Mälardalen University Licentiate Thesis
No.119

Partitioned Scheduling of Real-Time Tasks on Multi-core Platforms

Farhang Nemati

May 2010

MÄLARDALEN UNIVERSITY
SWEDEN

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Farhang Nemati, 2010
ISSN 1651-9256
ISBN 978-91-86135-74-4
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

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 licentiatavhandling 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 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.

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 methods 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 distribution of tasks among processors which is a bin-packing problem.

In this thesis we propose a partitioning framework for distributing tasks on the processors of multi-core platforms. Depending on the type of performance we desire to achieve, the framework may distribute a task set differently, e.g., in an application in which tasks process huge amounts of data the goal of the framework may be to decrease cache misses. Furthermore, 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 blocking overhead of tasks which reduces the total utilization and has the potential to reduce the number of required processors. Finally, we have implemented a tool to facilitate evaluation and comparison of different multiprocessor scheduling and synchronization approaches, as well as different partitioning heuristics. We have applied the tool in the evaluation of several partitioning heuristic algorithms, and the tool is flexible to which any new scheduling or synchronization protocol as well as any new partitioning heuristic can easily be added.

Acknowledgments

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

I would like to give many thanks to the people who, with their support, have made PROGRESS to progress; Hans Hansson, Ivica Crnkovic, Paul Pettersson, Sasikumar Punnekkat, Björn Lisper, Mikael Sjödin, Kristina Lundkvist, Jan Gustafsson, Cristina Seceleanu, Frank Lüders, Jan Carlson, Dag Nyström, Andreas Ermedahl, Radu Dobrin, Daniel Sundmark, Rikard Land and Jukka Mäki-Turja.

I also thank people at IDT; Gunnar, Malin, Åsa, Harriet, Monica, Jenny, Monika, Else-Maj, Susanne, Maria and Carola 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, Pasqualina, Rafia, Kathrin, Ana, Sara, Eun-Young, Adnan, Andreas H., Moris, Hüseyin, Marcelo, Bob (Stefan), Luis (Yue), Mikael, Jagadish, Nikola, Rui, Holger, Federico, Saad, Mehrdad, Johan K., Johan F., Juraj, Luka, Leo, Josip, Antonio, Tibi, Lars, Rikard Li., Etienne, Thomas Le., Amine, Adam, Andreas G., Batu, Fredrik, Jörgen, Giacomo, and 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.

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

Farhang Nemati
Västerås, May, 2010

List of Publications

Papers Included in the Licentiate Thesis[1]

Paper A *Efficiently Migrating Real-Time Systems to Multi-Cores.* Farhang Nemati, Moris Behnam, Thomas Nolte. In 14th IEEE International Conference on Emerging Techonologies and Factory (ETFA'09), pages 1205-1212, September, 2009.

Paper B *Blocking-Aware Partitioning for Multiprocessors.* Farhang Nemati, Thomas Nolte, Moris Behnam. Technical Report, MRTC (Mälardalen Real-Time Research Centre), Mälardalen University, March, 2010.

Paper C *Partitioning Real-Time Systems on Multiprocessors with Shared Resources.* Farhang Nemati, Thomas Nolte, Moris Behnam. In submission.

Paper D *A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors.* Farhang Nemati, Thomas Nolte. In submission.

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

Additional Papers, not Included in the Licentiate Thesis

Conferences and Workshops

- *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.

- *Investigation of Implementing a Synchronization Protocol under Multiprocessors Hierarchical Scheduling*. Farhang Nemati, Moris Behnam, Thomas Nolte, Reinder J. Bril (Eindhoven University of Technology, The Netherlands). In 14th IEEE International Conference on Emerging Technologies and Factory (ETFA'09), pages 1670-1673, September, 2009.

- *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.

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

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

- *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.

MRTC reports

- *A Framework for Real-Time Systems Migration to Multi-Cores*. Farhang Nemati, Johan Kraft, Thomas Nolte. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-235/2009-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2009.

Contents

I

Thesis

Chapter 1

Introduction

Inherent in problems with power consumption and related thermal problems, multi-core platforms seem to be the way towards increasing performance of processors, and single-chip multiprocessors (multi-cores) are today the dominating technology for desktop computing. The performance achieved by multi-core architectures was previously only provided by High Performance Computing (HPC) systems. The HPC programmers are required to have a deep understanding of the respective hardware architecture in order to adjust the program explicitly for that hardware. This is not a suitable approach in embedded systems development, due to requirements on productivity, portability, maintainability, and short time to market.

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 distribute tasks fairly on processors to increase the overall performance. Real-time systems are typically multi threaded, hence they are easier to adapt to multi-core platforms than single-threaded, sequential programs. If the tasks are independent of eachother, they can run concurrently to improve performance. Looking at real-time systems, from a practical point of view, a static and manual assignment of processors is often preferred for predictability reasons. Real-time systems can highly benefit from multi-core architectures, as critical functionality can have dedicated cores and independent tasks can run concurrently. Moreover, since the processors are located on the same chip and typically have shared memory, communication between them is very fast.

Many of todays existing *legacy* real-time systems are very large and complex, typically consisting of millions of lines of code which have been developed and maintained for many years. Due to the huge development investments made in these legacy systems, it is normally not an option to throw them away and to develop a new system from scratch. A significant challenge when migrating legacy real-time systems to multi-core platforms is that they have been developed for uniprocessor (single-core) platforms where the execution model is actually sequential. Thus the software may need adjustments where assumptions of uniprocessor have impact.

Mainly, two 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 in practice a typical real-time system includes tasks that share resources. On the other hand, synchronization in the multiprocessor context has not received enough attention. Under partitioned scheduling, if all tasks that share the same resource can be allocated on the same processor 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]. However, under global scheduling methods, the uniprocessor synchronization protocols [9, 1] can not be reused without modification. Instead, new lock-based synchronization protocols have been developed to support resource sharing under global scheduling methods [10, 11].

Partitioned scheduling protocols have been used more often and are supported by commercial real-time operating systems [12], because of their simplicity, efficiency and predictability. However, they suffer from the problem of allocating tasks to processors (partitioning), which is a bin-packing problem [13] and is known to be a NP-hard problem in the strong sense. Thus, to take advantage of performance offered by multi-cores, partitioned scheduling protocols should be coordinated with appropriate partitioning (allocating tasks on processors) algorithms. Heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been studied to find a near-optimal partitioning [2, 3]. However, the existing partitioning algorithms for multiprocessors (multi-cores) mostly assume independent tasks while in real applications, tasks often share resources.

1.1 Contributions

The main contributions of this thesis are as follows.

1. **Partitioning Framework**
   We have proposed a framework that coordinates partitioned scheduling with allocation of tasks (partitioning) on a multi-core platform. Depending on the application the coordination may be different, e.g., in an application in which tasks process huge amounts of data the goal of coordination may be decreasing cache misses, or in an application in which tasks heavily share resources, the coordination will be towards decreasing blocking overhead by allocating tasks sharing the same resources to the same processor as far as possible. Paper A directs this contribution.

2. **Partitioning Heuristic**
   We have proposed a partitioning algorithm, based on bin-packing, for allocating tasks onto processors of a multi-core platform (Chapter 3). Tasks can access mutually exclusive resources and the goal of the algorithm is to decrease the overall blocking overhead in the system. This may consequently increase the schedulability of a task set and reduce the number of processors. We proposed the the partitioning algorithm in Paper B. In Paper C we have further evaluated our algorithm and compared it to a similar algorithm originally proposed in [12].

3. **Implementation**
   We have implemented a tool to facilitate evaluation and comparison of different multiprocessor scheduling and synchronization approaches as well as different partitioning heuristics. We have implemented our partitioning algorithm together with a similar existing algorithm and added

them to the tool. By using the tool, we have performed experiments to evaluate the performance of our heuristic. This tool has been made extensible to allow easy addition of future protocols and algorithms. This contribution is directed by Paper D.

1.2 Thesis Outline

The outline of the thesis is as follows. In Chapter 2 we give a background describing of 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 partitioning framework, heuristic algorithm, and the evaluation tool. In Chapter 4 we present our conclusion and future work. We present the technical overview of the papers that are included in this thesis in Chapter 5, and we present these papers in Chapters 6 - 9.

Chapter 2

Background

2.1 Real-Time Systems

In a real-time system, besides the functional correctness of the system, the output should satisfy timing attributes as well [14], e.g., the outputs should be 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 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. 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 can not) 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.

Multi-core platforms introduce significant challenges, and 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 processors 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, 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. On the other hand, 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 [15]. 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.

A significant advantage of partitioned scheduling is that well-understood and verified scheduling analysis from the uniprocessor domain can 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 (cores) 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 [16, 17]. However, this particular class of scheduling algorithms suffer from high run-time overhead as they may have to increase the number of preemptions and migrations significantly.

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 [18, 19], 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. In turn, clusters are transformed into tasks and are globally scheduled on a multiprocessor. Cluster-based scheduling can be physical or virtual. In physical cluster-based scheduling the processors of each cluster are statically mapped to a subset of processors of the multiprocessor [18]. In virtual cluster-based scheduling 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

In the multiprocessor domain, considerable work has been done on scheduling protocols, but usually under the assumption that tasks are independent. However in practice a typical real-time system must allow for resource sharing among tasks. Generally there are two classes of resource sharing, i.e., lock-based and lock-free synchronization protocols. In the lock-free approach [20], 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 a lock-based approach, thus in this section we present an overview of the existing lock-based synchronization methods.

On a multiprocessor platform a job, besides lower priority jobs, can be blocked by higher priority jobs (those 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 uniprocessor synchronization, is that on a uniprocessor, a job J_i can not be blocked by a lower priority job J_j arriving after J_i. However, on a multiprocessor, assuming jobs J_i and J_j are assigned on different processors, the lower priority job J_i can arrive later than the higher priority job J_i and block J_i. Those cases introduce more complexity and pessimism into schedulability analysis.

For multiprocessor systems, Rajkumar present MPCP (Multiprocessor Priority Ceiling Protocol) [6], which extends PCP [9] to multiprocessors allowing for synchronization of tasks sharing mutually exclusive resources using the partitioned Fixed Priority Scheduling (FPS) protocol.

Gai et al. [7, 8] present the MSRP (Multiprocessor Stack Resource Policy), which extends SRP [1] to multiprocessor platforms and works under the P-EDF scheduling protocol.

Lopez et al. [5] present an implementation of SRP under P-EDF. In this work they propose a solution in which all tasks that directly or indirectly share resources are allocated to the same processor and a uniprocessor synchronization protocol, i.e., SRP, is used to manage resource sharing within each processor. However, if all tasks that directly or indirectly share resources can not be allocated to the same processor the solution can not be used.

Block et al. [10] present FMLP (Flexible Multiprocessor Locking Protocol), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [21] and a comparison between FMLP and MPCP has been presented in [22].

Recently, Easwaran and Andersson have proposed a synchronization protocol [11] under global fixed priority scheduling protocol. In this paper they have derived schedulability analysis of the priority inheritance protocol under global scheduling algorithms, for the first time.

2.4.1 The Multiprocessor Priority Ceiling Protocol (MPCP)

Definition The 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. Local resources are shared only among tasks from the same processor and global resources are shared by tasks assigned to different processors. The local resources are protected using a uniprocessor synchronization protocol, i.e., PCP. A task blocked on a global resource suspends making the processor available for the local tasks. A critical section in which a task performs a request for a global resource is called a *global critical section* (gcs). Similarly a critical section where a task requests for a local resource is denoted as a *local critical section* (lcs).

Under MPCP, the blocking time of a task, in addition to local blocking, has to include remote blocking terms where a task is blocked by tasks (with any priority) 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 priority of a job J_i executing within a gcs in which it requests R_k is called remote ceiling of gcs and 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).

To determine the schedulability of each processor under RM scheduling the following test is performed:

$$\forall k\, 1 \leq i \leq n, \sum_{k=1}^{i} C_k/T_k + B_i/T_i \leq i(2^{1/i} - 1) \tag{2.1}$$

where n is the number of tasks assigned to the processor, and B_i is the maximum blocking time of task τ_i which includes remote blocking factors as well as local blocking time. However this condition is sufficient but not necessary. Thus for more precise schedulability, a test of task response time [23] can be performed.

Blocking times under MPCP Before explaining the blocking factors of the blocking time of a job, the following terminology has to be explained:

- n_i^G: The number of global critical sections of task τ_i.
- $\{J'_{i,r}\}$: The set of jobs on processor P_r (other than J_i's processor) with global critical sections having priority higher than the global critical sections of jobs that can directly block J_i.
- $NH_{i,r,k}$: The number of global critical sections of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\{GR_{i,k}\}$: The set of global resources that will be locked by both J_i and J_k.
- $NC_{i,k}$: The number of global critical sections of J_k in which it request a global resource in $\{GR_{i,k}\}$.
- β_i^{local}: The longest local critical section among jobs with a priority lower than that of job J_i executing on the same processor as J_i which can block J_i.
- $\beta L_i^{\text{global}}$: The longest global critical section of any job J_k with a priority lower than that of job J_i executing on a different processor than J_i's processor in which J_k requests a resource in $\{GR_{i,k}\}$.
- $\beta H_{i,k}^{\text{global}}$: The longest global critical section of job J_k with a priority higher than that of job J_i executing on a different processor than J_i's processor. In this global critical section, J_k requests a resource in $\{GR_{i,k}\}$.

- ${\beta'}_{i,k}^{\text{global}}$: The longest global critical section of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\beta^{\text{lg}}_{i,k}$: The longest global critical section of a lower priority job J_k on the J_i's host processor.

The maximum blocking time B_i of task τ_i is a summation of five blocking factors:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5}$$

where:

1. $B_{i,1} = n_i^G \beta_i^{\text{local}}$ each time job J_i is blocked on a global resource and suspends, the local lower priority jobs may execute and lock local resources and block J_i when it resumes.
2. $B_{i,2} = n_i^G \beta L_i^{\text{global}}$ when a job J_i is blocked on a global resource which is locked by a lower priority job executing on another processor.
3. $B_{i,3} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is not on } J_i\textit{'s processor}}} NC_{i,k} \lceil T_i/T_k \rceil \beta H_{i,k}^{\text{global}}$ when higher priority jobs on processors other than J_i's processor block J_i.
4. $B_{i,4} = \sum_{\substack{J_k \in \{J'_{i,r}\} \\ P_r \neq J_i\textit{'s processor}}} NH_{i,r,k} \lceil T_i/T_k \rceil {\beta'}_{i,k}^{\text{global}}$ when the gcs's of lower priority jobs on processor P_r (different from J_i's processor) are preempted by higher priority gcs's of $J_k \in \{J'_{i,r}\}$.
5. $B_{i,5} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is on } J_i\textit{'s processor}}} \min n_i^G + 1, n_k^G \beta_{i,k}^{\text{lg}}$ when J_i is blocked on global resources and suspends a local job J_k can execute and enter a global section which can preempt J_i when it executes in non-gcs sections.

2.4.2 The Multiprocessor Stack Resource Policy (MSRP)

Definition The 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 (i) local resources that are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. Under MSRP, tasks synchronize 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 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 FCFS (First Come First Served) queue. Global critical sections are not allowed to be nested under MSRP.

Gai et al. in [8] compare their implementation of MSRP to MPCP. They point 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.

Blocking times under MSRP Under MSRP, if a task's job, J_i, attempts to access a global resource, R_q, it becomes non-preemptive. If the resource R_q is free it locks the resource, but if R_q is already locked by another job running on a different processor, J_i performs busy wait. The upper bound of busy wait time that any job executing on processor P_k can experience to access a global resource R_q is as follows.

$$\text{spin}(P_k, R_q) = \sum_{\forall P_l \neq P_k} \max_{\forall J_j \text{ on } P_l} (|Cs_{j,q}|) \tag{2.2}$$

where $|Cs_{j,q}|$ refers to the length of any critical section $Cs_{j,q}$ of J_j accessing R_q.

As a job performs busy wait its global critical sections become longer and consequently its Worst Case Execution Time (WCET) is increased. Thus, the worst case time any job, J_i executing on processor P_k, busy waits can be added to its WCET:

$$C'_i = C_i + \sum_{\forall \text{ global } R_q \text{ accessed by } J_i} \text{spin}(P_k, R_q) \tag{2.3}$$

where C'_i is the actual worst case execution time of J_i.

According to MSRP (similar to SRP), a job can be blocked only once, and as it starts executing it cannot be blocked. The worst case blocking time of any job J_i executing on processor P_k, is calculated as follows:

$$B_i = \max(B_i^{\text{local}}, B_i^{\text{global}}) \tag{2.4}$$

where B_i^{local} and B_i^{local} are the worst case blocking overhead from local resources and global resources respectively, and being defined as follows:

$$\begin{aligned} B_i^{\text{local}} = \max\{|Cs_{j,q}| \mid \ & (J_j \text{ is on } P_k) \ \wedge \ (R_q \text{ is local}) \ \wedge \ (\lambda_i > \lambda_j) \ \wedge \\ & (\lambda_i \leq \text{ceil}(R_q))\} \end{aligned} \tag{2.5}$$

where λ_i is the *static preemption level* of J_i [1].

$$\begin{aligned} B_i^{\text{global}} = \max\{|Cs_{j,q}| + \text{spin}(P_k, R_q) \mid \ & J_j \text{is not on } P_k \ \wedge \\ & R_q \text{ is global}\} \end{aligned} \tag{2.6}$$

2.4.3 The Flexible Multiprocessor Locking Protocol (FMLP)

Definition In FMLP, resources are categorized into *short* and *long* resources, and wether 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 from the group at any time can hold the lock.

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 under G-EDF holds the group lock and executes preemptively using priority inheritance, i.e., it inherits the maximum priority of any higher priority job blocked on any resource within the same 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) [10] 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.

Blocking times under FMLP In FMLP, any job J_i can face three types of blocking overhead:

- *Busy-wait blocking* of job J_i, specified by BW_i, is the maximum duration of time that the job can busy-wait on a short resource.
- *Non-preemptive blocking* occurs when a preemptable job J_i is one of the m highest priority jobs but it is not scheduled because a lower priority job is non-preemptively executing instead. Non-preemptive blocking of J_i denoted by NPB_i is the maximum duration of time that J_i is non-preemptively blocked.
- *Direct blocking* occurs when job J_i is one of the m highest priority jobs but it is suspended because it issues a request for an outermost long resource from group G but another job holds a resource from the same group (holds the group's lock). Direct blocking of job J_i specified by DB_i is the maximum duration of time that J_i can be direct blocked.

The worst case blocking time of any job J_i is the summation of the three sources of blocking times:

$$B_i = \mathrm{BW}_i + \mathrm{NPB}_i + \mathrm{DB}_i \tag{2.7}$$

The detailed calculations of the three sources of blocking times are presented in the appendix to the online version of [10][1].

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 future work we believe that this assumption can be relaxed.

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

Scheduling Protocol:
The focus of this thesis is partitioned scheduling approaches. In the future we will extend our research to global and hybrid scheduling protocols as well.

Synchronization Protocol:
We have focused on MPCP as the synchronization protocol under which our heuristic attempts to decrease blocking overhead, and extending the heuristic to other protocols remains a future work.

Chapter 3

Heuristic Methods for Partitioning Task Sets on Multiprocessors

In this chapter we present a partitioning framework in which a task set is attempted to be efficiently allocated onto a single-chip shared memory multiprocessor (multi-core) platform with identical processors.

A scheduling framework for multi-core processors is presented by Rajagopalan et al. [24]. 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.

An approach for migration to multi-core is presented by Lindhult in [25]. The author presents the parallelization of sequential programs as a way to achieve performance on multi-core processors. The targeted language is PLEX, Ericsson's in-house developed event-driven real-time programming language used for Ericsson's telephone exchange system.

The *grey-box* modeling approach for designing real-time embedded systems is presented in [26]. 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.

Furthermore,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 [13]. Heuristic algorithms have been developed to find near-optimal solutions.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [27]. The presented method partitions a software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of required bins (processors) and the required bandwidth for the communication between nodes.

Liu et al. [28] present 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, in [29] 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. [12] investigate and analyze two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. They have developed a blocking-aware task allocation algorithm, an extension to 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 detail 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.

3.1 Task and Platform Model

Our target system is a task set that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ 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 $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i, in which the task locks resource $R_q \in R$. Critical sections of tasks should be sequential or properly nested. The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

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

3.2 Partitioning Framework for Multi-cores

In this section we present our framework in which tasks are grouped into partitions and each partition is allocated onto one core (processor). At each step when a task is assigned to a partition the following requirements should be satisfied:

1. All partitions are schedulable.

2. The best partition for assigning the task is chosen in a way that the cost is minimized.

Cost function In the framework a cost function is used to calculate the cost of assigning a task to a partition. The cost function can be derived from constraints and preferences which are extracted from the system as well as those offered by the system experts. In the proposed framework, we use a set of constraints and preferences to derive the cost function and to test the schedulability of each partition (processor). The following constraints are used:

1. **Timing constraints**
   Specify timing attributes, e.g., deadline, Worst Case Execution Time

(WCET). Those constraints are used for schedulability test of each partition.

2. **Resource sharing constraints**
   In a typical real-time system, as tasks share resources the corresponding constraints should be considered for schedulability analysis. These constraints together with the timing constraints may be used for deriving the cost function as well.

3. **Task preferences**
   This may include more than one category of preferences. Each category consists (as a matrix) of cost values for each pair of tasks, when they are co-allocated on the same partition. These preferences facilitate allocation of tasks on the processors (partitions) by attracting dependent tasks on the same processor and forcing independent tasks to be allocated on different processors as far as possible. Each category (matrix) of cost values represents an aspect of system performance, e.g., an aspect can be increasing cash hits, or reducing blocking times (Section 3.3). The importance of each category is indicated by a coefficient. The number of categories as well as the value of their coefficient (their importance) depend on the *partitioning strategy*.

Partitioning strategy A partitioning strategy indicates the importance of the type of system performance (e.g., increasing cache hits, decreasing blocking overhead, etc.) we wish to achieve and gives a coefficient parameter to each matrix. The value of each coefficient depends on the importance of the performance that the matrix represents. For example in a system that processes large amounts of data the partitioning strategy can be that the tasks which share data heavily are assigned to the same partition to increase cache hits. Similarly in a system in which tasks share mutually exclusive resources, the target partitioning strategy can be assigning tasks sharing the same resources to the same processor as far as possible. This is the concrete partitioning strategy of our blocking-aware algorithm presented in Section 3.3.

Task weight Generally looking at bin-packing algorithms, e.g., the best-fit decreasing (BFD), objects are allocated into bins in the order of their size, e.g., the heavier objects are packed first. In the context of allocation of tasks onto processors, with independent tasks the utilization of the tasks are considered as their size. However, with dependent tasks other parameters (depending on the partitioning strategy) should be considered in their size (weight). The weight of a task indicates the importance of the task according to the partitioning strategy. For example in a partitioning strategy for reducing inter-core communication, the weight of a task may include the total number of messages it sends or receives during its execution time, or in a partitioning strategy for reducing blocking times of tasks their weight (size) should include blocking parameters.

3.3 Heuristic Partitioning Algorithms with Resource Sharing

In this section we present our proposed blocking-aware heuristic algorithm to allocate tasks onto the processors of a single chip multiprocessor (multi-core) platform. The algorithm extends a bin-packing algorithm with synchronization parameters. The results of our experimental evaluation [30] shows significant performance increment compared to the existing similar algorithm [12] 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 (based on the constraints and preferences) to decrease blocking overheads by assigning tasks to appropriate processors (partitions).

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

The algorithm proposed in [12] was called Synchronization-Aware Partitioning Algorithm, and we call our algorithm Blocking-Aware Partitioning Algorithm. However, to ease refereing them, from now on we refer them as SPA and BPA respectively. In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. To our knowledge the only synchronization protocol under fixed priority partitioned scheduling, for multiprocessor platforms, is MPCP. 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, e.g., MSRP.

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 (partitions) 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 past and future by looking at tasks allocated in the past and those that are not allocated yet. In the first round the algorithm considers the tasks that are not allocated to any processor yet; and attempts to take as many as possible of the best related tasks (based on remote blocking parameters) with the current task. 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., 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 interfered by other tasks:

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

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 (Section 2.4.1), the rationale behind the definition of weight is that the tasks that can be punished more by remote blocking become heavier. Thus, they can be allocated earlier and attract as many as possible of the tasks with which they share resources.

- Next, the *macrotasks* are generated. A macrotask includes 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 can not 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 partition (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 equals to the sum of weights of its tasks.

- After generationg 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 denote this list the *mixed list*.

In the both rounds the strategy of task allocation depends on attraction between tasks. In the partitioning framework in Section 3.2 co-allocation of tasks is based on a cost function. In our blocking-aware algorithm we denote the function *attraction function* which has the same role in partitioning tasks. 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 $v_{i,k}$:

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

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

The definition of weight (Equation 3.1) and attraction function (Equation 3.2) are heuristics to guide the algorithm under MPCP. These function 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 (partitions):

- 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 (all processors are schedulable), beginning from the top of the ordered processor list. If none of the processors can fit the task a new processor is added to the 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 (all processors can successfully be scheduled). If none of the processors can fit the tasks (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 tasks (those that are not allocated yet) within 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 a 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 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 partitions (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. in [12].

- Similar to BPA, the macrotasks are generated (in [12], macrotasks are denoted as bundles). A number of processors (enough processors that fit the total utilization of the task set, i.e., $\lceil u_i \rceil$) 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.

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

  where $\max(|Cs_q|)$ is the length of longest critical section accessing R_q. And the Local Discount is defined as follows.

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

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

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

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

- The macrotask with minimum breaking cost is picked and is broken in two pieces such that the size of one piece is as close as the largest utilization available among processors. This means, tasks within the selected macrotask are ordered in decreasing order of their size (utilization) and the tasks from the ordered list are added to the processor with the largest available utilization as far as possible. In this way, the macrotask has been broken in two pieces; (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 no relationship (e.g., 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 (Equation 3.2) facilitates the BPA, to allocate the most attracted tasks from the current task's broken macrotask, on a processor. As the experimental results in [30] show, considering these issues can improve the partitioning significantly.

3.3.3 Implementation

As the scheduling and synchronization protocols together with partitioning heuristics are being developed in the research community, the industry needs to evaluate the different methods to choose appropriate methods for migrating to multi-core platforms. This arises the need for developing tools to facilitate investigation and evaluation of different approaches and compare them to each other according to different factors.

We have developed a tool for evaluation of different scheduling and synchronization protocols as well as different partitioning algorithms. The output of the tool includes different information and graphs to facilitate evaluation and comparison of different approaches. Furthermore the tool can assist practitioners (given a scheduling and synchronization protocol as well as a partitioning heuristic) to find an appropriate solution for distributing a task set onto the processors of a multi-core platform.

We have implemented our blocking-aware partitioning algorithm together with the algorithm proposed in [12] and added both approaches to the tool. The tool has been developed in a modular manner to which any new scheduling and synchronization algorithms as well as any new partitioning heuristic can easily be added. However, the focus of the tool is partitioned scheduling and synchronization approaches as well as partitioning heuristics while extending the tool to global scheduling methods remains as a future work.

3.4 Summary

In this chapter we presented an overview of the contributions of the thesis, which is the development of a partitioning framework, a partitioning heuristic, and a tool to evaluate different scheduling, synchronization and partitioning algorithms for multi-cores. The target multi-core architecture is identical, unit capacity, shared memory multi-core platforms. Regarding the scheduling algorithm, the focus in this thesis is partitioned scheduling, and finally concerning the synchronization protocol, we have focused on MPCP.

In the framework, a task set is partitioned on a multi-core in a way to maximize the desired system performance. Tasks are grouped together based on timing constraints, resource sharing constraints, and task preferences. In the partitioning process, the framework uses a cost function to calculate the cost of allocating a task to a partition.

The heuristic algorithm for partitioning specifically focuses on decreasing the blocking overhead when allocating a task set on a multi-core platform. We have developed and implemented the blocking-aware heuristic algorithm together with a similar existing algorithm and a reference blocking-agnostic algorithm.

We have implemented a tool to evaluate different techniques of partitioned scheduling and synchronization, as well as task allocation algorithms. The tool has been developed in a modular manner to which new protocols and algorithms can easily be add in the future.

Chapter 4

Conclusions

4.1 Summary

In this thesis we have pointed out the increasing interest in multi-core architectures, and we have explained some of the challenges regarding migrating to these platforms. We have briefly discussed the existing scheduling approaches, e.g., partitioned and global scheduling. We have also presented an overview of the existing synchronization protocols for lock-based resource sharing on multiprocessor platforms, e.g., MPCP and FMLP.

We have proposed a general partitioning framework for distribution of a task set on a multi-core platform, which includes a heuristic algorithm that extends a bin-packing algorithm with a cost function based on task constraints and preferences.

We have also 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 multi-core platform. The objective of the algorithm is to decrease blocking times of tasks by means of allocating the tasks that directly or indirectly share resources onto the same processors as far as possible. This generally increases shedulability of a task set and can lead to fewer required processors compared to a blocking-agnostic bin-packing algorithm. Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling [6]. However, this protocol introduces large amounts of blocking time overheads especially when the critical sections (in which a task accesses to resources) are relatively long and the accessing ratio to the resources is high.

Finally, we have implemented a tool to facilitate evaluation and comparison of different multiprocessor scheduling and synchronization approaches as well as different partitioning heuristics.

4.2 Future Work

In the future we plan to work further on the resource sharing issue in the multi-core domains 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 and FMLP for partitioned scheduling. Another interesting future work is to apply our approach to real systems, and to study the performance gained by the algorithm on these systems.

In the domain of multiprocessor scheduling and synchronization our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Looking at the tool that we have developed, the focus of the tool is currently partitioned scheduling approaches, and extending the tool to global scheduling protocols is another interesting future work.

Chapter 5

Overview of Papers

5.1 Paper A

Farhang Nemati, Moris Behnam, Thomas Nolte. *Efficiently Migrating Real-Time Systems to Multi-Cores.* In 14th IEEE International Conference on Emerging Techonologies and Factory (ETFA'09), pages 1205-1212, September, 2009.

Summary Power consumption and thermal problems limit a further increase of speed in single-core processors. Multi-core architectures have therefore received significant interest. However, a shift to multi-core processors is a big challenge for developers of embedded real-time systems, especially considering existing "legacy" systems which have been developed with uniprocessor assumptions. These systems have been developed and maintained by many developers over many years, and cannot easily be replaced due to the huge development investments they represent. An important issue while migrating to multi-cores is how to distribute tasks among cores to increase performance offered by the multi-core platform. In this paper we propose a partitioning algorithm to efficiently distribute legacy system tasks along with newly developed ones onto different cores. The target of the partitioning is increasing system performance while ensuring correctness.

My contribution The basic idea of this paper was suggested by Farhang Nemati. Farhang was the main driver in writing and finalization of the paper.

5.2 Paper B

Farhang Nemati, Thomas Nolte and Moris Behnam. *Blocking-Aware Partitioning for Multiprocessors*. Technical Report, MRTC (Mälardalen Real-Time Research Centre), Mälardalen University, March, 2010.

Summary In the multi-core and multiprocessor domain there are two scheduling approaches, global and partitioned scheduling. Under global scheduling each task can execute on any processor while under partitioned scheduling tasks are allocated to processors and migration of tasks among processors is not allowed. Under global scheduling the higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand, besides simplicity and efficiency of partitioned scheduling protocols, existing scheduling and synchronization methods developed for uniprocessor 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 how to distribute tasks among processors/cores to increase performance offered by the platform. However, existing methods mostly assume independent tasks while in practice a typical real-time system contains tasks that share resources and they may block each other. In this paper we propose a blocking-aware partitioning algorithm to distribute tasks onto different processors. The proposed algorithm allocates a task set onto processors in a way that blocking times of tasks are decreased. This reduces the total utilization which has the potential to decrease the total number of needed processors/cores.

My contribution The idea of this paper was suggested by Farhang Nemati. Farhang was the main driver in writing the paper and he was responsible for implementation and evaluation of the algorithm proposed in the paper.

5.3 Paper C

Farhang Nemati, Thomas Nolte and Moris Behnam. *Partitioning Real-Time Systems on Multiprocessors with Shared Resources*. In submission.

Summary There are two main approaches to task scheduling on multiprocessor/multi-core platforms; 1) global scheduling, under which migration of tasks among processors is allowed, and 2) partitioned scheduling under which tasks are allocated onto processors and task migration is not allowed. Under global scheduling a higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand under partitioned scheduling, besides simplicity and efficiency, existing scheduling and synchronization methods developed for uniprocessor platforms can more easily be extended to partitioned scheduling. However the partitioned scheduling protocols suffer from the problem of partitioning tasks among processors/cores which is a bin-packing problem. Therefore, several heuristic algorithms have been developed for partitioning a task set on multiprocessor platforms. However, these algorithms typically assume independent tasks while in practice real-time systems often contain tasks that share resources and hence may block each other.

In this paper we propose a blocking-aware partitioning algorithm which allocates a task set onto processors 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 Farhang Nemati was the main driver in writing the paper and he was responsible for further evaluation of the algorithm. He was also responsible for implementing an algorithm similar to the algorithm proposed in Paper B, and comparing the two algorithms.

5.4 Paper D

Farhang Nemati, Thomas Nolte. *A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors.* In submission.

Summary Multi-core platforms seem to be the way towards increasing performance of processors. Single-chip multiprocessors (multi-cores) are today the dominating technology for desktop computing. As the multi-cores are becoming the defacto processors, the need for new scheduling and resource sharing protocols has arisen. There are two major types of scheduling under multiprocessor/multi-core platforms. Global scheduling, under which migration of tasks among processors is allowed, and partitioned scheduling under which tasks are allocated onto processors and task migration is not allowed. The partitioned scheduling protocols suffer from the problem of partitioning tasks among processors/cores, which is a bin-packing problem. Heuristic algorithms have been developed for partitioning a task set on multiprocessor platforms. However, taking such technology to an industrial setting, it needs to be evaluated such that appropriate scheduling, synchronization and partitioning algorithms are selected.

In this paper we present our work on a tool for investigation and evaluation of different approaches to scheduling, synchronization and partitioning on multi-core platforms. Our tool allows for comparison of different approaches with respect to a number of parameters such as number of schedulable systems and number of processors required for scheduling. The output of the tool includes a set of information and graphs to facilitate evaluation and comparison of different approaches.

My contribution Farhang Nemati was the main driver in writing the paper and he was responsible for implementation of the tool.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] J. A. Stankovic and K. Ramamritham, editors. *Tutorial: hard real-time systems*. 1989.

[15] 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.

[16] 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.

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

[18] 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 IEEE Euromicro Conference on Real-time Systems (ECRTS'07)*, pages 247–258, 2007.

[19] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *proceedings of 20th IEEE*

Euromicro Conference on Real-time Systems (ECRTS'08), pages 181–190, 2008.

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

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

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

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

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

[25] J. Lindhult. Operational semantics for plex a basis for safe parallelization. In *Licentiate thesis, Mälardalen University Press*, 2008.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *In submition*, 2010.

II

Included Papers

Chapter 6

Paper A: Efficiently Migrating Real-Time Systems to Multi-Cores

Farhang Nemati, Moris Behnam and Thomas Nolte
In ETFA'09 Conference, pages 1205–1212, 2009

Abstract

Power consumption and thermal problems limit a further increase of speed in single-core processors. Multi-core architectures have therefore received significant interest. However, a shift to multi-core processors is a big challenge for developers of embedded real-time systems, especially considering existing ”legacy” systems which have been developed with uniprocessor assumptions. These systems have been developed and maintained by many developers over many years, and cannot easily be replaced due to the huge development investments they represent. An important issue while migrating to multi-cores is how to distribute tasks among cores to increase performance offered by the multi-core platform. In this paper we propose a partitioning algorithm to efficiently distribute legacy system tasks along with newly developed ones onto different cores. The target of the partitioning is increasing system performance while ensuring correctness.

6.1 Introduction

Due to the problems with power consumption and related thermal problems, multi-core platforms seem to be the way towards increasing performance of processors. Multi-core is 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 distribute tasks fairly on cores to increase the performance. Real-time systems can highly benefit from the multi-core processors, as critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance and thereby enable new functionality. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast. Since embedded real-time systems are typically multi threaded, they are easier to adapt to multi-core than single-threaded, sequential programs, which need to be parallelized into multiple threads to benefit from multi-core. If the tasks are independent, it is simply a matter of deciding on which core each task should execute. For embedded real-time systems, a static and manual assignment of cores is often preferred for predictability reasons. However, many of today's existing ”legacy” real-time systems are very large and complex, typically consisting of millions of lines of code which have been developed and maintained for many years. Due to the huge development investments, it is normally not an option to throw them away and to develop a new system from scratch. However introducing new functionalities into the legacy systems may require more powerful processors, therefore, to benefit from multi-core processors, they need to be migrated from single-core architectures to multi-core architectures.

A significant challenge when migrating legacy real-time systems to multi-core processors is that they have been developed for single-core processors where the execution model is actually sequential. This assumption may introduce complications in a migration to multi-core [1]. Thus the software may need adjustments where assumptions of single-core have impact, e.g., non-preemptive execution may not be sufficient to protect shared resources.

Migrating legacy systems to multi-core processors is discussed in [2]. Advantages and disadvantages of different target architectures of multi-core processors are compared.

In this paper we present an algorithm for migration based on a heuristic *par-*

titioning which allocates tasks to the cores. Tasks can be both legacy tasks extracted from the legacy system as well as newly developed ones. The algorithm identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, which impact multi-core migration. The algorithm tries to increase the performance by reducing the overheads (e.g., blocking times and cache miss overheads) by assigning tasks to appropriate partitions. Partitioning 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. Heuristic functions have been considered to find near-optimal solutions. In this paper we extend a bin-packing algorithm with task constraints which considers performance as well as schedulability of partitions assigned to the cores.

6.1.1 Related Work

An approach for migration to multi-core is presented by Lindhult in [3]. The author presents the parallelization of sequential programs as a way to achieve performance on multi-core processors. The targeted language is PLEX, Ericsson's in-house developed event-driven real-time programming language used for Ericsson's telephone exchange system.

A work related to ours is presented in [4] where a scheduling framework for multi-core processors is presented. 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. However the framework targets new development systems and does not mention migration of existing legacy systems with single-core assumptions.

Liu et al. [5] present a heuristic algorithm for allocating tasks in multi-core based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this paper) are assigned to processing nodes, the second round allocates tasks in a process to the cores of a processor.

The grey-box modeling approach for designing real-time embedded systems [6] is of relevance to our work. 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 this approach the design problems that are targeted at task-level are (1) task concurrency extraction from the system specifications, (2) automatic scheduling algorithm selection, (3) allocation and assignment of processors, and (4) resource estimators, high level timing estimators and interface refinement. However, in our approach, except specifications of the new tasks, the legacy system is used as the main source of task concurrency and resource sharing information.

A study of bin-packing algorithms for designing distributed real-time systems is presented in [7]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes.

Baruah and Fisher have presented a bin-packing partitioning algorithm (first-fit decreasing (FFD) algorithm) in [8] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k, for which both of following conditions, under the Earliest Deadline First (EDF) scheduling, hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i and D_i specify worst-case execution time (WCET) and deadline of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0 & \text{if } t < D_i; \\ C_i + u_i(t - D_i) & \text{otherwise.} \end{cases}$$

The algorithm, however, assumes that tasks are independent while in practice tasks share resources and therefore blocking time overheads must be considered while schedulability of tasks assigned to the a core is checked. Our algorithm not only considers resource sharing when distributing tasks but it tries to reduce blocking times along with other costs. On the other hand their algorithm works under the EDF scheduling protocol while most of legacy real-time systems use fixed priority scheduling policies. Our proposed algorithm works under fixed priority scheduling protocols as well as other policies.

6.1.2 Multi-Core Platforms

A multi-core processor is a combination of two or more independent cores on a single chip. They are connected to a single shared memory via a shared bus. The cores typically have independent L1 caches and share an on-chip L2 cache. Figure 6.1 depicts an example of the architecture.

There are two approaches for scheduling sporadic and periodic task systems on multi-core systems [9, 8, 10, 11] which are inherited from multiprocessor systems; *global* and *partitioned* scheduling.

Under global scheduling, e.g., *Global Earliest Deadline First* (G-EDF), tasks are scheduled by a single scheduler based on their priorities and each task can be executed on any core. A single global queue is used for storing jobs. A task as well as a job can be preempted on a core and resumed on another core (migration of tasks among cores is permitted).

Under partitioned scheduling tasks are statically assigned to cores and tasks within each core are scheduled by uniprocessor scheduling protocols, e.g., *Rate Monotonic* (RM) and EDF. Each core is associated with a separate ready queue for scheduling task jobs.

However there are systems in which some tasks cannot migrate among cores while other tasks can migrate. For such systems neither of global or partitioned scheduling methods can be used. A two-level hybrid scheduling [11] which is a mix of global and partitioned scheduling methods is used for those systems.

Partitioned scheduling protocols have been used more often, as they are more predictable. However, finding an optimal partitioning of tasks on the cores is known to be NP-hard. Thus heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been studied to find a near-optimal partitioning [9, 10].

While in practice tasks share resources, many of scheduling protocols for multiprocessors (multi-cores) assume independent tasks. However, synchronization which is not less important than scheduling has received less attention.

Most legacy systems use Fixed Priority Scheduling (FP) protocols. To our knowledge the only synchronization protocol under fixed priority scheduling, for multiprocessor platforms is *Multiprocessor Priority Ceiling Protocol* (MPCP) which was proposed by Rajkumar in [12]. Thus the protocol is suitable for legacy systems when migrating to multi-cores. Our algorithm assumes that MPCP is used for lock-based synchronization. Hence, we will discuss this protocol in more details in Section 7.3.

The rest of the paper is as follows: we present the task and platform model

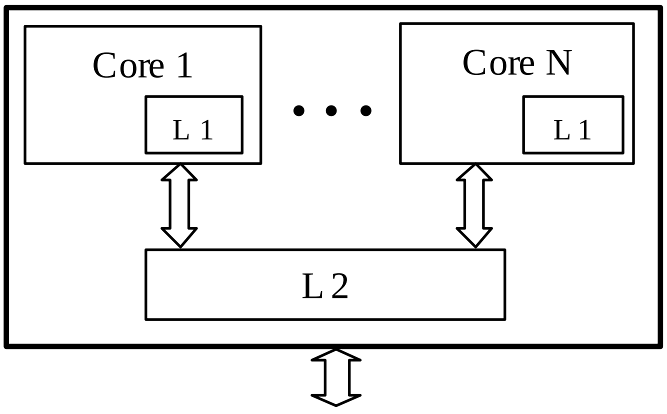


Figure 6.1: Multi-core architecture

in Section 9.2, describe the MPCP in Section 7.3. We present the migration framework and the partitioning algorithm in Sections 6.4 and 6.5 respectively. In Section 9.5 we use our algorithm to reduce blocking time overheads under MPCP.

6.2 Task and Platform Model

We will assume a task set (tasks extracted from legacy system along with new tasks) that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ 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 $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i in which the task locks resource $R_q \in R$. Critical sections of tasks can be sequential or properly nested. The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

We will also assume that the multi-core platform is composed of m identical, unit-capacity processors (cores). The task set is partitioned into m partitions $\{P_1, \ldots, P_m\}$, and each partition is allocated on one core.

6.3 The Multiprocessor Priority Ceiling Protocol (MPCP)

6.3.1 Definition

The MPCP was proposed by Rajkumar in [12] for synchronizing a set of tasks sharing lock-based resources under partitioned FP scheduling, i.e., RM. Under MPCP, resources are divided into *local* and *global* resources. Local resources are shared only among tasks from the same processor and global resources are shared by tasks assigned to different processors. The local resources are protected using a uniprocessor synchronization protocol, i.e., priority ceiling protocol (PCP)[13]. A task blocked on a global resource suspends and makes the processor available for the local tasks. A critical section in which a task performs a request for a global resource is called *global critical sections* (*gcs*). Similarly a critical section where a task requests for local resource is *local critical sections* (*lcs*).

The blocking time of a task in addition to local blocking, needs to include *remote blocking* where a task is blocked by tasks (with any priority) executing on other processors (cores). 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 priority of any other task, hence a *gcs* can only be blocked by another *gcs* and not by any non-critical section. If ρ_H is the highest priority among all tasks, the ceiling of any global resource R_k will be $\rho_H + 1 + \max\{\rho_j | \tau_j \textit{ requests } R_k\}$. The priority of a job executing within a *gcs* is the ceiling of the global resource it requests in the *gcs*.

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. In Section 9.5, our proposed algorithm attempts to reduce the blocking times by assigning tasks to appropriate processors.

To determine the schedulability of each processor under RM scheduling the following test is performed:

$$\forall k \, 1 \leq i \leq n, \sum_{k=1}^{i} C_k/T_k + B_i/T_i \leq i(2^{1/i} - 1) \qquad (6.1)$$

where n is the number of tasks assigned to the processor, and B_i is the maximum blocking time of task τ_i which includes remote blocking factors as well as local blocking time.

However this condition is sufficient but not necessary. Thus for schedulability test of tasks the response time analysis may be used to test if the condition 7.1 is not true for some tasks.

6.3.2 Blocking Times of Tasks

Before explaining the blocking factors of blocking time of a job, we have to explain the following terminology:

- n_i^G: The number of global critical sections of task τ_i.
- $NL_{i,r}$: The number of jobs with priority lower than the priority of J_i executing on processor P_r.
- $\{J'_{i,r}\}$: The set of jobs on processor P_r (other than J_i's processor) with global critical sections having priority higher than the global critical sections of jobs that can directly block J_i.
- $NH_{i,r,k}$: The number of global critical sections of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\{GR_{i,k}\}$: The set of global resources that will be locked by both J_i and J_k.
- $NC_{i,k}$: The number of global critical sections of J_k in which it request a global resource in $\{GR_{i,k}\}$.
- β_i^{local}: The longest local critical section among jobs with a priority lower than that of job J_i executing on the same processor as J_i which can block J_i.
- $\beta L_{i,k}^{\text{global}}$: The longest global critical section of job J_k with a priority lower than that of job J_i executing on a different processor than J_i's processor in which J_k requests a resource in $\{GR_{i,k}\}$.
- $\beta H_{i,k}^{\text{global}}$: The longest global critical section of job J_k with a priority higher than that of job J_i executing on a different processor than J_i's processor. In this global critical section, J_k requests a resource in $\{GR_{i,k}\}$.

- ${\beta'}_{i,k}^{\text{global}}$: The longest global critical section of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\beta_{i,k}^{\text{lg}}$: The longest global critical section of a lower priority job J_k on the J_i's host processor.

The maximum blocking time B_i of task τ_i is a summation of five blocking factors:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5}$$

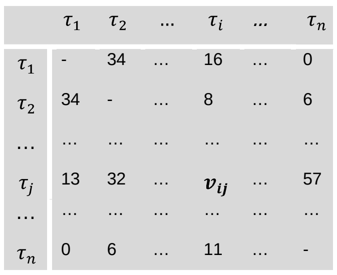
where:

1. $B_{i,1} = n_i^G \beta_i^{\text{local}}$ each time job J_i is blocked on a global resource and suspends the local lower priority jobs may execute and lock local resources and block J_i when it resumes.
2. $B_{i,2} = n_i^G \beta L_i^{\text{global}}$ when a job J_i is blocked on a global resource which is locked by a lower priority job executing on another processor.
3. $B_{i,3} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is not on } J_i\textit{'s processor}}} NC_{i,k} \lceil T_i/T_k \rceil \beta H_{i,k}^{\text{global}}$ when higher priority jobs on processors other than J_i's processor block J_i.
4. $B_{i,4} = \sum_{\substack{J_k \in \{J'_{i,r}\} \\ P_r \neq J_i\textit{'s processor}}} NH_{i,r,k} \lceil T_i/T_k \rceil {\beta'}_{i,k}^{\text{global}}$ when the gcs's of lower priority jobs on processor P_r (different from J_i's processor) are preempted by higher priority gcs's of $J_k \in \{J'_{i,r}\}$.
5. $B_{i,5} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is on } J_i\textit{'s processor}}} \min n_i^G + 1, n_k^G \beta_{i,k}^{\text{lg}}$ when J_i is blocked on global resources and suspends a local job J_k can execute and enter a global section which can preempt J_i when it executes in non-gcs sections.

6.4 Migration Framework

We propose an algorithm that groups tasks into partitions and allocates each partition to a core. At each step when the algorithm assigns a task to a partition the following requirements should be satisfied:

1. Schedulability of the partition is guaranteed.



	τ_1	τ_2	...	τ_i	...	τ_n
τ_1	-	34	...	16	...	0
τ_2	34	-	...	8	...	6
...	...	...	...	...	...	...
τ_j	13	32	...	v_{ij}	...	57
...	...	...	...	...	...	...
τ_n	0	6	...	11	...	-

Figure 6.2: Task preferences constraints

2. The cost of assigning the task to the partition is minimized.

We derive a cost function that calculates the cost value based on a set of task constraints and preferences which should be extracted from the system as well as those offered by the system experts (Figure 6.3). Task constraints and preferences are defined in next Section.

6.4.1 Constraints and Preferences

The partitioning algorithm uses the cost function to efficiently distribute tasks among partitions. The cost function is based on following constraints and preferences:

1. *Resource sharing constraints*:
   These constraints indicate the critical sections of, and the resources accessed by each task.

2. *Task constraints*:
   Specify timing attributes, e.g., deadline, worst-case execution time (WCET). Those constraints together with resource sharing constraints are used to check the schedulability of each partition.

3. *Task preferences*:
   A preference category for the task set is represented as a matrix. Figure 6.2 shows an example of such constraints. A cost given to a pair of tasks, τ_i and τ_j is denoted by v_{ij} and indicates the cost when they are assigned to the same partition, i.e., if two tasks are completely independent and can execute in parallel the cost is set to a large value, and for

two tasks that are highly recommended to belong to the same partition the cost is set to a very small value. Each matrix, M_k, represents an aspect of preferences (e.g. communication costs) and has a coefficient E_k which represents the importance of the preference category. Coefficient values depend on the *partitioning strategies* (Section 6.4.2).

Extracting preference matrices is not easy and for complex systems it may require a lot of engineering skills and system knowledge. Hence, the extraction complexity may differ for different matrices. For example Suppose in a system, tasks share large amounts of data, hence increasing cache hits is important. The values in the related matrix could be a function of amount of shared data between task pairs.

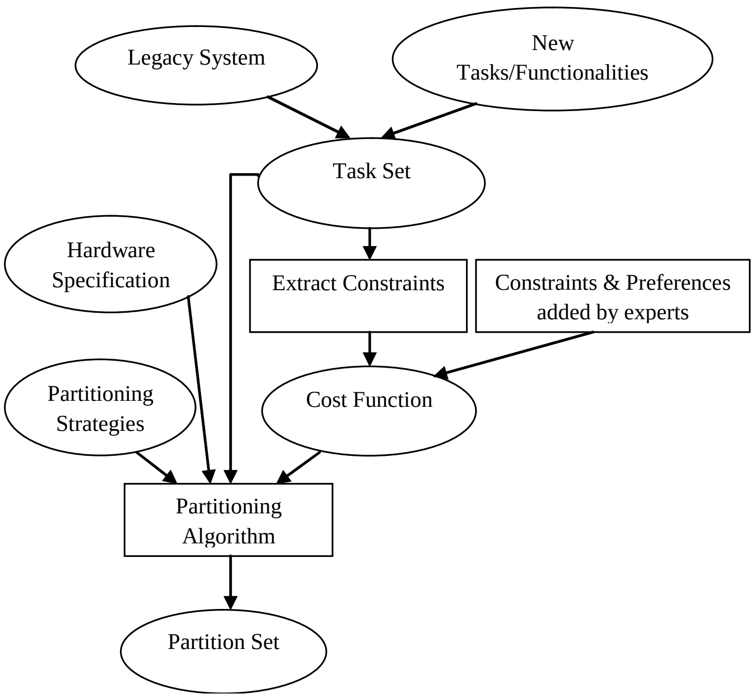


Figure 6.3: A framework for partitioning

6.4.2 Partitioning Strategies

Depending on the nature of a system the strategy of partitioning may differ and result in different partitions. A strategy indicates how tasks are grouped together and based on that the coefficient parameters are given to different preference matrices. For example in a system that processes large amounts of data it is important that the tasks that share data heavily are assigned to the same partition to increase cache hits. On the other hand for a system in which tasks share small amounts of data or are independent, it is important that the tasks are assigned to different partitions to increase parallelism.

The partitioning strategy in Section 9.5 represents extracting a matrix from resource sharing constraints which is used by partitioning algorithm. The partitioning strategy in Section 9.5 is to reduce blocking times under MPCP.

6.4.3 Cost Function

Considering z task preference matrices, the cost function for a partition is formulated based on the task preferences. Let $M_l(v_{ij})$ denote the cost of task τ_i and τ_j being assigned to the same partition in preference matrix M_l with coefficient value E_l. For any partition P_k (where $1 \leq k \leq m$ and m is the total number of partitions/cores), $\mathrm{cost}(P_k)$ denotes the total cost of the partition:

$$\mathrm{cost}(P_k) = u_k^{\alpha} \sum_{l=1}^{z} \left(E_l \sum_{\substack{\tau_i \in P_k \\ \tau_j \in P_k}} M_l(v_{ij})/2 \right) \tag{6.2}$$

where, $u_k = \sum_{\tau_i \in P_k} u_i$, and α is the *utilization parameter*.

The utilization parameter, α, where $\alpha = 0$ or $\alpha = 1$, indicates the importance of task utilizations in the cost function. By setting the utilization parameter to 0 ($\alpha = 0$), the cost function will only depend on the preference matrices. On the other hand by setting $\alpha = 1$ the cost function will also depend on utilization factor of the partition which will increase evenly distribution of tasks among partitions. The total cost of the system is the summation of costs of all partitions.

6.5 Partitioning Algorithm

Now we present an extension to the First-Fit bin-packing algorithm for partitioning sporadic task systems, similar to the algorithm presented in [8]. The major goal of bin-packing algorithms is minimizing the number of needed bins (cores). However our aim is to increase performance while guaranteeing correctness. Thus, we extend the bin-packing algorithm with task preferences (cost function) as well as resource sharing constraints.

The algorithm assumes that tasks are ordered non-increasingly based on their *weights*. The weight of a task τ_i, denoted by w_i indicates the importance of the task according to the partitioning strategy. For example in the partitioning strategy for reducing inter-core communication, the weight of a task may be the total number of messages it sends or receives during its execution time. Figure 6.4 depicts the pseudo-code for the partitioning algorithm.

```
//  The task set $\{\tau_1, ..., \tau_n\}$ is to be assigned into m partitions, $\{P_1, ..., P_m\}$, which will
//  be allocated on m identical cores.
1  order the task set $\{\tau_1, ..., \tau_n\}$ based on their weights;
2  for each partition $P_k$   // m partitions
3       empty $P_k$;
4  end for
5  for i = 1 to n    // n is th enumber of tasks
6        pick the task $\tau_i$ from the top of the ordered list;
7        order partitions by ascending order in cost increment assuming $\tau_i$ is assigned to them;
8        for j = 1 to m    // i ranges over the ordered partitions
9              calculate blocking times of all tasks in all partitions according to MPCP;
10             if all tasks in all partitions satisfy condition (1) then
11                   assign $\tau_i$ to $P_k$;
12             end if
10       end for
13 end for
14 if all tasks are assigned to partitions then
15      partitioning succeeded;
16      goto line 19
17 end if
18 partitioning failed;
19 end
```

Figure 6.4: Partitioning algorithm

The schedulability test 7.1 (Section 7.3) is used for schedulability analysis of any partition, P_k. At each step that the algorithm assigns a task to a partition, P_k, the schedulability test should be performed for all other partitions as well, since the remote blocking term of any task in any partition may be affected.

The algorithm is not limited to FPS and MPCP, and the schedulability test can be extended to other scheduling and resource sharing protocols, e.g., for Partitioned Earliest Deadline First (P-EDF) using the Multiprocessor Stack-based Resource sharing Protocol (MSRP)[14], the following schedulability test from[15] may be used:

$$\sum C_i/T_i + \max_{\tau_i}(B_i/T_i) \leq 1 \tag{6.3}$$

6.6 Reduce Blocking Times under MPCP

6.6.1 Partitioning Strategy

In this section we present a partitioning strategy that targets reducing the blocking times under MPCP. We will use our algorithm to assign tasks to partitions according to the partitioning strategy.

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

The algorithm (Section 6.5) assumes that the tasks are ordered according to their weights. Since the partitioning strategy is to reduce blocking times, the tasks that may cause higher blocking times should get higher weights. Thus the weight of task τ_i should be a function of the number of its critical sections as well as the length of its largest critical sections:

$$w_i = \sum_{q=1}^{|R|}(n_{\forall\,\mathrm{cs}_p}\{c_{i,p,q}\} \times m_{\forall\,\mathrm{cs}_p}\{c_{i,p,q}\})/T_i \tag{6.4}$$

where $n\{c_{i,p,q}\}$ is the number of critical sections in which τ_i requests resource R_q, $m\{c_{i,p,q}\}$ denotes the largest critical section of τ_i requesting R_q, and $|R|$ is the total number of resources in R.

The tasks will be ordered based on their weights and each time the algorithm attempts to assign a task to a partition it will pick the first task (with the highest weight).

Now we will derive a preference matrix which will contain the pair costs (v_ij) for each task pair τ_i and τ_j (Section 6.4.3). First, for any resource R_q we derive an individual matrix in which the cost of pair τ_i and τ_j denoted as $v_{ij,q}$ will be a function of the number of critical sections as well as the length of largest critical sections of tasks τ_i and τ_j:

$$v_{ij,q} = -n\{c_{i,p,q}\} \times m\{c_{i,p,q}\} \times n\{c_{j,k,q}\} \times m\{c_{j,k,q}\} + 1 \qquad (6.5)$$

As the number and the maximum length of critical sections of task pairs increases the cost of assigning them to the same partition should decrease. This is why that first term of the cost in 6.5 has a negative form. If two tasks do not share resource R_q the first term of $v_{ij,q}$ will be 0, hence $v_{ij,q} = 1$ which means if they are assigned to the same partition the cost of the partition should be increased. This is logical because regarding R_q they are independent and are not recommended to be assigned to the same partition.

Task	Period	C_i in non-critical sections	$n\{c_{i,p,1}\}$	$m\{c_{i,p,1}\}$	$n\{c_{i,p,2}\}$	$m\{c_{i,p,2}\}$	$n\{c_{i,p,3}\}$	$m\{c_{i,p,3}\}$	$n\{c_{i,p,4}\}$	$m\{c_{i,p,4}\}$	$n\{c_{i,p,5}\}$	$m\{c_{i,p,5}\}$
τ_1	39	4	1	1	0	0	1	1	0	0	0	0
τ_2	41	5	0	0	1	1	1	1	0	0	0	0
τ_3	42	4	0	0	0	0	0	0	1	1	0	0
τ_4	48	3	0	0	1	2	0	0	1	1	0	0
τ_5	52	5	0	0	0	0	1	2	0	0	1	1
τ_6	57	5	0	0	0	0	0	0	1	1	1	1
τ_7	58	6	1	1	0	0	0	0	2	1	0	0
τ_8	63	8	0	0	0	0	0	0	0	0	0	0

Table 6.1: The task set to be partitioned

The individual matrices for each resource are then used to derive the preference matrix in which v_ij (the cost of pair τ_i and τ_j if they are assigned to the same partition) will be as follows:

$$v_{ij} = \sum_{R_q \in R} v_{ij,q} \qquad (6.6)$$

The partitioning algorithm will use the obtained preference matrix for assigning the tasks to partitions.

6.6.2 Example

In this section we present an example in which our algorithm will attempt to reduce blocking times while partitioning a task set onto different cores of a multi-core processor. The partitioning is performed based on the partitioning strategy in Section 6.6.1.

In this example we set $\alpha = 0$ in the cost function so that the cost only depends on blocking time costs. We attempt to assign a task set consisting of eight tasks (Table 6.1) into four partitions which will be assigned onto a processor with four cores. There are five resources, $\{R_1, R_2, R_3, R_4, R_5\}$ which are shared among tasks and are protected by semaphores. The tasks in Table 6.1 are indexed based on their periods (priority). For each task τ_i, the table contains the period, WCET of non-critical sections, the number of critical sections in which the task request R_q $(n\{c_{i,p,q}\})$ and WCET of the largest critical section for resource R_q $(m\{c_{i,p,q}\})$.

Task	Period
τ_4	0.063
τ_5	0.058
τ_1	0.053
τ_7	0.052
τ_2	0.049
τ_6	0.035
τ_3	0.024
τ_8	0

Table 6.2: The task weights

First, the weights of tasks are calculated based on formula 6.4. Table 6.2 shows the ordered list of tasks based on the calculated weights. For each resource a matrix was created which contains the costs for each task pairs calculated by formula 6.5 and the final preference matrix was obtained based on the resource matrices. Table 6.3 shows the preference matrix which includes the costs for each pair of tasks. Since we only have one preference matrix we set the coefficient of the matrix, E_1, to 1 ($E_1 = 1$).

While partitioning the task set using the bin-packing algorithm without considering the blocking costs does not result in a schedulable system, our algorithm, based on the preference matrix, successfully partitions the task set onto four partitions. Task sets $\{\tau_4, \tau_2\}$, $\{\tau_5, \tau_3\}$, $\{\tau_1, \tau_8\}$, and $\{\tau_7, \tau_6\}$ are assigned to partitions P_1, P_2, P_3, and P_4 respectively. Table 6.4 shows the five blocking factors and total blocking time for each task in the obtained system.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8
τ_1	-	4	5	5	3	5	4	5
τ_2	4	-	5	3	3	5	5	5
τ_3	5	5	-	4	5	4	3	5
τ_4	5	3	4	-	5	4	3	5
τ_5	3	3	5	5	-	4	5	5
τ_6	5	5	4	4	4	-	3	5
τ_7	4	5	3	3	5	3	-	5
τ_8	5	5	5	5	5	5	5	-

Table 6.3: The preference matrix

Task	$B_{i,1}$	$B_{i,2}$	$B_{i,3}$	$B_{i,4}$	$B_{i,5}$	B_i
τ_1	0	4	0	0	0	4
τ_2	2	2	2	0	1	7
τ_3	0	1	0	4	4	9
τ_4	0	1	2	4	0	7
τ_5	0	2	4	1	0	7
τ_6	0	0	6	10	3	19
τ_7	0	0	6	6	0	12
τ_8	0	0	0	0	0	0

Table 6.4: The blocking times of tasks

6.7 Summary and Future Work

In this paper we have mentioned the major challenges (targeting performance and correctness) of migrating a legacy real-time system to multi-core architectures where it will execute along with other systems, e.g., how to take advantage of performance offered by multi-core platforms while guaranteeing correctness. We have proposed a framework for migrating legacy real-time systems to multi-core processors, which includes a heuristic algorithm that extends a bin-packing algorithm with a cost function based on preference matrices. Each obtained partition will be mapped on one core.

Since most legacy real-time systems use fixed priority scheduling protocols, we have developed our framework based on MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. However, this protocol introduces large amounts of blocking time overheads especially when the global resources are relatively long and the access ratio to them is high. As an example we have presented a partitioning strategy and we have obtained preference matrices based on critical sections. The cost function is calculated based on the obtained preference matrix and finally, the algorithm uses the cost function to reduce blocking times.

Our algorithm depends on attributes of tasks, and for legacy systems some information about tasks should be extracted from the existing system. In the future we will study and investigate techniques including reverse engineering methods such as static and dynamic analysis. We will use these methods to extract required information from the legacy system, e.g., information about shared resources, and timing attributes.

A future work will be evaluation of our framework by means of simulation and applying it to a real system. We also plan to study industrial legacy real-time systems and investigate the challenges and possibility of migrating these systems to multi-core architectures. Our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Bibliography

[1] R. Craig and P. N. Leroux. Case study - making a successful transition to multi-core processors. In *QNX Software Systems GmbH & Co. KG*, 2006.

[2] P. N. Leroux and R. Craig. Migrating legacy applications to multicore processors. In *Military Embedded Systems available at http://www.mil-embedded.com /pdfs/QNX.Sum06.pdf*, 2006.

[3] J. Lindhult. Operational semantics for plex a basis for safe parallelization. In *Licentiate thesis, Mälardalen University Press*, 2008.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

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

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

Chapter 7

Paper B: Blocking-Aware Partitioning for Multiprocessors

Farhang Nemati, Thomas Nolte and Moris Behnam
MRTC Technical Report, 2010

Abstract

In the multi-core and multiprocessor domain there are two scheduling approaches, global and partitioned scheduling. Under global scheduling each task can execute on any processor while under partitioned scheduling tasks are allocated to processors and migration of tasks among processors is not allowed. Under global scheduling the higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand, besides simplicity and efficiency of partitioned scheduling protocols, existing scheduling and synchronization methods developed for uniprocessor 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 how to distribute tasks among processors/cores to increase performance offered by the platform. However, existing methods mostly assume independent tasks while in practice a typical real-time system contains tasks that share resources and they may block each other. In this paper we propose a blocking-aware partitioning algorithm to distribute tasks onto different processors. The proposed algorithm allocates a task set onto processors in a way that blocking times of tasks are decreased. This reduces the total utilization which has the potential to decrease the total number of needed processors/cores.

7.1 Introduction

Multi-core (single chip multiprocessor) is today the dominating technology for desktop computing and the performance of using multiprocessors 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 distribute tasks on cores to increase the performance. Real-time systems can highly benefit from multi-core processors, as critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance and thereby enable new functionality. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast. Since embedded real-time systems are typically multi threaded, they are easier to adapt to multi-core than single-threaded, sequential programs. If the tasks are independent, it is a matter of deciding on which core each task should execute. For embedded real-time systems, practically, a static and manual assignment of processors is often preferred for predictability reasons.

There are two approaches for scheduling task systems on multiprocessors systems [1, 2, 3, 4]; global and partitioned scheduling. Under global scheduling, e.g., *Global Earliest Deadline First* (G-EDF), tasks are scheduled by a single scheduler based on their priorities and each task can be executed on any core. A single global queue is used for storing jobs. A task as well as a job can be preempted on a core and resumed on another core (migration of tasks among cores is permitted).

Under partitioned scheduling tasks are statically assigned to processors and tasks within each processor are scheduled by uniprocessor scheduling protocols, e.g., *Rate Monotonic* (RM) and EDF. Each processor is associated with a separate ready queue for scheduling task jobs.

However there are systems in which some tasks cannot migrate among cores while other tasks can migrate. For such systems neither of global or partitioned scheduling methods can be used. A two-level hybrid scheduling [4] which is a mix of global and partitioned scheduling methods is used for those systems.

Partitioned scheduling protocols have been used more often and are supported (with fixed priority scheduling) widely by commercial real-time operating systems[5], because of their simplicity, efficiency and predictability. However, *partitioning*, which allocates tasks to processors, 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. Thus heuristic approaches and sufficient feasibility tests for bin-packing algorithms have been studied to find a near-optimal partitioning [1, 3].

While in real applications tasks often share resources, many of the scheduling protocols and existing partitioning algorithms for multiprocessors (multi-cores) assume independent tasks.

7.1.1 Contributions

The first contribution of this paper is to propose a blocking-aware heuristic algorithm to allocate tasks onto the processors of a single chip multiprocessor (multi-core) platform. The algorithm extends a bin-packing algorithm with synchronization parameters. The second contribution is to implement and evaluate the algorithm and compare it to the blocking-agnostic bin-packing partitioning algorithm. Blocking-agnostic algorithm, in the context of this paper refers to a bin packing algorithm that does not consider blocking parameters to increase the performance of partitioning, although blocking times are included in the schedulability test. The new algorithm identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, and extends the *best-fit decreasing* (BFD) bin-packing algorithm with blocking time parameters. The objective of the algorithm is to decrease blocking overheads by assigning tasks to appropriate processors (partitions).

In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. To our knowledge the only synchronization protocol under fixed priority partitioned scheduling, for multiprocessor platforms is *Multiprocessor Priority Ceiling Protocol* (MPCP) which was proposed by Rajkumar in [6]. Our algorithm assumes that MPCP is used for lock-based synchronization. Hence, we will discuss this protocol in more details in Section 7.3.

The rest of the paper is as follows: we present the task and platform model in Section 9.2, describe the MPCP in Section 7.3. We present the partitioning algorithm in Section 6.5. In Section 8.4 the experimental results of our algorithm are presented and the results are compared to the blocking-agnostic BFD.

7.1.2 Related Work

A study of bin-packing algorithms for designing distributed real-time systems is presented in [7]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes. However, their partitioning method assumes independent tasks.

Liu et al. [8] present a heuristic algorithm for allocating tasks in multi-core-based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this paper) 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 (first-fit decreasing (FFD) algorithm) in [9] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k, for which both of following conditions, under the Earliest Deadline First (EDF) scheduling, hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i and D_i specify Worst Case Execution Time (WCET) and deadline of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0 & \text{if } t < D_i; \\ C_i + u_i(t - D_i) & \text{otherwise.} \end{cases}$$

The algorithm, however, assumes independent tasks while in practice tasks often share resources and therefore blocking time overheads must be considered while schedulability of tasks assigned to a processor is checked. Our algorithm not only considers resource sharing when distributing tasks but it tries to reduce blocking times as well. On the other hand their algorithm works under the EDF scheduling protocol while most existing real-time systems use fixed priority scheduling policies. Our proposed algorithm works under fixed priority scheduling protocols, although it can easily be extended to other policies.

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

In their partitioning algorithm, the tasks that directly or indirectly share resources are put into what they call bundles (we call them macrotasks) and each bundle is tried to be allocated onto a processor. The bundles that can not fit into any existing processors are ordered by their cost, which is the blocking overhead that they introduce into the system. Then the bundle with minimum cost is broken and the algorithm is run from the beginning. However, their algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its size (utilization). Furthermore, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle is considered which could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times.

However, their experimental results show that a blocking-aware bin-packing algorithm for suspend-based execution control policy does not have significant benefits compared to a blocking-agnostic bin-packing algorithm. Firstly, for the comparison, they have only focused on the processor reduction issue; they suppose that the algorithm is better if it reduces the number of processors. In this perspective they claim that in the worst case the number of needed processors would be equal to the number of tasks, while the worst case could be the case that an algorithm fails to schedule a task set. In our experimental evaluation, besides processor reduction, we have considered this issue as well. If an algorithm can schedule some task sets while others fail, we consider it as a benefit. Secondly, in their experiments they have not investigated the effect of some parameters such as the different number of resources, variation in the number and length of critical sections of tasks. By considering these parameters, our experimental results show that in most cases our blocking-aware algorithm has significantly better results than blocking-agnostic algorithms.

In the context of multiprocessor synchronization, the first protocol was MPCP presented by Rajkumar in [6], which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Our partitioning algorithm attempts to decrease blocking times under MPCP and consequently decrease worst case response times which in turn may reduce the number of needed processors. Gai et al. [10, 11] present MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors. The shared resources are classified as either (i) local resources that are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. In MSRP, tasks synchronize local resources using SRP [2], and access to global resources is guaranteed a bounded blocking time. Lopez et al. [12] present an implementation of SRP under P-EDF. Devi et al. [13] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [14] present FMLP (Flexible Multiprocessor Locking Protocol), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [15]. However, although in a longer version of [14][1], the blocking times have been calculated, but to our knowledge there is no schedulability test for FMLP.

Recently, a synchronization protocol under fixed priority scheduling, has been proposed by Easwaran and Andersson in [16], however, they focus on a global scheduling approach.

7.2 Task And Platform Model

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

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

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

onto one processor (core), thus m represent the minimum number of processors needed.

7.3 The Multiprocessor Priority Ceiling Protocol (MPCP)

7.3.1 Definition

The MPCP was proposed by Rajkumar in [6] for synchronizing a set of tasks sharing lock-based resources under partitioned FP scheduling, i.e., RM. Under MPCP, resources are divided into *local* and *global* resources. Local resources are shared only among tasks from the same processor and global resources are shared by tasks assigned to different processors. The local resources are protected using a uniprocessor synchronization protocol, i.e., Priority Ceiling Protocol (PCP)[17]. A task blocked on a global resource suspends and makes the processor available for the local tasks. A critical section in which a task performs a request for a global resource is called *global critical sections* (*gcs*). Similarly a critical section where a task requests for local resource is *local critical sections* (*lcs*).

The blocking time of a task in addition to local blocking, needs to include *remote blocking* where a task is blocked by tasks (with any priority) executing on other processors (cores). 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 priority of any other task, hence a *gcs* can only be blocked by another *gcs* and not by any non-critical section. If ρ_H is the highest priority among all tasks, the ceiling of any global resource R_k will be $\rho_H + 1 + \max\{\rho_j | \tau_j \textit{ requests } R_k\}$. The priority of a job executing within a *gcs* is the ceiling of the global resource it requests in the *gcs*.

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. In Section 9.5, our proposed algorithm attempts to reduce the blocking times by assigning tasks to appropriate processors.

To determine the schedulability of each processor under RM scheduling the following test is performed:

$$\forall k\, 1 \leq i \leq n, \sum_{k=1}^{i} C_k/T_k + B_i/T_i \leq i(2^{1/i} - 1) \tag{7.1}$$

where n is the number of tasks assigned to the processor, and B_i is the maximum blocking time of task τ_i which includes remote blocking factors as well as local blocking time. However this condition is sufficient but not necessary. Thus for more precise schedulability test of tasks our algorithm performs response time analysis [18].

7.3.2 Blocking Times under MPCP

Before explaining the blocking factors of the blocking time of a job, the following terminology has to be explained:

- n_i^G: The number of global critical sections of task τ_i.
- $\{J'_{i,r}\}$: The set of jobs on processor P_r (other than J_i's processor) with global critical sections having priority higher than the global critical sections of jobs that can directly block J_i.
- $NH_{i,r,k}$: The number of global critical sections of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\{GR_{i,k}\}$: The set of global resources that will be locked by both J_i and J_k.
- $NC_{i,k}$: The number of global critical sections of J_k in which it request a global resource in $\{GR_{i,k}\}$.
- β_i^{local}: The longest local critical section among jobs with a priority lower than that of job J_i executing on the same processor as J_i which can block J_i.
- $\beta L_i^{\text{global}}$: The longest global critical section of any job J_k with a priority lower than that of job J_i executing on a different processor than J_i's processor in which J_k requests a resource in $\{GR_{i,k}\}$.
- $\beta H_{i,k}^{\text{global}}$: The longest global critical section of job J_k with a priority higher than that of job J_i executing on a different processor than J_i's processor. In this global critical section, J_k requests a resource in $\{GR_{i,k}\}$.

- ${\beta'}_{i,k}^{\text{global}}$: The longest global critical section of job $J_k \in \{J'_{i,r}\}$ having priority higher than a global critical section on processor P_r that can directly block J_i.
- $\beta_{i,k}^{\text{lg}}$: The longest global critical section of a lower priority job J_k on the J_i's host processor.

The maximum blocking time B_i of task τ_i is a summation of five blocking factors:

$$B_i = B_{i,1} + B_{i,2} + B_{i,3} + B_{i,4} + B_{i,5}$$

where:

1. $B_{i,1} = n_i^G \beta_i^{\text{local}}$ each time job J_i is blocked on a global resource and suspends the local lower priority jobs may execute and lock local resources and block J_i when it resumes.
2. $B_{i,2} = n_i^G \beta L_i^{\text{global}}$ when a job J_i is blocked on a global resource which is locked by a lower priority job executing on another processor.
3. $B_{i,3} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is not on } J_i \textit{'s processor}}} NC_{i,k} \lceil T_i/T_k \rceil \beta H_{i,k}^{\text{global}}$ when higher priority jobs on processors other than J_i's processor block J_i.
4. $B_{i,4} = \sum_{\substack{J_k \in \{J'_{i,r}\} \\ P_r \neq J_i \textit{'s processor}}} NH_{i,r,k} \lceil T_i/T_k \rceil {\beta'}_{i,k}^{\text{global}}$ when the gcs's of lower priority jobs on processor P_r (different from J_i's processor) are preempted by higher priority gcs's of $J_k \in \{J'_{i,r}\}$.
5. $B_{i,5} = \sum_{\substack{\rho_i \leq \rho_k \\ J_k \textit{ is on } J_i \textit{'s processor}}} \min n_i^G + 1, n_k^G \beta_{i,k}^{\text{lg}}$ when J_i is blocked on global resources and suspends a local job J_k can execute and enter a global section which can preempt J_i when it executes in non-gcs sections.

7.4 Partitioning Algorithm

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

Considering the blocking factors of tasks under MPCP, tasks with more and longer global critical sections lead to more blocking times. This is also shown by experiments presented in [11]. Our goal is to (i) decrease the number of global critical sections by assigning the tasks sharing resources to the same partition as far as possible, (ii) decrease the ratio and time of holding global resources by assigning the tasks that request the resources more often and hold them longer to the same partition as long as possible.

In our previous work [19, 20] we have proposed a partitioning algorithm in which tasks are grouped together based on task preferences and constraints. The algorithm partitions tasks based on a cost function which is derived from task preferences and constraints. In [19] the resource sharing is only local by means of allocating the tasks that directly or indirectly share resources onto the same processor. Tasks that directly or indirectly share resources are called macrotasks, e.g. if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask. However if a macrotask does not fit in one processor (is not schedulable) the algorithm fails. In [20] tasks belonging to the same macrotask can be allocated to different partitions (processors), thus it is more flexible but it introduces remote blocking overhead into the systems. The goal of the algorithm is to put the tasks into appropriate partitions so that the costs are minimized. The algorithm may have different partitioning strategies, e.g., increasing cash hits, decreasing blocking times, etc. The strategy of partitioning may differ, depending on the nature of a system, and result in different partitions. In current work, however, we focus on decreasing remote blocking overheads of tasks which leads to increasing the schedulability of a task set and possibly reducing the number of processors needed for scheduling the task set.

We have developed a blocking-aware algorithm that is an extension to the BFD algorithm. In a blocking agnostic BFD algorithm, bins (processors) are ordered in non-increasing order of their utilization and tasks are ordered in non-increasing order of their size (utilization). The algorithm tries to allocate the task from the top of the ordered task set 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 processor list. At each step the schedulability of all processors should be tested, because allocating a task to a processor can increase the remote blocking time of tasks previously allocated to other processors and may make the other processors unschedulable. This means that it is possible that even if a task is allocated to a new processor, some of the previous processors become unschedulable which makes the algorithm fail.

7.4.1 The Algorithm

The algorithm performs partitioning of a task set in two parallel alternatives and the result will be the output of the alternative with better partitioning results. However, the algorithm performs a few common steps before starting to perform the parallel alternatives. Each alternative allocates tasks to the processors (partitions) in a different strategy. When a bin-packing algorithm allocates an object (task) to a bin (processor), it usually puts the object in a bin that fits it better, and it does not consider the unallocated objects that will be allocated after the current object. However, the first alternative of our algorithm considers the tasks that are not allocated to any processor yet; and tries to take as many as possible of the best related tasks (based on remote blocking parameters) with the current task. On the other hand, the second alternative considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. The second alternative performs more like the usual bin packing algorithms, although it considers the remote blocking parameters while allocating a task to a processor.

The common steps of the algorithm before the two alternatives are performed in parallel are as follows.

1. Each task is assigned a weight. The weight of each task, besides its utilization, depends on parameters that lead to potential remote blocking time caused by other tasks:

$$w_i = \lceil (C_i + \sum_{\rho_i < \rho_k} \mathrm{NC}_{i,k}\beta_{i,k}\lceil \frac{T_i}{T_k} \rceil + \mathrm{NC}_i \max_{\rho_i \geq \rho_k} \beta_{i,k})/T_i \rceil \tag{7.2}$$

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

2. Macrotasks are generated; the tasks that directly or indirectly share resources are put into the same macrotask. A macrotask has two alternatives; it can either be broken or unbroken. If a macrotask cannot fit (cannot be scheduled) in one processor, it is assigned as broken, otherwise it is denoted as unbroken. If a macrotask is unbroken, the partitioning algorithm always allocate all tasks in the macrotask to the same partition (processor). This means that all tasks in the macrotask will share resources locally relieving tasks from remote blocking. However, tasks

within a broken macrotask will be distributed into more than one partition. Similar to tasks, a weight is assigned to each unbroken macrotask, which equals to the sum of weights of its tasks.

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

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

$$v_{i,k} = \begin{cases} \mathrm{NC}_{i,k}\beta_{i,k}\lceil\frac{T_i}{T_k}\rceil & \rho_i < \rho_k; \\ \mathrm{NC}_i\beta_{i,k} & \rho_i \geq \rho_k \end{cases} \tag{7.3}$$

Now we present the continuation of each alternative separately.

Alternative 1: After step 3 the following steps are repeated by alternative 1 until all tasks are allocated to processors (partitions):

1. All processors are ordered in their non-increasing order of utilization.

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

   (a) If the object is a task and it does not belong to a broken macrotask it will be allocated onto the first processor that fits it, beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit the task a new processor is added to the list and the task is allocated onto it. In this case if one or more of the processors becomes unschedulable this alternative of the algorithm fails.

   (b) If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them. If none of the processors can fit the tasks, they will be allocated onto a new processor and in this case, if one or more of the processors becomes unschedulable the Alternative 1 fails.

   (c) If the object is a task that belongs to a broken macrotask, the algorithm orders the tasks (those that are not allocated yet) within the macrotask in non-increasing order of attraction to the task based on equation 8.2. We call this list the *attraction list* of the task. The task itself will be on the top of its attraction list. The best processor for allocation is selected, which is the processor that fits the most tasks from the attraction list, beginning from the top of the list. 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., one or more of the processors become unschedulable, the Alternative 1 fails.

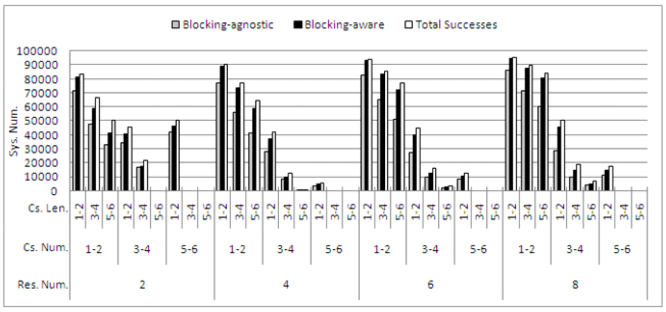
Alternative 2: The following steps are repeated until all tasks are allocated to processors:

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

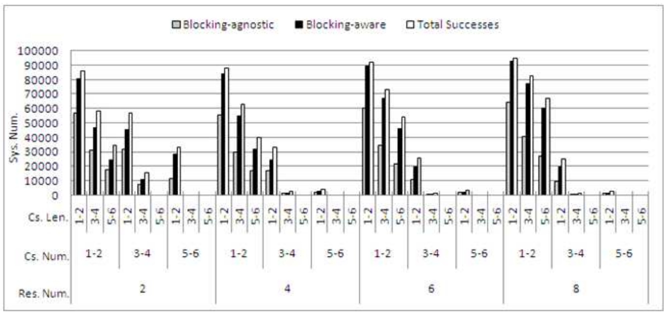
   (a) If the object is a task and it does not belong to a broken macrotask it will be allocated onto the first processor that fits it, beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit the task a new processor is added to the list and the task is allocated onto it. In this case if one or more of the processors becomes unschedulable the Alternative 2 fails.

   (b) If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them. If none of the processors can fit the tasks, they will be allocated onto a new processor and in this case, if one or more of the processors becomes unschedulable the Alternative 2 fails.

   (c) 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 task; this list is ordered in non-increasing order of processors' attraction to the task based on equation (3), 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 those 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. And this alternative of the algorithm fails if allocating the task to the new processor makes some of the processors unschedulabe.



(a) 3 tasks per processor



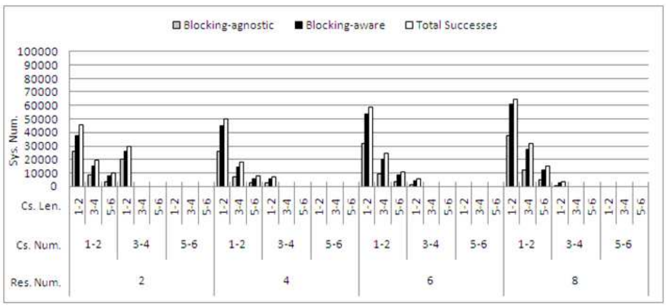
(b) 6 tasks per processor

Figure 7.1: Total number of task sets that the algorithms successfully schedule (task sets generated from 3 fully utilized processors).

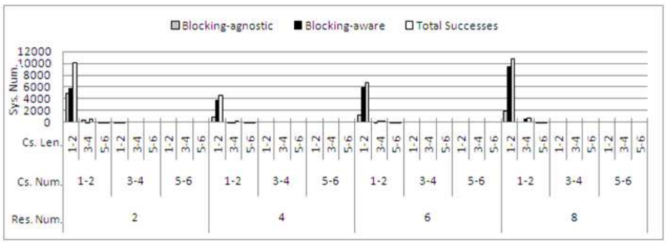
The algorithm fails if both alternatives fail to schedule a task set. If one of the alternatives fails the result will be the output of the other one. Finally if both succeed to schedule the task set, the one with less partitions (processors) will be the output of the algorithm.

7.5 Experimental Evaluation

In this section we present our experimental results of the blocking-aware bin-packing algorithm together with the blocking-agnostic algorithm. For a number of systems (task sets), we have compared the performance of the algorithms in two different aspects; 1) The total number of systems that each of the algorithms can schedule, 2) The total number of systems that one of the algorithms schedules with fewer processors when both succeed.



(a) Workload: 6 fully utilized processors, 3 tasks per processor



(b) Workload: 8 fully utilized processors, 6 tasks per processor

Figure 7.2: Total number of task sets that the algorithms successfully schedule.

7.5.1 Task Set Generation

We generated systems (task sets) for different workloads; we denote workload as a defined number of fully utilized processors. Given a workload, the full capacity of each processor (utilization of 1) is randomly divided into a defined number of tasks utilizations. Usually for generating systems, utilization and periods are randomly assigned to tasks and worst case execution times of tasks are calculated based on them. However, in our system generation, the worst case execution times (WCET) of tasks are randomly assigned and the period of each task is calculated based on its utilization and WCET. The reason is that we had to restrict that the WCET of a task not to be less than the total length of its critical sections. Since we have limited the maximum number of critical sections to 10 and the maximum length of any critical section to 10 time units, hence the WCET of each task should be greater than $100(10 \times 10)$ time units. The WCET of each task was randomly chosen between 100 and 150 time units. The system generation was based on different settings; the input parameters for settings are as follows.

1. Workload (3, 4, 6, or 8 fully utilized processors),
2. The number of tasks per processor (3 or 6 tasks per processor),
3. The number of resources (2, 4, 6, or 8),
4. The range of the number of critical sections per task (1 to 2, 3 to 4 or 5 to 6 critical sections per task),
5. The range of length of critical sections (1 to 2, 3 to 4, or 5 to 6).

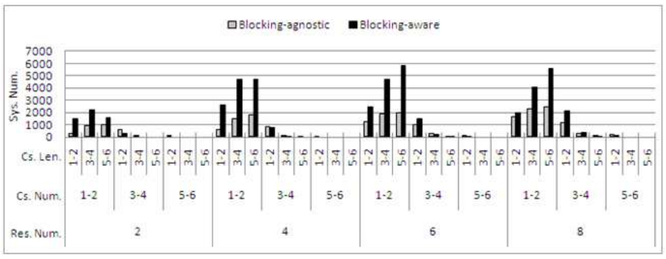
For each setting, we generated 100.000 systems, and combining the parameters of settings (288 different settings), the total number of systems generated for the experiment were 28.800.000.

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

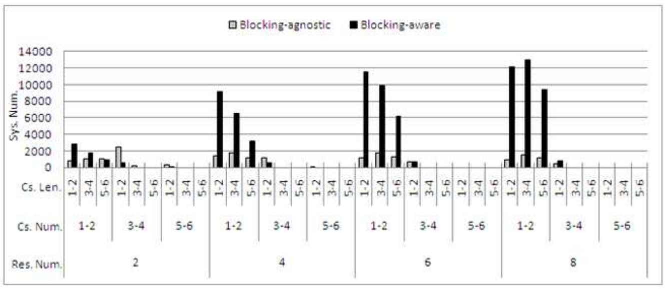
7.5.2 Results

In this section we present the evaluation results of our blocking-aware algorithm. We compare them to the results of the blocking-agnostic bin-packing algorithm.

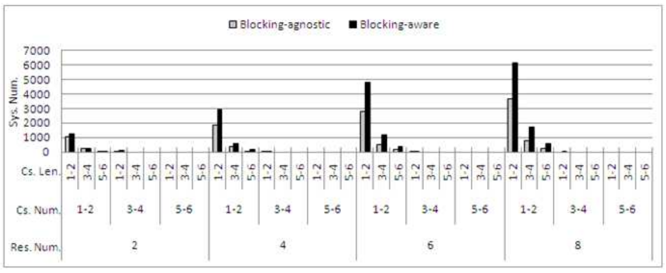
The first aspect of comparison of the results from the two algorithms is the total number of systems that each algorithm succeeds to schedule. Comparison for 3 fully utilized processors is represented in Figure 7.1. Figures 8.1 and 8.2 represent the results for 3 task per processor and 6 tasks per processors respectively. The vertical axis shows the total number of systems that the algorithms



(a) 3 fully utilized processors, 3 tasks per processor



(b) 3 fully utilized processors, 6 tasks per processor



(c) 6 fully utilized processors, 3 tasks per processor

Figure 7.3: Total number of task sets that either of algorithms schedule with fewer processors than the other.

could schedule successfully. The horizontal axis shows three factors in three different lines; the bottom line shows the number of shared resources within systems (Res. Num.), the second line shows the number of critical sections per task (Cs. Num.), and the top line represents the length of critical sections within each task (Cs. Len.), e.g., Res. Num.= 4, Cs. Num.= $1 - 2$, and Cs. Len.= $1 - 2$ represents the systems that share 4 resources, the number of critical sections are between 1 and 2, and the length of these critical sections are between 1 and 2.

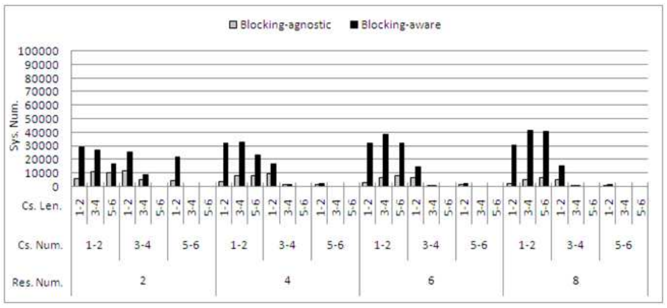


Figure 7.4: Total number of task sets that the algorithms exclusively schedule successfully (workload of 3 fully utilized processors, 6 tasks per processor).

As depicted in Figure 7.1, considering the total number of systems that each algorithm succeeds to schedule, our blocking aware algorithm performs better (more systems are schedulable) compared to the blocking-agnostic algorithm. By increasing the number of resources, the number of successfully scheduled systems in both algorithms is slightly increased. The reason for this behavior is that with fewer resources, more tasks share the same resource introducing more blocking overheads which leads to fewer schedulable systems. However, it is shown that the blocking-aware algorithm performs better as the number of resources is increased. It is also shown that increasing the number and/or the length of critical sections significantly reduces the number of schedulable systems in both algorithms. As the number of tasks per processor is increased from 3 (Figure 8.1) to 6 (Figure 8.2), the blocking-aware algorithm performs significantly better (schedules more systems) than the blocking-agnostic algorithm.

As the workload (the number of fully utilized processors) is increased, although the blocking aware algorithm still performs better than the blocking-agnostic algorithm, the number of schedulable systems by both algorithms is reduced (Figure 7.2). The reason for this behavior is that the number of tasks within systems are relatively many (48 tasks per each system in Figure 7.3(b)) and the workload is high (8 fully utilized processors in Figure 7.3(b)), and all the tasks within systems share resources. This introduces a lot of interdependencies among tasks and consequently a huge amount of blocking overheads, making fewer systems schedulable. In practice in big systems with many tasks, not all of the tasks share resources, which leads to fewer interdependencies among tasks and less blocking times. However, we continued the experiment with higher workload in the same way as the other experiments (that all tasks share resources) to be able to compare the results with the previous results. We believe that realistic systems, even with high workload and many tasks can significantly benefit from our partitioning algorithm to increase the performance.

The second aspect for comparison of performance of the algorithms is the total number of systems that each algorithm schedules with fewer processors than the other one (better in processor reduction). The results show (Figure 7.3) that our blocking aware algorithm mostly performs significantly better than the blocking-agnostic bin packing algorithm, especially given a lower number of critical sections per task and shorter critical sections. However, for lower number of shared resources (e.g., 2 shared resources) especially for higher workloads (e.g., 6 fully utilized processors) the blocking aware algorithm does not always perform better (Figure 7.3).

In the experiment we also studied the results of both alternatives of the algorithm separately. The results show that the alternative 1 mostly performs significantly better than the alternative 2, although in some cases the alternative 2 performs better especially as the number and the length of critical sections increase.

7.5.3 Combination of Algorithms

The results in Section 7.5.2 show that our blocking aware partitioning algorithm mostly performs significantly better in both increasing the number of schedulable systems as well as processor reduction. However, finding an optimal solution with a bin packing algorithm is not realistic (bin packing is a NP-hard problem in the strong sense), hence there may exist schedulable systems that our algorithm fails to schedule. As illustrated in Figure 7.4, the number of systems that our algorithm can exclusively schedule (i.e., the blocking agnostic algorithm fails to schedule them) are significantly higher compared to the number of systems exclusively schedulable by the blocking agnostic algorithm. However, there are still some systems that are only schedulable by the blocking agnostic algorithm. Thus, combination of both algorithms can be convenient to improve the overall results. It can also be noticed in Figure 7.1 that combining the results of both algorithms leads to more schedulable systems (Total successes). Furthermore, as shown in Figure 7.2, there are some systems that the blocking agnostic algorithm schedules with fewer processors (it performs better in processor reduction). Hence a combined approach will lead to an improvement in processor reduction as well.

7.6 Summary and Future Work

In this paper we have proposed a heuristic blocking aware algorithm, for real-time multiprocessor systems, which extends a bin-packing algorithm with synchronization parameters. The algorithm allocates a task set onto the processors of a single-chip multiprocessor (multi-core) with shared memory. The objective of the algorithm is to decrease blocking times of tasks by means of allocating the tasks that directly or indirectly share resources onto appropriate processors. This generally increases shedulability of a task set and can lead to fewer processors compared to blocking-agnostic bin-packing algorithms.

Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling. This protocol introduces large amounts of blocking time overheads especially when the global resources are relatively long and the access ratio to them is high.

Our experimental results confirm that our algorithm mostly performs significantly better with respect to system schedulability and processor reduction. However, given a NP-hard problem, a bin packing algorithm may not achieve the optimal solution, i.e., our results show that, although our algorithm mostly performs significantly better, there still exist some cases that can only be solved by the blocking agnostic approach. Thus we show that a combination of both algorithms improves the results with respect to the total number of schedulable systems and processor reduction.

A future work will be extending our partitioning algorithm to other synchronization protocols, e.g., MSRP and FMLP for partitioned scheduling. Another interesting future work is to apply our approach to real systems and study the performance gained by the algorithm on these systems. In the domain of multiprocessor scheduling and synchronization our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Bibliography

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

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

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

[19] F. Nemati, M. Behnam, and T. Nolte. Multiprocessor synchronization and hierarchical scheduling. In *proceedings of 38th International Conference on Parallel Processing (ICPP'09) Workshops*, pages 58–64, 2009.

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

Chapter 8

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

Farhang Nemati, Thomas Nolte and Moris Behnam
In submission

Abstract

There are two main approaches to task scheduling on multiprocessor/multi-core platforms; 1) global scheduling, under which migration of tasks among processors is allowed, and 2) partitioned scheduling under which tasks are allocated onto processors and task migration is not allowed. Under global scheduling a higher utilization bound can be achieved, but in practice the overheads of migrating tasks is high. On the other hand under partitioned scheduling, besides simplicity and efficiency, existing scheduling and synchronization methods developed for uniprocessor platforms can more easily be extended to partitioned scheduling. However the partitioned scheduling protocols suffer from the problem of partitioning tasks among processors/cores which is a bin-packing problem. Therefore, several heuristic algorithms have been developed for partitioning a task set on multiprocessor platforms. However, these algorithms typically assume independent tasks while in practice real-time systems often contain tasks that share resources and hence may block each other.

In this paper we propose a blocking-aware partitioning algorithm which allocates a task set onto processors 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.

8.1 Introduction

Single-chip multiprocessors (multi-cores) are becoming defacto processors in practice. 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 distribute tasks fairly on cores to increase the performance. Real-time systems are typically multi threaded, hence they are easier to adapt to multi-core than single-threaded, sequential programs. If the tasks are independent, it is a matter of deciding on which core each task should execute. For real-time systems, from a practical and complexity point of view, a static and manual assignment of processors is often preferred for predictability reasons.

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

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

We have developed a heuristic partitioning algorithm [6], under which our system assumptions include presence of mutually exclusive shared resources. The heuristic partitions a system (task set) on an identical shared memory single-chip multiprocessor platform. The objective of the algorithm is to decrease blocking overheads by assigning tasks to appropriate processors (partitions). This consequently increases the schedulability of the system and may reduce the number of processors. Our heuristic identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, and extends the best-fit decreasing (BFD) bin-packing algorithm with blocking time parameters. In practice, industrial systems mostly use Fixed Priority Scheduling (FPS) protocols. To our knowledge the only synchronization protocol under fixed priority partitioned scheduling, for multiprocessor platforms, is Multiprocessor Priority Ceiling Protocol (MPCP) which was proposed by Rajkumar in [7]. Both our algorithm and an existing similar algorithm proposed in [5] assume that MPCP is used for lock-based synchronization. We have investigated MPCP in more details in [6]. The algorithm proposed in [5] is named the Synchronization-Aware Partitioning Algorithm (SPA), and our algorithm is named the Blocking-Aware Partitioning Algorithm (BPA). From now on we refer them as SPA and BPA respectively.

8.1.1 Contributions

The contributions of this paper are threefold:

- Firstly, we propose a blocking-aware heuristic algorithm to allocate tasks onto the processors of a single chip multiprocessor (multi-core) platform. The algorithm extends a bin-packing algorithm with synchronization parameters.

- Secondly, we implement our algorithm together with the best known existing heuristic [5][1]. The implementation is modular in which any new partitioned scheduling and synchronization protocol as well as any new partitioning heuristic can easily be inserted.

- Thirdly, we evaluate our algorithm together with the existing heuristic and compare the two approaches to each other as well as to an blocking-agnostic bin-packing partitioning algorithm, used as reference. The blocking-agnostic algorithm, in the context of this paper, refers to a bin-packing algorithm that does not consider blocking parameters to increase

[1]Best paper award at IEEE Real-Time Systems Symposium RTSS 2009

the performance of partitioning, although blocking times are included in the schedulability test.

The rest of the paper is as follows: we present the task and platform model in Section 9.2. We explain the existing algorithm (SPA) and present our partitioning algorithms (BPA) in Section 9.3. In Section 8.4 the experimental results of both algorithms are presented and the results are compared to each other as well as to the blocking-agnostic algorithm.

8.1.2 Related Work

A study of bin-packing algorithms for designing distributed real-time systems is presented in [8]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes. However, their partitioning method assumes independent tasks.

Liu et al. [9] present 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 paper) 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 (first-fit decreasing (FFD) algorithm) in [10] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k, for which both of following conditions, under the Earliest Deadline First (EDF) scheduling, hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i and D_i specify Worst Case Execution Time (WCET) and deadline of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0 & \text{if } t < D_i; \\ C_i + u_i(t - D_i) & \text{otherwise.} \end{cases}$$

The algorithm, however, assumes independent tasks while in practice tasks often share resources and therefore blocking time overheads must be considered while schedulability of tasks assigned to a processor is checked. Our algorithm not only considers resource sharing when distributing tasks but it tries to reduce blocking times as well. On the other hand their algorithm works under the EDF scheduling protocol while most existing real-time systems use fixed priority scheduling policies. Our proposed algorithm works under fixed priority scheduling protocols, although it can easily be extended to other policies.

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

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

In the context of multiprocessor synchronization, the first protocol was MPCP presented by Rajkumar in [7], which extends PCP to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned FPS. Our partitioning algorithm attempts to decrease blocking times under MPCP and consequently decrease worst case response times which in turn may reduce the number of needed processors. Gai et al. [11, 12] present MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors. The shared resources are classified as either (i) local resources that are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. In MSRP, tasks synchronize local resources using SRP [2], and access to global resources is guaranteed a bounded blocking time. Lopez et al. [13] present an implementation of SRP under P-EDF. Devi et al. [14] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [15] present FMLP (Flexible Multiprocessor Locking Protocol), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [16]. However, although in a longer version of [15][2], the blocking times have been calculated, but to our knowledge there is no schedulability test for FMLP.

Recently, a synchronization protocol under fixed priority scheduling, has been proposed by Easwaran and Andersson in [17], however, they focus on a global scheduling approach.

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

8.2 Task and Platform Model

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

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

8.3 The Blocking Aware Partitioning Algorithms

8.3.1 Blocking-Aware Partitioning Algorithm (BPA)

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

Considering the blocking factors of tasks under MPCP, tasks with more and longer global critical sections lead to more blocking times. This is also shown by experiments presented in [12]. Our goal is to (i) decrease the number of global critical sections by assigning the tasks sharing resources to the same partition as far as possible, (ii) decrease the ratio and time of holding global resources by assigning the tasks that request the resources more often and hold them longer to the same partition as long as possible.

In our previous work [18, 19] we have presented a partitioning algorithm in which tasks are grouped together based on task preferences and constraints. The algorithm partitions tasks based on a cost function which is derived from task preferences and constraints. In [19] the resource sharing is only local by means of allocating the tasks that directly or indirectly share resources onto the same processor. Tasks that directly or indirectly share resources are called *macrotasks*, e.g., if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask. However the algorithm fails if a macrotask does not fit in one processor (i.e., assuming that the tasks in the macrotask are the only tasks allocated on a processor, still it can not be scheduled by the processor). In [18] tasks belonging to the same macrotask can be allocated to different partitions (processors), thus the approach is more flexible but it introduces remote blocking overhead into the system. The goal of the algorithm is to put the tasks into appropriate partitions so that the costs are minimized. The algorithm may have different partitioning strategies, e.g., increasing cash hits, decreasing blocking times, etc. The strategy of partitioning may differ, depending on the nature of a system, and result in different partitions. In current work, however, we focus on decreasing remote blocking overheads of tasks which leads to increasing the schedulability of a task set and possibly reducing the number of processors needed for scheduling the task set.

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

The Algorithm

The algorithm performs partitioning of a task set in two rounds and the result will be the output of the round with better partitioning results. However, the algorithm performs a few common steps before starting to perform the rounds. Each round allocates tasks to the processors (partitions) in a different strategy. When a bin-packing algorithm allocates an object (task) to a bin (processor), it usually puts the object in a bin that fits it better, and it does not consider the unallocated objects that will be allocated after the current object. The rational behind the two rounds is that the heuristic tries to consider both past and future by looking at tasks allocated in the past and those that are not allocated yet. In the first round the algorithm considers the tasks that are not allocated to any processor yet; and tries to take as many as possible of the best related tasks (based on remote blocking parameters) with the current task. On the other hand, in the second round it considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. In the second round, the algorithm performs more like the usual bin packing algorithms (i.e., tries to find the best bin for the current object), although it considers the remote blocking parameters while allocating a task to a processor. Any time the algorithm performs schedulability test, for more precise schedulability analysis, it always performs response time analysis [20]. The common steps of the algorithm before the two rounds are performed are as follow:

1. Each task is assigned a weight. The weight of each task, besides its utilization, should depend on parameters that lead to potential remote blocking time caused by other tasks:

$$\begin{aligned} &w_i = u_i + \\ &\lceil (\sum_{\rho_i<\rho_k} \mathrm{NC}_{i,k}\beta_{i,k}\lceil \frac{T_i}{T_k}\rceil + \mathrm{NC}_i \max_{\rho_i\geq\rho_k} \beta_{i,k})/T_i\rceil \end{aligned} \tag{8.1}$$

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

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

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

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

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

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

$$v_{i,k} = \begin{cases} \mathrm{NC}_{i,k}\beta_{i,k}\lceil \frac{T_i}{T_k} \rceil & \rho_i < \rho_k; \\ \mathrm{NC}_i\beta_{i,k} & \rho_i \geq \rho_k \end{cases} \tag{8.2}$$

The rationale of the attraction function is to allocate the tasks that may remotely block a task, τ_i, to the same processor as of τ_i (in order of the amount of remote blocking overhead) as far as possible. Please notice, the definition of weight (Equation 8.1) and attraction function (Equation 8.2) guide the algorithm under MPCP. However, these function may differ under other synchronization protocols, e.g., MSRP, which have different remote blocking terms.

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

First Round

After the common steps the following steps are repeated within the first round until all tasks are allocated to processors (partitions):
1. All processors are ordered in their non-increasing order of utilization.
2. The object at the top of the mixed list is picked. **(i)** If the object is a task and it does not belong to a broken macrotask (it does not share any resource) it will be allocated onto the first processor that fits it (all processors are schedulable), beginning from the top of the ordered processor list (similar to blocking-agnostic BFD). If none of the processors can fit the task (at least one becomes unschedulable) a new processor is added to the list and the task is allocated onto it. In this case if one or more of the processors become(s) unschedulable this round of the algorithm fails and the algorithm moves to the second round.

(ii) If the object is an unbroken macrotask, all its tasks will be allocated onto the first processor that fits them (all processors are schedulable). If none of the processors can fit the macrotasks, it (all its tasks) will be allocated onto a new processor and in this case, if one or more of the processors becomes unschedulable the first round fails and the algorithm starts the second round. **(iii)** If the object is a task that belongs to a broken macrotask, the algorithm orders the tasks (those that are not allocated yet) within the macrotask in non-increasing order of attraction to the task based on equation 8.2. We call this list the *attraction list* of the task. The task itself will be on the top of its attraction list. The best processor for allocation is selected, which is the processor that fits the most tasks from the attraction list, beginning from the top of the list. As many as possible of the tasks from the attraction list are then allocated to the processor. If none of the existing processors can fit any of the tasks, a new processor is added and as many tasks as possible from the attraction list are allocated to the processor. However, if the new processor cannot fit any task from the attraction list, i.e., at least one of the processors become unschedulable, the first round fails and the second round is started.

Second Round

The following steps are repeated until all tasks are allocated to processors:
1. The object at the top of the mixed list is picked. **(i)** If the object is a task and it does not belong to a broken macrotask, this step is performed the same way as in the first round. If the second round fails here and if the first round has also failed the algorithm fails. **(ii)** If the object is an unbroken macrotask, in this the algorithm performs the same way as in the first round. If the second round fails and if the first round has also failed the algorithm fails. **(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 task; this list is ordered in non-increasing order of processors' attraction to the task based on equation 8.2, 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 those 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. And the second round of the algorithm fails if allocating the task to the new processor makes some of the processors unschedulable.

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

8.3.2 Synchronization-Aware Partitioning Algorithm (SPA)

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

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

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

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

$$\mathrm{Cost}(R_q) = \text{Global Overhead} - \text{Local Discount} \tag{8.3}$$

The Global Overhead is calculated as follows:

$$\text{Global Overhead} = \max(|Cs_q|)/\min_{\forall\tau_i}\{\rho_i\} \tag{8.4}$$

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

The Local Discount is defined as follows:

$$\text{Local Discount} = \max_{\forall\tau_i \text{ accessing } R_q}(\max(|Cs_{i,q}|)/\rho_i) \tag{8.5}$$

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

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

$$\mathrm{Cost}(\mathrm{mTask}_k) = \sum_{\forall R_q \text{ accessed by } \mathrm{mTask}_k} \mathrm{Cost}(R_q) \tag{8.6}$$

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

Firstly, as one can see, the SPA algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its utilization, i.e. the tasks are ordered in order of their utilization only. However, our algorithm assigns a weight (Equation 8.1) which besides the utilization includes the blocking terms as well. Secondly, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle (macrotask) is considered which could help to allocate tasks from a broken bundle to appropriate processors to decreases the blocking times. In our heuristic, we have defined an attraction function (Equation 8.2), which attempts to allocate the most attracted tasks from the current task's broken macrotask, on a processor. As the experimental evaluation in Section 8.4 shows, considering these issues can improve the partitioning significantly.

8.4 Experimental Evaluation and Comparison of Algorithms

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

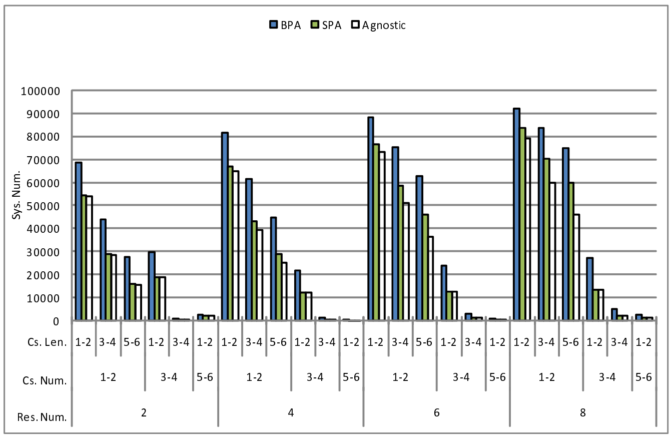


Figure 8.1: Total number of task sets each algorithm schedules. Workload: 3 processors, 3 tasks per processor

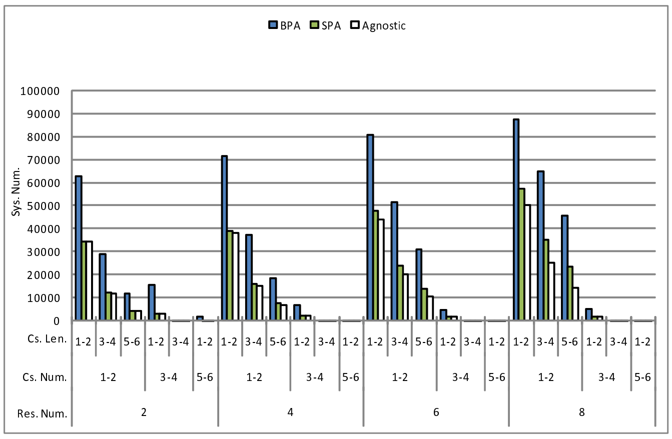


Figure 8.2: Total number of task sets each algorithm schedules. Workload: 3 processors, 6 tasks per processor

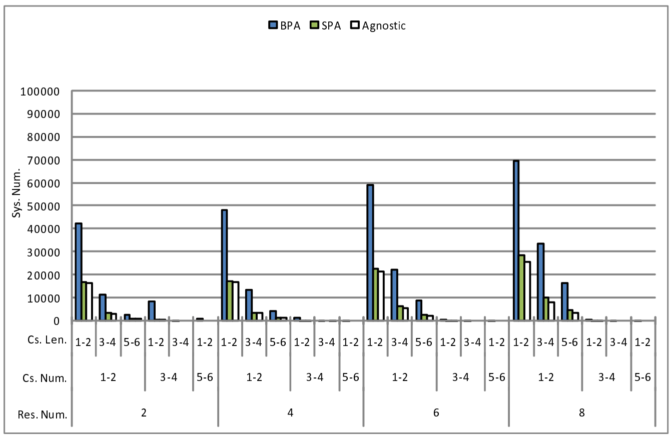


Figure 8.3: Total number of task sets each algorithm schedules. Workload: 3 processors, 9 tasks per processor

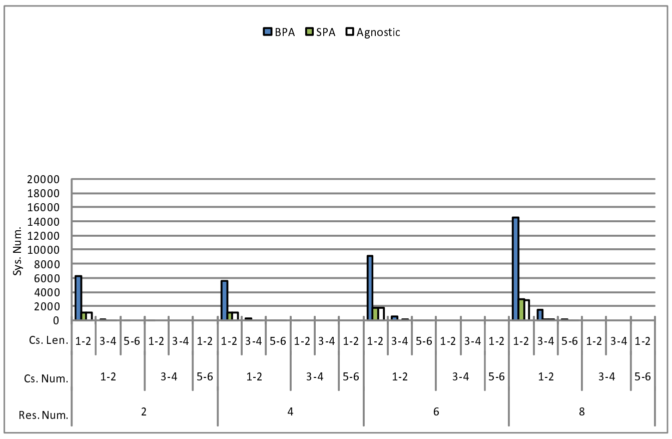


Figure 8.4: Total number of task sets each algorithm schedules. Workload: 6 processors, 6 tasks per processor

8.4.1 Experiment Setup

We generated systems (task sets) for different workloads; we denote workload as a defined number of fully utilized processors. Given a workload, the full capacity of each processor (utilization of 1) is randomly divided into a defined number of tasks utilizations. Usually for generating systems, utilization and periods are randomly assigned to tasks, and worst case execution times of tasks are calculated based on them. However, in our system generation, the worst case execution times (WCET) of tasks are randomly assigned and the period of each task is calculated based on its utilization and WCET. The reason is that we had to restrict that the WCET of a task not to be less than the total length of its critical sections. Since we have limited the maximum number of critical sections to 6 and the maximum length of any critical section to 6 time units, hence the WCET of each task should be greater than 36 (6×6) time units. The WCET of each task was randomly chosen between 36 and 150 time units. The system generation was based on different settings; the input parameters for settings are as follows:

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

2. The number of tasks per processor (3, 6 or 9 tasks per processor).

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

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

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

For each setting, we generated 100.000 systems, and combining the parameters of settings (432 different settings) the total number of systems generated for the experiment were $43.200.000$.

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

8.4.2 Results

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

The first aspect of comparison of the results from the algorithms is, given a number of systems, the total number of systems each algorithm successfully schedules (Figures 8.1, 8.2, 8.3 and 8.4). Figures 8.1, 8.2 and 8.3 represent the results for 3, 6 and 9 tasks per processor respectively. The vertical axis shows the total number of systems that the algorithms could schedule successfully. The horizontal axis shows three factors in three different lines; the bottom line shows the number of shared resources within systems (Res. Num.), the second line shows the number of critical sections per task (Cs. Num.), and the top line represents the length of critical sections within each task (Cs. Len.), e.g., Res. Num.=4, Cs. Num.=1-2, and Cs. Len.=1-2 represents the systems that share 4 resources, the number of critical sections per each task are between 1 and 2, and the length of these critical sections are between 1 and 2 time units.

As depicted in Figures 8.1, 8.2, 8.3 and 8.4, considering the total number of systems that each algorithm succeeds to schedule, our blocking-aware algorithm (BPA) performs better (can schedule more systems) compared to the SPA and the blocking-agnostic algorithm. However the SPA performs better than the blocking-agnostic algorithm. As shown in the figure, by increasing the number of resources, the number of successfully scheduled systems in all algorithms is increased. The reason for this behavior is that with fewer resources, more tasks share the same resource introducing more blocking overheads which leads to fewer schedulable systems. However, it is illustrated that the blocking-aware algorithms perform better as the number of resources is increased. It is also shown that increasing the number and/or the length of critical sections generally reduces the number of schedulable systems significantly. The reason is that more and longer critical sections introduce greater blocking overhead into the tasks making fewer systems schedulable.

As the number of tasks per processor is increased from 3 (Figure 8.1) to 6 (Figure 8.2) and to 9 (Figure 8.3), the BPA performs significantly better (i.e., schedules significantly more systems) than the SPA and blocking-agnostic bin-packing. However, as one can see, the SPA does not perform significantly better than the blocking-agnostic algorithm as the number of tasks per processor are increased. Increasing the number of tasks per processor lead to smaller tasks (tasks with smaller u_i). The BPA allocates tasks from a broken macrotask based on Equations 8.1 and 8.2, which are functions of the blocking parameters (the number and length of critical sections) as well as the size of the tasks. On the other hand, with the smaller size of tasks, the blocking parameters have a bigger role in these functions, hence more dependent tasks are allocated to the same processor. This lead to less blocking overhead and increased schedulability, hence more systems are scheduled by BPA as the tasks per processor are increased. On the other hand, in SPA, allocation of tasks from a broken macrotask is only based on their size, and this does not necessarily allocates highly dependent tasks to the same processor.

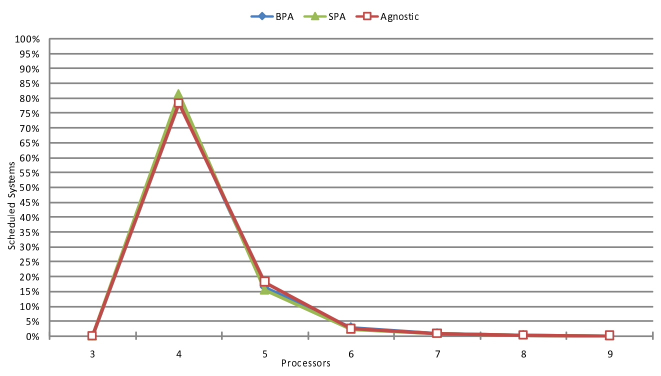


Figure 8.5: Percentage of systems each algorithm schedules, ordered by required number of processors for 3 tasks per processor

As the workload (the number of fully utilized processors) is increased, although the BPA still performs better than the SPA and the blocking-agnostic algorithm, generally the number of schedulable systems by all algorithms is significantly reduced (Figure 8.4). The reason for this behavior is that the number of tasks within systems are relatively many (36 tasks per each system in Figure 8.4) and the workload is high (6 fully utilized processors), and all the tasks within systems share resources. On the other hand, the MPCP is very pessimistic. This introduces a lot of interdependencies among tasks and consequently a huge amount of blocking overheads, making fewer systems schedulable. In practice in big systems with many tasks, not all of the tasks share resources, which leads to fewer interdependencies among tasks and less blocking times. However, we continued the experiment with higher workload in the same way as the other experiments (that all tasks share resources) to be able to compare the results with the previous results. We believe that re-

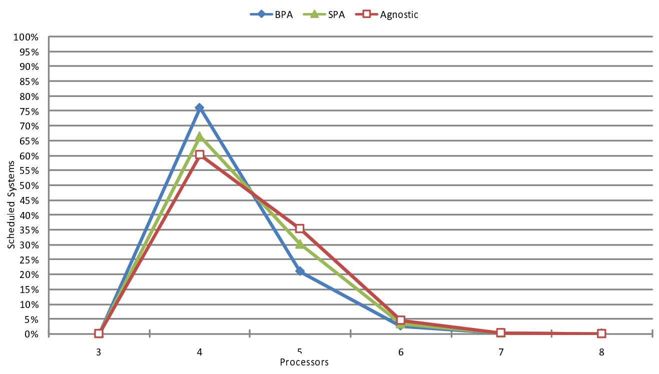


Figure 8.6: Percentage of systems each algorithm schedules, ordered by required number of processors for 6 tasks per processor

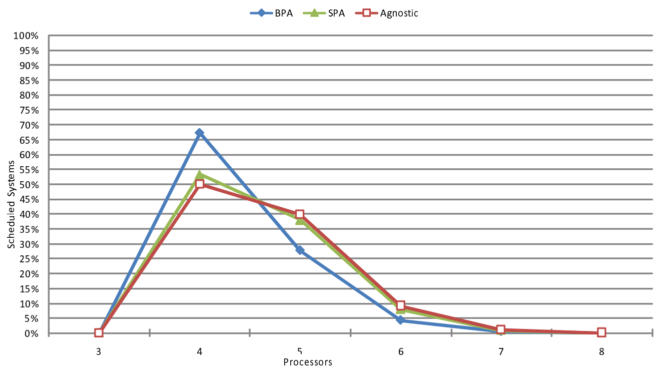


Figure 8.7: Percentage of systems each algorithm schedules, ordered by required number of processors for 9 tasks per processor

alistic systems, even with high workload and many tasks can benefit from our partitioning algorithm to increase the performance.

The second aspect for comparison of performance of the algorithms is the processor reduction aspect. To show this, for each algorithm, we ordered the total schedulable systems in order of the number of required processors. Figures 8.5, 8.6 and 8.7 illustrates the results for the workload of 3 fully packed processors and different number of tasks (3, 6 and 9) per processor. For each

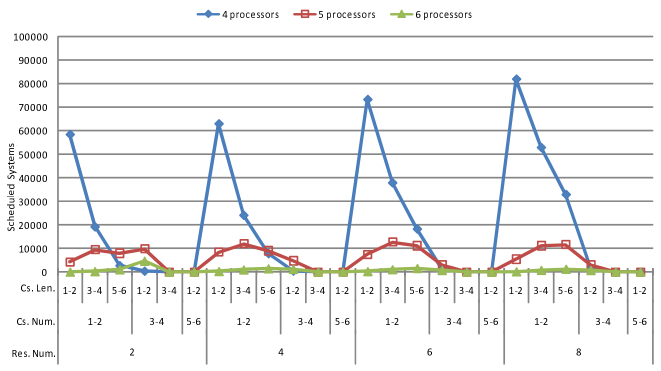


Figure 8.8: BPA. The number of systems scheduled by different number of processors. Workload: 3 processors, 6 tasks per processor.

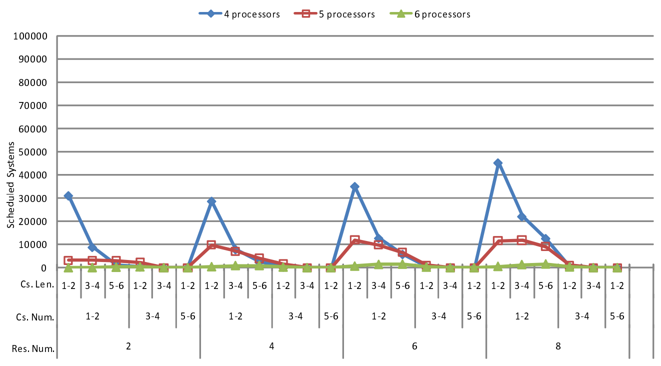


Figure 8.9: SPA. The number of systems scheduled by different number of processors. Workload: 3 processors, 6 tasks per processor.

algorithm, the schedulable systems by each number of processors are shown as percentage of the total scheduled systems by that algorithm. As the results show, for 3 tasks per processor all three algorithms perform almost the same (Figure 8.5), i.e., each algorithm schedules around 80% of its schedulable systems by 4 processors, 15% to 18% by 5 processors and less than 3% by 6 processors, etc. The reason is that the tasks are large (the utilization of a processor is divided among 3 task), thus the blocking-aware algorithms do not

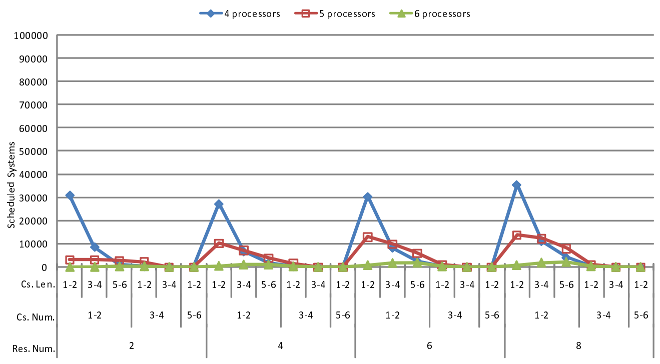


Figure 8.10: Blocking-agnostic. The number of systems scheduled by different number of processors. Workload: 3 processors, 6 tasks per processor.

have much possibility to increase the performance. However as the number of tasks per processor is increased (Figures 8.6 and 8.7 for 6 and 9 tasks per processor respectively), the blocking-aware algorithms, generally, perform better in processor reduction aspect. Especially the BPA, performs significantly better than the the SPA and the blocking-agnostic algorithm.

In the experiments we investigated the processor reduction aspect of the algorithms against the number of shared resources, the number of critical sections per task and the length of critical sections. The results show (Figures 8.8, 8.9 and 8.10) as the number of resources are increased the performance of the BPA is higher than the SPA in processor reduction. Furthermore, given a number of shared resources, increasing the number of critical sections per task and the length of them, the BPA performs better than SPA.

8.5 Conclusion

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

Since in practice most systems use fixed priority scheduling protocols, we have developed our algorithm under MPCP, the only existing synchronization protocol for multiprocessors (multi-cores) which works under fixed priority scheduling, although our algorithm can easily be extended to other synchronization protocols such as MSRP. The MPCP is pessimistic and introduces large amounts of blocking time overheads especially when the global resources are relatively long and the access ratio to them is high.

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

A future work will be extending our partitioning algorithm to other synchronization protocols, e.g., MSRP and FMLP for partitioned scheduling. Another interesting future work is to apply our approach to real systems and study the performance gained by the algorithm on these systems. In the domain of multiprocessor scheduling and synchronization our future work also includes investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Acknowledgments

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

Bibliography

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

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

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] F. Nemati, M. Behnam, and T. Nolte. Multiprocessor synchronization and hierarchical scheduling. In *proceedings of 38th International Conference on Parallel Processing (ICPP'09) Workshops*, pages 58–64, 2009.

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

Chapter 9

Paper D: A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors

Farhang Nemati and Thomas Nolte
In submission

Abstract

Multi-core platforms seem to be the way towards increasing performance of processors. Single-chip multiprocessors (multi-cores) are today the dominating technology for desktop computing. As the multi-cores are becoming the defacto processors, the need for new scheduling and resource sharing protocols has arisen. There are two major types of scheduling under multiprocessor/multi-core platforms. Global scheduling, under which migration of tasks among processors is allowed, and partitioned scheduling under which tasks are allocated onto processors and task migration is not allowed. The partitioned scheduling protocols suffer from the problem of partitioning tasks among processors/cores, which is a bin-packing problem. Heuristic algorithms have been developed for partitioning a task set on multiprocessor platforms. However, taking such technology to an industrial setting, it needs to be evaluated such that appropriate scheduling, synchronization and partitioning algorithms are selected.

In this paper we present our work on a tool for investigation and evaluation of different approaches to scheduling, synchronization and partitioning on multi-core platforms. Our tool allows for comparison of different approaches with respect to a number of parameters such as number of schedulable systems and number of processors required for scheduling. The output of the tool includes a set of information and graphs to facilitate evaluation and comparison of different approaches.

9.1 Introduction

The multiprocessor architectures are getting an increasing interest as the multi-cores offer higher performance and are becoming defacto processors in practice. This arises the need for new methods to take advantage of the multi-core platforms. A multi-core processor is a combination of two or more independent processors (cores) on a single chip, also called single-chip multiprocessors. The different cores are connected to a single shared memory via a shared bus. The cores typically have independent L1 caches and share an on-chip L2 cache. However, previously well-known and verified scheduling and synchronization protocols with an assumption of uniprocessors can not work properly on multi-cores, especially with the presence of shared resources. The industry has already begun to migrate towards multi-cores, although the existing scheduling and synchronization protocols are not yet mature enough to take advantage of the performance offered by multi-cores.

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

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

We have proposed a *blocking-aware* partitioning algorithm [6, 7]. The assumption include presence of mutually exclusive shared resources. The heuristic partitions a system (task set) on an identical shared memory single-chip multiprocessor (multi-core) platform. In the context of this paper the blocking-aware algorithm refers to an algorithm that attempts to decrease blocking overheads by assigning tasks to appropriate processors (partitions). This consequently increases the schedulability of the system and may reduce the number of processors. In contrast, a *blocking-agnostic* algorithm refers to a bin-packing algorithm that does not consider blocking parameters and does not attempt to decrease the blocking overhead, although blocking times are included in the schedulability test. Our blocking-aware algorithm identifies task constraints, e.g., dependencies between tasks, timing attributes, and resource sharing, and extends the best-fit decreasing (BFD) bin-packing algorithm with blocking time parameters. A similar heuristic has been proposed in [5].

As the scheduling and synchronization protocols together with partitioning algorithms are being developed, the industry needs to evaluate the different methods to choose appropriate methods and apply them in their applications. This arises the need for development of tools to facilitate investigation and evaluation of different approaches and compare them to each other according to different parameters. Hence, in this paper we present a tool which we have developed for evaluation of different scheduling, synchronization protocols coordinated with different partitioning algorithms. The output of the tool includes a set of information and graphs to facilitate evaluation and comparison of different approaches. We have implemented our blocking-aware partitioning algorithm together with the algorithm proposed in [5] and added them to the tool. The tool is modular making it possible to easily add any new scheduling, synchronization and partitioning algorithm. However, in this paper the focus of the tool has been directed to partitioned scheduling and synchronization approaches as well as partitioning heuristics while extending the tool to global scheduling methods remains as a future work.

The rest of the paper is as follows: we present the task and platform model in Section 9.2. We briefly explain our partitioning algorithms together with the existing algorithm in Section 9.3. In Section 9.4 we present the tool. In Section 9.5 we present some examples of the outputs of the tool in which we have compared different partitioning algorithms.

9.1.1 Related Work

A study of bin-packing algorithms for designing distributed real-time systems is presented in [8]. The method partitions software into modules to be allocated on hardware nodes. In their approach they use two graphs; a graph which models software modules and a graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of bins (processors) needed and the bandwidth required for the communication between nodes. However, their partitioning method assumes independent tasks.

Liu et al. [9] present a heuristic algorithm for allocating tasks in multi-core-based massively parallel systems. Their algorithm has two rounds; in the first round processes (groups of threads - partitions in this paper) 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 (first-fit decreasing (FFD) algorithm) in [10] for a set of sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines and the algorithm assigns the tasks to the processors in first-fit order. The algorithm assigns each task τ_i to the first processor, P_k for which both of following conditions, under the Earliest Deadline First (EDF) scheduling, hold:

$$D_i - \sum_{\tau_j \in P_k} DBF^*(\tau_j, D_i) \geq C_i$$

and

$$1 - \sum_{\tau_j \in P_k} u_j \geq u_i$$

where C_i and D_i specify Worst Case Execution Time (WCET) and deadline of task τ_i respectively, $u_i = \frac{C_i}{T_i}$, and

$$DBF^*(\tau_i, t) = \begin{cases} 0 & \text{if } t < D_i; \\ C_i + u_i(t - D_i) & \text{otherwise.} \end{cases}$$

The algorithm, however, assumes independent tasks.

In the work presented by Lakshmanan et al. in [5] they investigate and analyze two alternatives of execution control policies (suspend-based and spin-based remote blocking) under multiprocessor Multiprocessor Priority Ceiling Protocol (MPCP) [11]. They have developed a blocking-aware task allocation algorithm (an extension to BFD) and evaluated it under both execution control policies.

In their partitioning algorithm, the tasks that directly or indirectly share resources are put into what they call bundles (in this paper we call them macro-tasks) and each bundle is tried to be allocated onto a processor. The bundles that can not fit into any existing processors are ordered by their cost, which is the blocking overhead that they introduce into the system. Then the bundle with minimum cost is broken and the algorithm is run from the beginning. However, their algorithm does not consider blocking parameters when it allocates the current task to a processor, but only its size (utilization). Furthermore, no relationship (e.g., as a cost based on blocking parameters) among individual tasks within a bundle is considered which could help to allocate tasks from a broken bundle to appropriate processors to decrease the blocking times. However, according to our experimental results performed by the our tool, their heuristic performs slightly better than blocking-agnostic algorithm, and our algorithm performs significantly better than both.

In the context of multiprocessor synchronization, the first protocol was MPCP presented by Rajkumar in [11], which extends PCP [12] to multiprocessors hence allowing for synchronization of tasks sharing mutually exclusive resources using partitioned Fixed Priority Scheduling (FPS) protocols. Our partitioning algorithm attempts to decrease blocking times under MPCP and consequently decrease worst case response times which in turn may reduce the number of needed processors. Gai et al. [13, 14] present MSRP (Multiprocessor SRP), which is a P-EDF (Partitioned EDF) based synchronization protocol for multiprocessors. The shared resources are classified as either (i) local resources that are shared among tasks assigned to the same processor, or (ii) global resources that are shared by tasks assigned to different processors. In MSRP, tasks synchronize local resources using SRP [2], and access to global resources is guaranteed a bounded blocking time. Lopez et al. [15] present an implementation of SRP under P-EDF. Devi et al. [16] present a synchronization technique under G-EDF. The work is restricted to synchronization of non-nested accesses to short and simple objects, e.g., stacks, linked lists, and queues. In addition, the main focus of the method is on soft real-time systems.

Block et al. [17] present FMLP (Flexible Multiprocessor Locking Protocol), which is the first synchronization protocol for multiprocessors that can be applied to both partitioned and global scheduling algorithms, i.e., P-EDF and G-EDF. An implementation of FMLP has been described in [18]. However, although in a longer version of [17][1], the blocking times have been calculated, but to our knowledge there is no schedulability test for FMLP.

Recently, a synchronization protocol under fixed priority scheduling, has been proposed by Easwaran and Andersson in [19], but they focus on a global scheduling approach.

9.2 Task and Platform Model

The tool is capable of performing evaluations by both fixed priority and dynamic scheduling scheduling protocols. The tasks can also share resources. Thus the task model is assumed as a task set that consists of n sporadic tasks, $\tau_i(T_i, C_i, \rho_i, \{c_{i,p,q}\})$ and $\tau_i(T_i, C_i, \{c_{i,p,q}\})$ for dynamic and fixed priority scheduling protocols respectively, 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 (in fixed priority scheduling task model) as its priority. The tasks share a set of resources, $R = \{R_q\}$ which are protected using semaphores. The set of critical sections, in which task τ_i requests resources in R is denoted by $\{c_{i,p,q}\}$, where $c_{i,p,q}$ indicates the maximum execution time of the p^{th} critical section of task τ_i in which the task locks resource $R_q \in R$. Critical sections of tasks should be sequential or properly nested. The deadline of each job is equal to T_i. A job of task τ_i, is specified by J_i. The utilization factor of task τ_i is denoted by u_i where $u_i = C_i/T_i$.

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

9.3 Included Partitioning Algorithms

In this section we briefly present the partitioning algorithms we have developed and added to the tool. Please observe that the tool is flexible and any new partitioning algorithm can be added to the tool easily.

We have implemented and added three partitioning algorithms: (i) a blocking-aware algorithm which we proposed in [7], (ii) a similar blocking-

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

aware algorithm proposed in [5] and (iii) a blocking-agnostic algorithm. We have explained these algorithms in details in [7].

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

Our algorithm attempts to decrease the blocking times of tasks by partitioning a task set on processors based on heuristics. This generally increases the schedulability of a task set which may reduce the number of partitions (processors). The algorithm attempts to allocate the tasks that directly or indirectly share resources onto the same processor. Tasks that directly or indirectly share resources are called *macrotasks*, e.g., if tasks τ_i and τ_j share resource R_p and tasks τ_j and τ_k share resource R_q, all three tasks belong to the same macrotask.

The algorithm performs partitioning of a task set in two rounds and the result will be the output of the round with better partitioning results. Each round allocates tasks to the processors in a different strategy. When a bin-packing algorithm allocates an object (task) to a bin (processor), it usually allocates the object in a bin that fits it better, and it does not consider the unallocated objects that will be allocated after the current object. The rational behind the two rounds is that the heuristic tries to consider both past and future by looking at tasks allocated in the past and those that are not allocated yet. In the first round the algorithm considers the tasks that are not allocated to any processor yet; and tries to take as many as possible of the best related tasks (based on remote blocking parameters) with the current task. On the other hand, in the second round it considers the already allocated tasks and tries to allocate the current task onto the processor that contains best related tasks to the current task. In our tool, for more precise schedulability analysis, it always performs response time analysis [20] to check schedulability test of a task set.

We have also implemented and added the partitioning algorithm proposed in [5] which is similar to our blocking-aware algorithm. Their algorithm, similarly to our algorithm, attempts to group tasks in macrotasks and allocate each macrotask on a processor. The macrotasks that can not fit onto processors are ordered in the order of the cost of breaking them. The cost of breaking a macrotask is defined based on the estimated cost (blocking overhead) introduced into the tasks by transforming a local resource into a global resource (i.e., the tasks sharing the resource are allocated to different processors). The macrotask with minimum breaking cost is picked and is broken in two pieces such that the size of one piece is as close as the largest utilization available among processors. If the fitting is still not possible a new processor is added and the whole algorithm is repeated again. However, their algorithm does not consider blocking parameters when it allocates a task from a broken macrotask to a processor, but only its utilization, i.e. the tasks are ordered in order of their utilization only. On the other hand, our algorithm assigns a weight which besides the utilization includes the blocking terms as well. Besides, in our heuristic, we have defined an attraction function, which attracts the most attracted tasks to the picked task from its broken macrotask, and attempts to allocate them on the same processor.

9.4 The Tool

In this section we present our evaluation and partitioning tool.

9.4.1 The Structure

The tool has been developed in an object-oriented manner and every concept has been treated as an object, e.g., tasks, critical sections, resources, processors, etc.

We aimed to make the tool flexible to be able to easily add any partitioning, scheduling and synchronization (lock-based) algorithm. Thus, we have separated the development in three major parts (packages) as shown in Figure 9.1; *Scheduling Analysis* package, *Partitioning Algorithms* package, and *Task Generation* package.

When a partitioning algorithm attempts to assign a task to a processor it should test the schedulability of the all processors, hence it uses the classes in the scheduling analysis to perform the test. As the schedulability analysis is different depending on different scheduling or synchronization protocols (e.g., different blocking time terms), several classes are provided in the scheduling

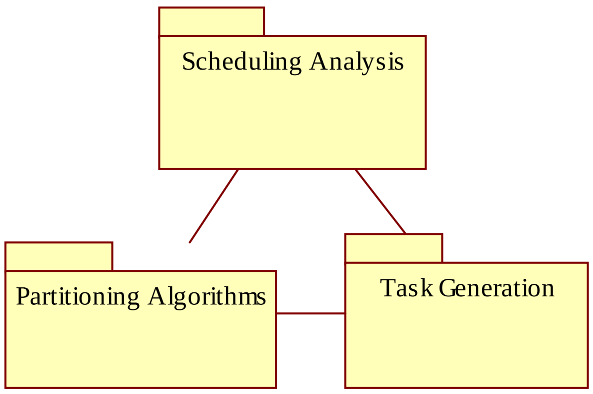


Figure 9.1: The three major packages

analysis package to facilitate the schedulability test in an object-oriented manner. This makes the package reusable and extendable as the new scheduling and synchronization protocols are added.

The Scheduling Analysis package

This package contains classes associated with scheduling protocols, e.g., RM, as well as synchronization protocols, e.g., MPCP. Besides classes used for schedulability test for each scheduling protocol, the package contains classes to facilitate calculation of blocking times of tasks to be used in the schedulability analysis. Under a multiprocessor synchronization protocol, any task, τ_i, may face mainly two types of blocking times; (i) the local blocking times by interference from the lower priority tasks assigned to the same processor as τ_i's processor, (ii) the remote blocking times introduced by the tasks (with any priority) assigned to different processors than of τ_i's processor.

To easily and in a modular manner calculate the local and remote blocking times of each task, in this package a task class includes a local processor class and a set of remote processors, i.e., the local processor is the processor that the task is assigned to and the rest of processors are contained in the remote processors set. Under partitioned scheduling approaches the total blocking time of a task is the summation of the local and the global blocking terms.

Depending on the used synchronization protocol, the local and remote blocking terms of the task on each processor may be different, e.g., under MPCP the total blocking time (B_i) of task τ_i, consists of five blocking terms, of which one is local and four different remote blocking terms [11]. Figure 9.2 shows the local processor and the remote processors associated with the task class. This structure facilitates calculating each term of the blocking time of a task from each processor, i.e., the local blocking terms introduced from the local tasks (tasks allocated on the same processor as τ_i's processor) are calculated using local processor class and the remote blocking terms from remote tasks (tasks allocated on a different processor than of τ_i's processor) are calculated by the remote processors. Each scheduling and synchronization protocol uses these classes differently as they may have different blocking time terms.

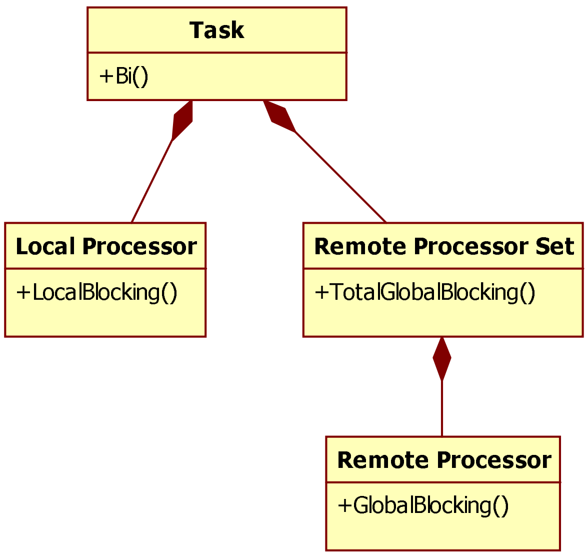


Figure 9.2: The local and remote blocking times calculation

For more precise schedulability test in scheduling protocols, in this package response times analysis is performed by calculating the worst case response time of each task. Thus, another output of the package is the worst case response times of the tasks within a task set.

The Partitioning Algorithms package

This package is used to partition a task set to be allocated onto a multiprocessor platform. Any partitioning algorithm can easily be plugged into the package. The only requirement of the a new algorithm is that it has to have the task set, the target scheduling and synchronization protocols as inputs. The output of the algorithm will be a set processors each of which contains the allocated tasks. Each allocated task contains its calculated worst case response time.

The purpose of a partitioning may be different. The goal can be to plug in a partitioning heuristic to reduce the required number of processors. On the other hand, the goal of a partitioning algorithm may be to distribute the tasks fairly onto processors to balance the utilization of processors, thus, the output of each algorithm include the utilization of each processor as well.

We have developed a blocking-aware partitioning heuristic (Section 9.3). We have implemented our algorithm together with a similar blocking-aware algorithm and plugged into the partitioning algorithms package. The objective of those algorithms is to reduce the blocking times of tasks by co-allocating the tasks sharing the same resources as far as possible. Furthermore, we have implemented a BFD bin-packing algorithm (blocking-agnostic algorithm) and inserted this algorithm into the package.

Any partitioning algorithm needs to test schedulability of each processor each time it allocates a task or a group of tasks on a processor. The algorithms, within this package use the schedulability analysis provided in the scheduling analysis package. This separates the partitioning algorithms from the schedulability analysis making it easy to develop any new partitioning algorithms and scheduling protocols independently and insert them into the tool.

The Task Generation package

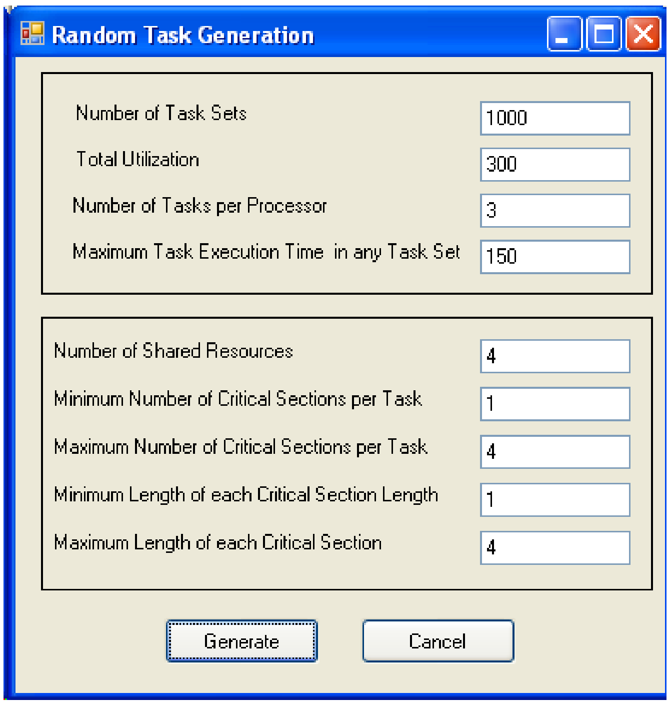
This package is used for task set generation in two different ways. The tool can be used for two different purposes; (i) the schedulability analysis and partitioning of a task set defined by a user, or (ii) evaluation and comparison of different scheduling, synchronization and partitioning algorithms according to a number of randomly generated task sets. This package provides two different ways of task set generation. One way is to provide the user to enter the tasks, critical sections, resources, and relationships between tasks and resources. In this case the tool partitions the task set using the selected partitioning algorithm and the selected schedulability test. The second way is to generate a number of task sets according to several given parameters (Figure 9.3). In this case the tool uses the generated task sets to perform evaluation and comparison of different scheduling, synchronization and partitioning algorithms.

As shown in Figure 9.3, for the random task set generation two groups of parameters are provided. The first group includes the desired number of task sets, total workload, the number of tasks per processor and maximum execution time of each task (maximum WCET). The minimum WCET is limited by the maximum number and length of critical sections per each task, e.g., with maximum number of critical section set to 5 and maximum length of any critical section set to 6 the minimum execution time of any task will be 30. The second group of parameters for the task sets are resource sharing parameters, i.e., the number of resources shared among tasks of each task set, minimum number, maximum number of critical sections per each task, minimum length and maximum length of each critical section. The random task generation provides the possibility of generating task sets by combination of the parameters which can be used to evaluate algorithms.

The random task generation process performed by this package is as follows.

The total workload presents the number of fully utilized processors (e.g., Total Utilization $= 300$ means 3 fully utilized processors). The utilization of each of the processors (utilization $= 100\%$) is randomly divided among the given number of tasks per processor, e.g., for 3 tasks per processor a possible utilization assignment of the tasks of a processor can be 25%, 45% and 30% respectively. Usually for generating task sets, utilization and periods are randomly assigned to tasks and worst case execution times of tasks are calculated based on them. However, in our random task set generation package, the worst case execution times (WCET) of tasks are randomly assigned and the period of each task is calculated based on its utilization and WCET. The reason is that we had to restrict that the WCET of a task not to be less than the maximum length of its critical sections restricted by the maximum number of critical sections per each tasks and the maximum length of each critical section.

The number of critical sections for each task is randomly chosen from the range between the defined minimum and maximum number of critical sections. The length of each critical section is chosen the same way. For each critical section the accessed resource is randomly chosen from the defined resources, e.g, given the number of resources $= 2$ (R_1 and R_2), the resource accessed by any critical section is randomly chosen from $\{R_1, R_2\}$.



Random Task Generation

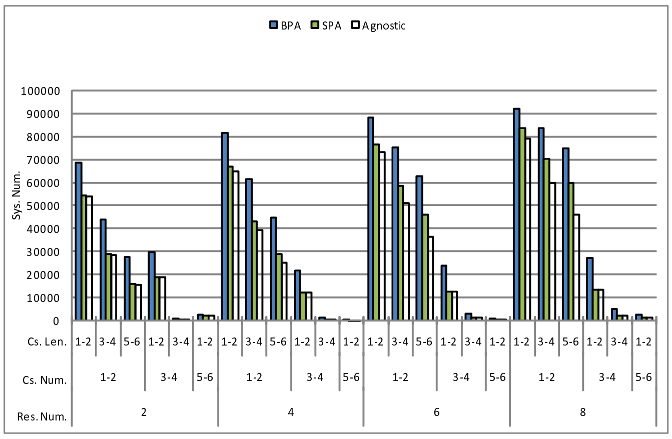
Number of Task Sets	1000
Total Utilization	300
Number of Tasks per Processor	3
Maximum Task Execution Time in any Task Set	150
Number of Shared Resources	4
Minimum Number of Critical Sections per Task	1
Maximum Number of Critical Sections per Task	4
Minimum Length of each Critical Section Length	1
Maximum Length of each Critical Section	4

Figure 9.3: The randon task generation

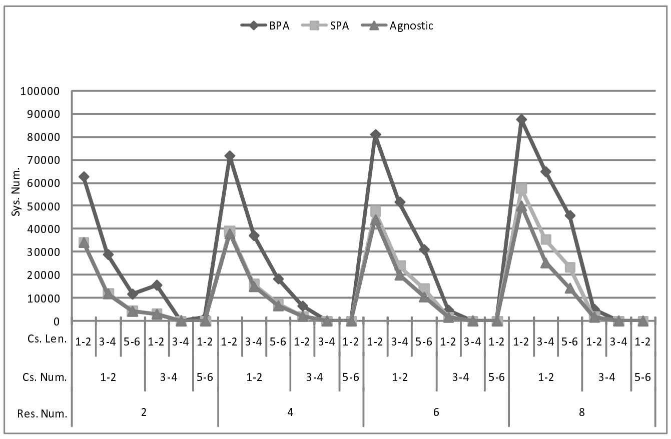
9.5 Example: An Evaluation and Comparison of Partitioning Algorithms

In this section we present an experimental example which we have performed by our tool. In this example we have compared the outputs of the two blocking-aware heuristics as well as the blocking-agnostic bin-packing algorithm. To ease referring to our blocking-aware algorithm and the similar algorithm proposed in [5] in this example we refer them as BPA and SPA respectively. The scheduling and synchronization protocols under which the algorithms were evaluated were the RM and the MPCP respectively.

For a number of systems (task sets), we have compared the performance of the algorithms in two aspects; (i) Given a number of systems, what is the total number of systems that each of the algorithms can schedule, (ii) what is the processor reduction aspect of the two algorithms.



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



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

Figure 9.4: Output the tool for performance of the algorithms with respect to task sets each algorithm schedules.

9.5.1 Task Set Generation

Using the random task set generation, we generated systems (task sets) for different workloads. Since we have limited the maximum number of critical sections to 6 and the maximum length of any critical section to 6 time units, hence the execution time of each task should be greater than 36 time units. The maximum execution time of any task was defined as 150 time units.

We ran the evaluation tool with respect to different configurations with the input parameters as follows.

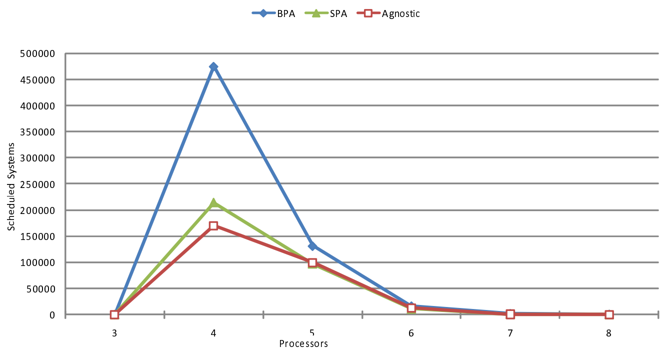
- Workload (3, 4, 6, or 8 fully utilized processors).
- The number of tasks per processor (3, 6 or 9 tasks per processor).
- The number of resources (2, 4 or 6).
- The range of the number of critical sections per task (1 to 2, 3 to 4 or 5 to 6 critical sections per task).
- The range of length of critical sections (1 to 2, 3 to 4, or 5 to 6).

For each configuration, we chose the number of systems to be 100.000, and combining the parameters of configurations (432 different configurations), the total number of systems generated for the experiment sums up to 43.200.000.

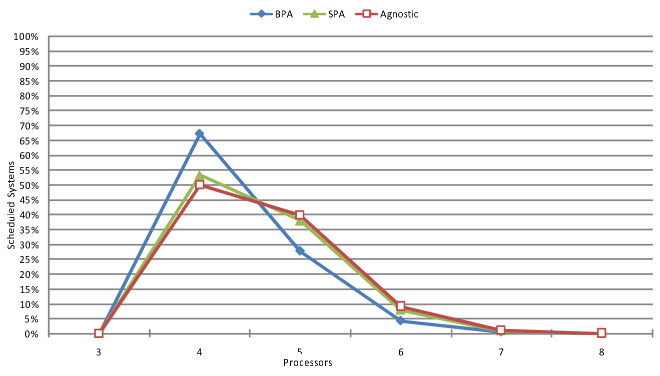
With the generated systems we were able to evaluate the partitioning algorithms with respect to different factors, i.e., various workloads (number of fully utilized processors), number of tasks per processor, number of shared resources, number of critical sections per task, and length of critical sections.

Figure 9.4 shows the examples of the output regarding the number of schedulable systems by each algorithm, i.e., the first aspect of comparison of the partitioning algorithms. The vertical axis shows the total number of systems that the algorithms could schedule successfully. The horizontal axis shows three factors in three different lines; the bottom line shows the number of shared resources within systems (Res. Num.), the second line shows the number of critical sections per task (Cs. Num.), and the top line represents the length of critical sections within each task (Cs. Len.), e.g., Res. Num.=4, Cs. Num.=1-2, and Cs. Len.=1-2 represents the systems that share 4 resources, the number of critical sections per each task are between 1 and 2, and the length of these critical sections are between 1 and 2 time units.

The second aspect for comparison of performance of the algorithms is the processor reduction aspect. The examples of the output of the tool for illustrating this aspect are shown in Figure 9.5. For each algorithm, the total number



(a) 6 tasks per processor



(b) 9 tasks per processor (as percentage of the total scheduled systems)

Figure 9.5: Task sets each algorithm schedules, ordered by required number of processors. Workload: 3 processors.

of schedulable systems are ordered in order of the number of required processors. The schedulable systems of an algorithm with each number of processors can also be illustrated as percentage of the total scheduled systems.

9.6 Conclusion

In this paper we have presented our work on a tool that we have developed for evaluation of different scheduling and synchronization protocols coordinated with different partitioning algorithms. The output of the tool includes a set of information and graphs to facilitate evaluation and comparison of different approaches.

Moreover, we briefly presented our blocking-aware partitioning algorithm proposed in [7] together with a similar algorithm proposed in [5]. We have implemented the two approaches together with an usual bin-packing (blocking-agnostic) algorithm and added all three approaches to the tool. The tool has the possibility to evaluate and compare different multiprocessor scheduling, synchronization and partitioning algorithms. The tool has been developed in an object-oriented manner making the tool flexible. Any new scheduling, synchronization or partitioning algorithm can be developed and added to the tool easily. We have presented a few examples of the illustrated outputs of the tool for evaluation and comparison of the partitioning algorithms included in the tool.

The focus of the tool is currently multiprocessor partitioned scheduling protocols and extending the tool to global scheduling and synchronization protocols remains as a future work. Another plan for future work is to extend the tool to simulate the execution of a task set on a multi-core platform and visualize the simulated timing behavior of the task set. In the domain of multiprocessor scheduling and synchronization we will also work on investigating global and hierarchical scheduling protocols and appropriate synchronization protocols.

Bibliography

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

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

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

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

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

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

[7] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In *In submition*, 2010.

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

[19] 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.

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