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

Mälardalen University Press Dissertations
No. 124

RESOURCE SHARING IN REAL-TIME SYSTEMS ON MULTIPROCESSORS

Farhang Nemati

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 25 maj 2012, 14.00 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: professor James H Anderson,
University of North Carolina at Chapel Hill

Akademin för innovation, design och teknik

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.

ISBN 978-91-7485-063-5
ISSN 1651-4238

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:

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

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:

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

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 removedAppList ← all applications sharing resources;
2  for each application A in removedAppList
3     A.priority ← 0;
4  end for
5  while (removedAppList is not empty)
6        List SchedulableApps ← [];
7        List NotSchedulableApps ← [];
8        for each application A in removedAppList
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 failure;
17       removedAppList ← NotSchedulableApps;
18       for each application A in removedAppList
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

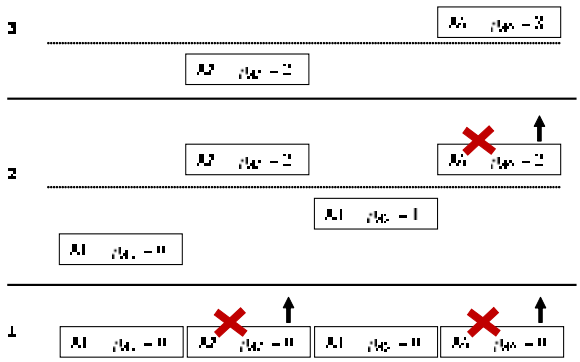

Figure 4.2: Illustrative Example for the Priority Assignment Algorithm

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

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

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.