower priority tasks can only delay τ_i if they hold R_q before τ_i arrives, i.e., only one critical section of such tasks in which they access R_q can delay the execution of τ_i. If at some time instant the number of these tasks is less than m, and there is a free processor they do not delay the execution of τ_i. Thus, to extend the response time calculation of τ_i (Equation 11.2), Ilp_i has to be replaced by Ilp'_i where Ilp'_i is calculated as follows.

$$Ilp'_i = Ilp_i + \frac{\sum\limits_{\substack{R_q \in R_{A_k} \\ \wedge\ \tau_i \in \tau_{q,k} \\ \wedge\ boost_q = \rho_i}} \max\limits_{\substack{\rho_j < \rho_i \\ \wedge\ \tau_j \in \tau_{q,k}}} \{Cs_{j,q}\}}{m} \tag{11.10}$$

The improved response times for m highes priority tasks is also different for I-PIP. Looking at the rationale explained in [27] to derive the improved response times for such tasks (i.e., $|\tau_H(\tau_i)| < m$) it is only valid for B-PIP. Under I-PIP, the effective priority of a task accessing a resource is immediately raised to the highest priority of any task that may request the resource. Thus, when at some time instant a job of a task, τ_i where $|\tau_H(\tau_i)| < m$, (say J_i) is arrived, it may happen that there are more than m jobs executing with their effective priority higher than τ_i's base priority, some of which with base priorities lower than the priority of τ_i. In this case, τ_i may suffer from interference from those lower priority tasks. This interference for B-PIP is specified by Ilp_i, hence to extend the improved response time analysis to I-PIP the Equation 11.3 has to be rewritten as follows:

$$RT_i = \begin{cases} E_i + DB_i + Ihp_i^{(dsr)} + Ilp'_i & |\tau_H(\tau_i)| < m \\ E_i + DB_i + Ihp_i^{(dsr)} & \\ +Ihp_i^{(osr)} + Ihp_i^{(nsr)} + Ilp'_i & \text{Otherwise} \end{cases} \tag{11.11}$$

11.5 Computing Resource Hold Times

In this section, assuming that a real-time application A_k is schedulable under the static-priority global scheduling policy and PIP, we determine the computation of resource hold times of each global resource for A_k. In the existing synchronization protocols under multiprocessors, e.g., MPCP [21], the common case is that the priority of tasks locking global resources is boosted immediately, in this paper similarly we focus on the immediate priority boosting (the I-PIP variant). However, computing resource hold times under B-PIP can be derived similar to that under I-PIP.

Hereafter, we assume that the boost level for any global resource R_q is set such that application A_k is still schedulable and the following condition is satisfied:

$$\forall\, R_q \in R_k^G,\ boost_q \geq \lceil R_q \rceil \tag{11.12}$$

Theorem 1. *Assuming that A_k is schedulable under PIP, there is at least one setting for the boost level of any global resource (shared by A_k) that satisfies Condition 11.12 without making A_k unschedulable.*

Proof. Since it is assumed that A_k is schedulable under PIP, setting the boost level of any global resource R_q to $\lceil R_q \rceil$ (i.e., $boost_q = \lceil R_q \rceil$) does not change the semantics of the application, which means that all resources (local and global) are accessed using PIP (I-PIP) and the application will still remain schedulable. □

However, assigning the boost level of a global resource R_q to $\lceil R_q \rceil$ (i.e., $boost_q = \lceil R_q \rceil$) may cause a job J_i holding R_q to be delayed by other jobs (not executing in global critical sections) with effective priorities higher than $boost_q$ and thus the resource hold time of R_q will become longer. In this paper, we will show how to decrease the resource hold time of a global resource by increasing the boost level of the resource.

As shown in Equation 11.1 the resource hold time of a resource in an application is the longest resource hold time among all tasks sharing the resource.

Thus we describe how to compute the maximum duration of time that any τ_i can lock a global resource R_q, i.e., $RHT_{q,k,i}$.

When a job of task τ_i (say J_i) holds the lock of a global resource R_q, its priority is immediately raised to $boost_q$. The execution of J_i can then be delayed by any other job generated by any other task that belongs to at least one of the following three categories:

Category 1: The first category represents the set of tasks with base priority higher than or equal to $boost_q$. The jobs generated by these tasks can delay the execution of J_i when it is holding R_q.We denote $Rh_{q,i}$ as an upper bound for the maximum cumulative execution of those jobs, while J_i holds the lock of R_q. However, if at some point of time the number of these jobs is less than m, and there is a free processor, they do not delay the execution of J_i. Thus to calculate $Rh_{q,i}$, the workload of the jobs generated by tasks in this category is divided by m. To upper bound the total worst case workload of any task τ_x in this category, during the interval $RHT_{q,k,i}$, we use the execution pattern in Figure 11.1 and the definition of the worst case workload in Equation 11.4. Please note that any job J_x generated by any task in this category can delay the execution of J_i for the whole duration of its execution time (i.e. E_x). Thus we can compute $Rh_{q,i}$ as follows:

$$Rh_{q,i} = \frac{\sum\limits_{\substack{\rho_x \geq boost_q \\ \wedge\ x \neq i}} W_x\big(RHT_{q,k,i}, E_x\big)}{m} \tag{11.13}$$

Category 2: The second category represents the set of tasks with priorities lower than $boost_q$, whose generated jobs may hold any local resource R_p where $\lceil R_p \rceil \geq boost_q$. In this case these generated jobs may delay the execution of J_i while J_i holds R_q since their effective priority is at least as high as J_i's boosted priority. The upper bound for the maximum cumulative execution (workload) of these jobs when they hold R_p during the interval that J_i holds R_q is denoted by $Rl_{q,i}$. Similar to the previous category, if the number of such jobs at some time instant is less than m and all processors are not busy, they do not interfere with J_i. Hence to calculate $Rl_{q,i}$, the maximum cumulative execution of the jobs in their local critical sections in which they hold local resource R_p such that $\lceil R_p \rceil \geq boost_q$ should be divided by m. To upper bound the total worst case workload of local critical sections of any task τ_x in which τ_x holds any local resource R_p where $\lceil R_p \rceil \geq boost_q$, during interval $RHT_{q,k,i}$, we

use the execution pattern in Figure 11.1 and the definition of the worst case workload in Equation 11.4 (similar to the first category). Hence $Rl_{q,i}$ can be computed as follows:

$$Rl_{q,i} = \frac{\sum\limits_{\substack{\rho_x < boost_q \\ \wedge\ x \neq i}} W_x\Big(RHT_{q,k,i}, \sum\limits_{\substack{R_l \in R^L_{A_k} \\ \wedge\ \lceil R_l \rceil \geq boost_q \\ \wedge\ \tau_x \in \tau_{l,k}}} Cs_{x,l}\Big)}{m} \tag{11.14}$$

Category 3: The third category represents the set of tasks with priorities lower than $boost_q$, whose generated jobs hold the lock of any global resource R_l other than R_q with a boost level higher than or equal to R_q's boost level, i.e., $boost_l \geq boost_q$. These jobs holding R_l may delay the execution of J_i while J_i holds R_q because they have a boosted priority at least as high as J_i's boosted priority. However, these jobs executing in their global critical sections (in which they hold global resources with boost level higher than or equal to $boost_q$) will not delay the execution of J_i if the number of such jobs (at some time instant) is less than m and not all processors are busy. Thus, the maximum cumulative execution (workload) of these jobs while holding global resources with boost level higher than or equal to $boost_q$, should be divided by m. We denote $Rb_{q,i}$ as an upper bound of the workload of these jobs during the interval that J_i holds the lock for R_q, i.e., $RHT_{q,k,i}$. We can compute Rb_i (similar to computing $Rh_{q,i}$ and $Rl_{q,i}$) as follows:

$$Rb_{q,i} = \frac{\sum\limits_{\substack{\rho_x < boost_q \\ \wedge\ x \neq i}} W_x\Big(RHT_{q,k,i}, \sum\limits_{\substack{R_l \in R^G_{A_k} \\ \wedge\ boost_l \geq boost_q \\ \wedge\ \tau_x \in \tau_{l,k}}} Cs_{x,l}\Big)}{m} \tag{11.15}$$

11.5.1 Resource Hold Time Calculation

J_i itself will hold R_q for at most $Cs_{i,q}$ time units. Thus the maximum duration of time that any job of τ_i can lock R_q, i.e., $RHT_{q,k,i}$, can be computed as follows:

$$RHT_{q,k,i} = Cs_{i,q} + Rh_{q,i} + Rl_{q,i} + Rb_{q,i} \tag{11.16}$$

11.6 Decreasing Resource Hold Times

Since our focus in this paper is on independently-developed real-time applications on a shared multiprocessor platform, it is important to reduce the interference among the applications while they co-execute on the multiprocessor platform. Each application will be allocated on a dedicated cluster, thus they do not share processors, however, they share other resources and they interfere with each other by sharing the (global) resources. Hence, decreasing the resource hold times will reduce the interference among those applications.

11.6.1 Decreasing Resource Hold Time of a Single Global Resource

In an application A_k, for a given global resource R_q, we describe in this section how to reduce the resource hold time for the resource, i.e., $RHT_{q,k}$.

We suppose that R_q is held by a task, τ_i. As shown in Section 11.5, τ_i (while holding R_q) can be delayed by three categories of tasks. Looking at the upper bounds for the portions of the execution of those tasks that may delay τ_i (Equations 11.13, 11.14, and 11.15), it can be shown that the delay from these tasks is decreased as the boosting level of R_q (i.e., $boost_q$) is increased. In the extreme case if the boosting level of R_q is increased such that

$$boost_q > \rho_{max}, \text{ where } \rho_{max} = \max_{\tau_i \in \tau(A_k)} \{\rho_i\} \wedge \forall R_l \neq R_q, boost_q > boost_l$$

then

$$RHT_{q,k,i} = Cs_{i,q}$$

This means that if the boosting level of R_q is higher than any boosting level of any other (global) resource as well as any priority of any task in application A_k, then a task holding R_q will not be delayed by any task in A_k.

Thus, to minimize $RHT_{q,k}$, the boosting level of R_q (i.e., $boost_q$) has to be as high as possible without making application A_k unschedulable. The pseudo code of a simple algorithm for increasing the boosting level of a global resource R_q is shown in Figure 11.2. The algorithm initiates by assigning the boosting level of R_q to its minimum value $boost_q = \lceil R_q \rceil$. This initiation does not compromise the schedulability of application A_k (Theorem 1).

```
1 ceil_q ← max_{τ_i ∈ τ_{q,k}}{ρ_i};
2 ρ_max ← max_{τ_i ∈ A_k}{ρ_i};
3 for b = ceil_q + 1 to ρ_max + 1
4    boost_q ← b;
5    if A_k is NOT schedulable then
6       boost_q ← b - 1;
7       return;
8    end if
9 end for
```

Figure 11.2: Increasing the boosting level for R_q in A_k

11.6.2 Decreasing Resource Hold Time of all Global Resources

In this section we describe how to decrease the resource hold times for all global resources shared by a real-time application allocated on a sub set (cluster) of processors of a multiprocessor platform.

On uniprocessor platforms, Fisher et al. [17] has shown how their algorithm for decreasing the resource hold time of a single resource is used to reduce the resource hold time for all shared resources. They have shown that under EDF and SRP the order in which they apply the algorithm to each resource has no effect on minimizing the resource hold times, i.e., the minimum possible resource hold time of a resource by their algorithm is not influenced by if the algorithm has been called for another resource previously. In a later work Bertogna et al. [18] have showed that under SPS (static priority scheduling) and SRP, their algorithm for reducing resource hold time for a single task has an optimal solution when it is used to minimize the resource hold times for all resources. The optimal solution is achieved if the algorithm is called for the resources in a specific order, i.e., if the algorithm is called for the resources in the order of the length of the maximum critical section in which each resource is accessed.

However, in the context of a multiprocessor system (static-priority global scheduling and PIP in this paper), there can be more than one optimal solution. Depending on the order in which the algorithm in Figure 11.2 is called for the global resources in application A_k, the resulting boosting levels for the resources may differ. There can be several *Pareto-optimal* allocations of boost-

ing levels to global resources. A Pareto-optimal allocation in the context of maximizing of the boosting levels of global resources refers to an allocation in which the boosting level of no global resource can be further increased without decreasing the boosting level for any other global resource. In the illustrative example in Section 11.7 we show that there can exist several Pareto-optimal allocations of boosting levels for the global resources.

11.7 An Illustrative Example

In this section we illustrate how to decrease the resource hold time for each global resource by increasing its boosting level using the algorithm presented in Section 11.6. We further show that there can be more than one Pareto-optimal allocation of boosting levels for global resources. We assume that the real-time application is allocated on a cluster consisting of two processors ($m = 2$), and that the application is compromised of the following task set:

Table 11.1: Task set.

τ_i	E_i	D_i	T_i	ρ_i
τ_1	2	3	8	4
τ_2	6	16	16	3
τ_3	10	30	32	2
τ_4	3	28	32	1

There are two global resources accessed by the tasks in the application. Task τ_2 accesses global resource R_1 for 2 time units, and global resource R_2 is accessed by τ_3 for 2 time units. Thus $\lceil R_1 \rceil = 3$ and $\lceil R_2 \rceil = 2$.

11.7.1 Testing the Schedulability

We test the schedulability by investigating three cases: (1) setting the boosting levels to their minimum values, (2) setting the boosting values to maximum (i.e., higher than any priority in the application), and (3) setting boosting levels using the algorithm that we have presented in Section 11.6.1.

Setting the Boosting Levels to Minimum

Here we set the boosting levels to their minimum values, i.e., $boost_1 = 3$ and $boost_2 = 2$. In this case the analysis is the same as P-PIP (Theorem 1). Using

the calculations in Section 11.4.1, the resulting response times of the five tasks are listed in Table 11.2.

Table 11.2: Response times of tasks.

τ_i	RT_i
τ_1	2
τ_2	6
τ_3	25
τ_4	26

In the table one can see that the response times of all tasks are less than their corresponding deadlines (Table 11.1) which means that the task set is schedulable under PIP (P-PIP), i.e., when the boosting levels are at their minimum values.

Setting the Boosting Levels to Maximum

By setting the boost levels of both global resources R_1 and R_2 to be higher than any priority in the application (i.e., $boost_1 = boost_2 = 5$) gives rise to a scenario where the application become unschedulable (at least τ_1 will miss its deadline when having such boost levels). Thus using the common technique for handling global resources in the state-of-art protocols for partitioned scheduling [21, 23, 26, 30, 32, 16] will result the task set in Table 11.1 being unschedulable. In all the existing protocols, the priority of any task holding a global resource is boosted to be higher than any priority on a processor, or they execute non-preemptively.

Setting the Boosting Levels In-Between

By setting the boosting levels for the global resources to the minimum values (e.g., $boost_1 = \lceil R_1 \rceil$) will lead to a longer resource hold time which in turn punishes other applications sharing the resources. The actual resource hold times of R_1 and R_2 with their boost levels set to the minimum values ($\lceil R_1 \rceil$ and $\lceil R_2 \rceil$ respectively) are shown in Table 11.3.

When calling the algorithm presented in Section 11.6.1 for a single resource (any of two resources), the boosting level of the resource can be increased to 5 and the application remains schedulable. Consequently the RHT

Table 11.3: Resource hold times of global resources. $boost_1 = \lceil R_1 \rceil = 3$ and $boost_2 = \lceil R_2 \rceil = 2$.

R_q	$RHT_{q,k}$
R_1	3
R_2	6

of the resources decreases, i.e., $RHT_{1,k} = 2$ and $RHT_{2,k} = 2$ if the algorithm is called for R_1 or R_2 respectively.

However, when increasing the boosting levels for both resources there can be more than one Pareto-optimal solution depending on the order in which the algorithm is called for the resources. Hence, all possible allocations of boosting levels for the resources (i.e., for which the application is schedulable) including the Pareto-optimal allocations are illustrated in Figure 11.3.

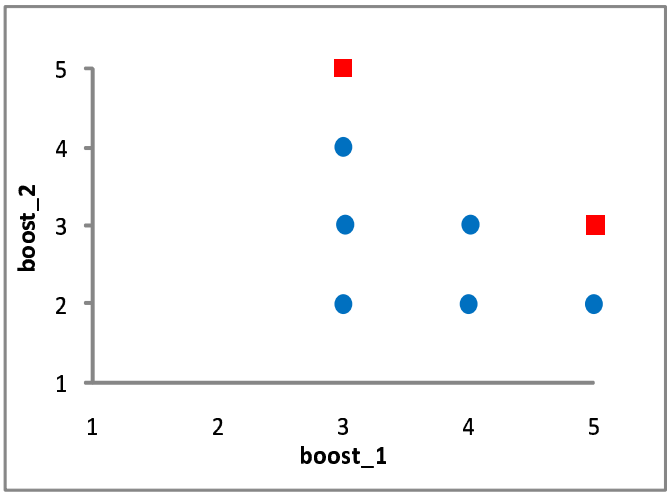


Figure 11.3: The possible allocations of boosting levels for resources. The allocations shown in squares are Pareto-optimal allocations.

Any allocation of boosting levels can be chosen for the resources without compromising the schedulability of the application. However, the allocations shown by squares in Figure 11.3 (i.e., $\{boost_1, boost_2\} = \{(5,3),(3,5)\}$) are the Pareto-optimal allocations which dominate all other allocations.

In this simple example, there are only two global resources, which makes finding all Pareto-optimal allocations relatively easy. However, in a more complex application in which there are more global resources it will not be easy to find all Pareto-optimal allocations of their boost levels. The existing techniques in the domain of multiobjective optimization [33, 34] can be used to find the Pareto-optimal allocations of boost levels in such complex applications.

11.8 Conclusions

In this paper we have studied the resource hold times of resources shared by independently-developed real-time applications on multiprocessor platforms assuming that each application is allocated on a dedicated sub set (cluster) of processors.

We have motivated the work as a step towards co-executing independently-developed real-time applications in an open shared multiprocessor environment.

For a given application, we have derived the computation of Resource Hold Time (RHT) of each resource it shares, under static-priority global scheduling and Priority Inheritance Protocol (PIP), where the RHT is defined as the maximum duration of time a given application may lock a resource. We have reviewed the schedulability analysis of PIP as a locking protocol under multiprocessor global scheduling [27]. The analysis in [27] is developed assuming the Basic PIP (B-PIP) in which a job locking a resource inherits the highest priority among all the jobs blocked on the resource. We have extended the analysis to be applicable to the Immediate Priority Inheritance Protocol (I-PIP) in which a job locking a resource immediately inherits the highest priority of any task that may request the resource.

We have assumed that the real-time applications are scheduled using clustered scheduling, i.e., tasks within applications are scheduled using global scheduling, while each application is statically allocated on a cluster of processors. Considering that clustered scheduling is a combination of partitioned and global scheduling, the usual technique to handle mutually exclusive global resources in the state-of-art locking protocols under partitioned scheduling is to boost the priority of jobs holding the resources to be higher than the priority of any task in a processor (application). However, as we have shown in this paper this technique may make an application unschedulable.

Therefore, in this paper, for a given application, we have shown how to allocate boosting levels for global resources without compromising the schedu-

lability of the application. We have further presented an algorithm that for a given global resource decreases the resource hold time by means of increasing the boosting level of the resource. To increase the boosting levels of all global resources, despite of similar algorithms for uniprocessors [17, 18] that find an optimal allocation of ceilings for the resources, we have shown that under multiprocessor global scheduling and PIP there can exist multiple Pareto-optimal allocations of boosting levels.

In the future we will further study techniques and protocols needed to facilitate co-executing of independently-developed real-time applications in an open environment on multiprocessor platforms.

Bibliography

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of 4th ACM International Conference on Embedded Software (EMSOFT'04)*, pages 95–103, 2004.

[2] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of 23th IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[3] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 166–175, 2000.

[4] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *Proceedings of 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 99–110, 2005.

[5] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, pages 75–84, 2001.

[6] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[7] C. Bialowas. Achieving Business Goals with Wind Rivers Multicore Software Solution. *Wind River white paper*.

[8] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In

Proceedings of 7th ACM & IEEE International conference on Embedded software (EMSOFT'07), pages 279–288, 2007.

[9] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[10] N. Fisher, M. Bertogna, and S. Baruah. The Design of an EDF-Scheduled Resource-Sharing Open Environment. In *Proceedings of 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, 2007.

[11] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[12] U. Devi. Soft real-time scheduling on multiprocessors. In *PhD thesis, available at www.cs.unc.edu/˜anderson/diss/devidiss.pdf*, 2006.

[13] T. Baker. A comparison of global and partitioned EDF schedulability test for multiprocessors. Technical report, 2005.

[14] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of 19th Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[15] Theodore P. Baker and Sanjoy K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*, 2007.

[16] F. Nemati, M. Behnam, and T. Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of 23th Euromicro Conference on Real-time Systems (ECRTS'11)*, pages 251–261, 2011.

[17] N. Fisher, M. Bertogna, and S. Baruah. Resource-Locking Durations in EDF-Scheduled Systems. In *Proceedings of 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 91–100, 2007.

[18] M. Bertogna, N. Fisher, and S. Baruah. Static-priority scheduling and resource hold times. In *Proceedings of 21st IEEE Parallel and Distributed Processing Symposium (IPDPS'07) Workshops*, pages 1–8, 2007.

[19] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of 20th Euromicro Conference on Real-time Systems (ECRTS'08)*, pages 181–190, 2008.

[20] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of 9th IEEE Real-Time Systems Symposium (RTSS'88)*, 1988.

[21] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[22] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of 9th IEEE Real-Time And Embedded Technology Application Symposium (RTAS'03)*, pages 189–198, 2003.

[23] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–83, 2001.

[24] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Journal of Real-Time Systems*, 28(1):39–68, 2004.

[25] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of 18th Euromicro Conference on Real-time Systems (ECRTS'06)*, pages 75–84, 2006.

[26] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 47–56, 2007.

[27] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 377–386, 2009.

[28] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of 31st IEEE Real-Time Systems Symposiumst (RTSS'10)*, pages 49–60, 2010.

[29] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[30] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS. In *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 185–194, 2008.

[31] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of 22nd IEEE Real-Time Systems Symposium (RTSS'07)*, pages 149–160, 2007.

[32] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 469–478, 2009.

[33] M. Ehrgott, editor. *Multicriteria Optimization*. Springer, 2000.

[34] K. Miettinen, editor. *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, 1999.