Advanced Scheduling Techniques in Virtualized Multi-core Systems

PhD Thesis

PhD Student: Georgiana-Irina Macariu
Scientific Advisor: Prof. Dr. Eng. Vladimir-Ioan Creţu

Computer Software and Engineering Department
"Politehnica" University of Timişoara

March 2011

Contents

List of Figures

List of Tables

List of Acronyms

ACSR	Algebra of Communicating Shared Resources
ACSR-VP	Algebra of Communicating Shared Resources with Value Passing
BHP	Bounded Blocking with High Parallelism
BROE	Bounded-delay Resource Open Environment
BWI	Bandwidth Inheritance
CBS	Constant Bandwidth Server
DM	Deadline Monotonic
DP-FAIR	Deadline Partitioning Fair
EDF	Earliest Deadline First
EDF-US	Earliest Deadline First with Utilization Separation
EDF-DS	Earliest Deadline First with Density Separation
EDZL	Earliest Deadline until Zero Laxity
FMLP	Flexible Multiprocessor Locking Protocol
HSF	Hierarchical Scheduling Framework
HSRP	Hierarchical Stack Resource Policy
LB-PCP	Limited Blocking Priority Ceiling Protocol
LCM	Least Common Multiple
LLF	Least Laxity First
MHSP	Multiprocessor Hierarchical Synchronization Protocol
MPR	Multiprocessor Periodic Resource
MPSoC	Multiprocessor System-on-Chip
MSC	Message Sequence Charts
M-BWI	Multiprocessor Bandwidth Inheritance
M-CBS	Multiprocessor Constant Bandwidth Server
M-PCP	Multiprocessor Priority Ceiling Protocol
M-SRP	Multiprocessor Stack Resource Policy

M-TBS	Multiprocessor Total Bandwidth Server
MTF	Major Time Frame
MTT	Multi-threaded Task
QCIF	Quarter Common Intermediate Format
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
P-HRP	Parallel Hierarchical Resource Policy
P-PCP	Parallel Priority Ceiling Protocol
RTA	Response Time Analysis
RM	Rate Monotonic
RM-US	Rate Monotonic with Utilization Separation
SG	Server Generator
SIRAP	Subsystem Integration and Resource Allocation Policy
SRP	Stack Resource Policy
SWA	Stopwatch automaton
TBS	Total Bandwidth Server
TGS	Task Generator Stopwatch automaton
TGT	Task Generator
TPN	Time Petri Nets

This thesis is based on and extends the work and results presented in the following publications:

1. **G. Macariu** and V. Cretu, *Enabling Parallelism and Resource Sharing in Multi-core Component-based Systems*, in ISORC '11: Proceedings of the 14th IEEE International Symposium on Object/Component/Service-oriented Real- time Distributed Computing, IEEE Computer Society, March 2011, in print.

2. **G. Macariu**, *A Model Checking Approach for Multi-core Time Partitions Design*, in ICESS '10: Proceedings of the 7th IEEE International Conference on Embedded Software and Systems, pp. 1910–1917, IEEE Computer Society, June 2010, ISBN 978-0-7695-4108-2. (IEEE eXplore, SCOPUS, ACM, DBLP)

3. **G. Macariu** and V. Cretu, *Timed Automata Model for Component-based Real-Time Systems*, in ECBS '10: Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, pp. 121–130, IEEE Computer Society, March 2010, ISBN 978-1-4244-6537-8. (IEEE eXplore, SCOPUS, ACM, DBLP)

4. **G. Macariu** and V. Cretu, *Model-based Analysis of Contract-based Real-Time Scheduling*, in SEUS '09: Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems, Lecture Notes in Computer Science, vol. 5860, pp. 227–239, Springer Berlin/ Heidelberg, November 2009, ISBN 978-3-642-10264-6, ISSN 0302-9743 (Print) 1611-3349 (Online). (ISI Proceedings, Lecture Notes in Computer Science, SCOPUS, ACM, DBLP)

5. D. Tudor, **G. Macariu**, C. Jebelean, V. Cretu, *Towards a Load Balancer Architecture for Multi-Core Mobile Communication Systems*, SACI '09: Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics, pp. 391 - 396, IEEE Computer Society, May 2009, ISBN 978-1-4244-4477-9. (ISI Proceedings, IEEE eXplore, SCOPUS, DBLP)

6. D. Tudor, **G. Macariu**, C. Jebelean, V. Cretu, *Load Balancing in Modern Multi-Core Mobile Communication Systems*, CSCS 17: Proceedings of 17th International Conference on Control Systems and Computer Science, Vol. 1, pp. 255-261, May 2009, ISSN: 2066-4451.

Prior to the thesis, two PhD Reports prepared this work:

1. **G. Macariu**, *Model-based Schedulability Analysis of Virtualized Multi-core Systems*, PhD Report 1, Computer Science and Engineering Department Politehnica University of Timişoara, Romania, January 2010.

2. **G. Macariu**, *Data Sharing in Multi-core Real-time Systems*, PhD Report 2, Computer Science and Engineering Department Politehnica University of Timişoara, Romania, November 2010.

This thesis has been partially supported from the ICT-eMuCo (grant no. 216378) a European project supported under the Seventh Framework Programme (7FP) for research and technological development.

Full publications list

Book chapters

1. D. Petcu, **G. Macariu**, A. Cârstea, and M. Frîncu, *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, chap. Service-Oriented Symbolic Computations, pp. 1053–1075, Information Science Publishing, 1 ed., February 2010, ISBN 978-1-6152-0686-5

Papers in ISI classified journals and series

A. LNCS (ISI proceedings)

1. **G. Macariu** and V. Creţu, Model-Based Analysis of Contract-Based Real-Time Scheduling, in *SEUS '09: Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pp. 227–239, Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-10264-6
2. A. Cârstea, **G. Macariu**, D. Petcu, and A. Konovalov, Pattern Based Composition of Web Services for Symbolic Computations, in *ICCS '08: Proceedings of the 8th International Conference on Computational Science, Part I*, pp. 126–135, Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 978-3-540-69383-3
3. A. Cârstea, M. Frîncu, A. Konovalov, **G. Macariu**, and D. Petcu, On service-oriented symbolic computing, in *PPAM'07: Proceedings of the 7th International Conference on Parallel processing and Applied Mathematics*, pp. 843–851, Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 3-540-68105-1, 978-3-540-68105-2

B. IEEE (ISI proceedings)

1. D. Tudor, **G. Macariu**, W. Schreiner, and V. Cretu, Experiments on a grid layer prototype for shared data programming model, in *SACI '09: Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics*, pp. 441–446, May 2009, ISBN 978-1-4244-4477-9
2. D. Tudor, **G. Macariu**, C. Jebelean, and V. Cretu, Towards a load balancer architecture for multi-core mobile communication systems, in *SACI '09: Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics*, pp. 391–396, May 2009, ISBN 978-1-4244-4477-9
3. **G. Macariu**, D. Tudor, and V. Cretu, Designing a Dynamic Replication Engine for Grid Shared Data Programming, in *SYNASC '08: Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 402–409, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3523-4

4. A. Cârstea, **G. Macariu**, M. Frîncu, and D. Petcu, Workflow Management for Symbolic Grid Services, in *SYNASC '08: Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 373–379, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3523-4
5. **G. Macariu**, A. Cârstea, M. Frîncu, and D. Petcu, Towards a Grid Oriented Architecture for Symbolic Computing, in *ISPDC '08: Proceedings of the 2008 International Symposium on Parallel and Distributed Computing*, pp. 259–266, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3472-5
6. **G. Macariu** and D. Petcu, Parallel Multiple Polynomial Quadratic Sieve on Multi-Core Architectures, in *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 59–65, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3078-8
7. A. Cârstea, **G. Macariu**, M. Frîncu, and D. Petcu, Composing Web-Based Mathematical Services, in *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 327–334, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3078-8
8. A. Cârstea, M. Frîncu, **G. Macariu**, D. Petcu, and K. Hammond, Generic Access to Web and Grid-based Symbolic Computing Services: the SymGrid-Services Framework, in *Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, pp. 143–150, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2936-4
9. D. Tudor, **G. Macariu**, and V. Cretu, A Performance Analysis on Message Passing Tools for the Grid, in *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 317–322, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3078-8
10. D. Tudor, **G. Macariu**, and V. Cretu, Dynamic Policy Based Replication on the Grid, in *SACI '07: Proceedings of the 4th International Symposium on Applied Computational Intelligence and Informatics*, pp. 77–81, May 2007, ISBN 1-4244-1234-X

Papers in international journals, indexed in international databases (SCOPUS, ACM, DBLP)

1. A. Cârstea, M. Frîncu, **G. Macariu**, and D. Petcu, Event-Based Simulator for SymGrid-Services Framework, *International Journal of Grid and Utility Computing*, in print
2. D. Tudor, **G. Macariu**, W. Schreiner, and V. I. Cretu, Experiences on grid shared data programming, *International Journal of Grid and Utility Computing*, vol. 1: pp. 296–307, August 2009, ISSN 1741-847X

Papers in other international refereed journals

1. M. Frîncu, **G. Macariu**, and A. Cârstea, Dynamic and Adaptive Workflow Execution Platform for Symbolic Computations, *Pollack Periodica*, vol. 4 (1): pp. 145–156, September 2009, ISSN 1788-1994

2. A. Cârstea and **G. Macariu**, Towards a Grid Enabled Symbolic Computation Architecture, *Pollack Periodica*, vol. 3 (2): pp. 15–26, September 2008, ISSN 1788-1994
3. D. Petcu, A. Cârstea, **G. Macariu**, and M. Frîncu, Service-oriented Symbolic Computing with SymGrid, *Scalable Computing: Practice and Experience*, vol. 9 (2): pp. 111–125, June 2008

Papers indexed in international databases (SCOPUS, IEEExplore, ACM, DBLP)

1. **G. Macariu** and V. Cretu, Enabling Parallelism and Resource Sharing in Multi-core Component-based Systems, in *ISORC '11: Proceedings of the 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, IEEE Computer Society, March 2011, in print
2. **G. Macariu**, A Model Checking Approach for Multi-core Time Partitions Design, in *ICESS '10: Proceedings of the 7th IEEE International Conference on Embedded Software and Systems*, pp. 1910–1917, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-0-7695-4108-2
3. **G. Macariu** and V. Cretu, Timed Automata Model for Component-Based Real-Time Systems, in *ECBS '10: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pp. 121–130, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-0-7695-4005-4
4. D. Tudor, **G. Macariu**, W. Schreiner, and V. Cretu, Shared Data Grid Programming Improvements Using Specialized Objects, in *CISIS '10: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pp. 398–403, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-0-7695-3967-6
5. A. Cârstea, **G. Macariu**, M. Frîncu, and D. Petcu, Description and Execution of Patterns for Symbolic Computations, in *SYNASC '09: Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 197–204, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3964-5
6. **G. Macariu**, M. Frîncu, A. Cârstea, D. Petcu, and A. Eckstein, Redesigning Parallel Symbolic Computations Packages, in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, p. 417, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2944-5

Papers in international conference proceedings

1. D. Tudor, **G. Macariu**, C. Jebelean, and V. Cretu, Load Balancing in Modern Multi-Core Mobile Communication Systems, in *CSCS 17: Proceedings of 17th International Conference on Control Systems and Computer Science*, vol. 1, pp. 255–261, Bucharest, Romania, May 2009, ISSN 2066-4451

Technical reports

1. **G. Macariu**, Model-based Schedulability Analysis of Virtualized Multi-core Systems, PhD Report 1, Computer Science and Engineering Department "Politehnica" University of Timişoara, Timişoara, Romania, January 2010

2. **G. Macariu**, Data Sharing in Multi-core Real-time Systems, PhD Report 2, Computer Science and Engineering Department "Politehnica" University of Timişoara, Timişoara, Romania, November 2010

Research Grants

International grants

1. **eMuCo** - FP7 Embedded Multi-Core Processing for Mobile Communication (2008 – 2010): Project founded by European Commission Seventh Framework Programme - member
2. **SCIEnce** - FP6-2005-Infrastructure-5, Symbolic Computation in Europe (2006 – 2011): Project founded by European Commission Sixth Framework Programme - member

National grants

1. **Falx Daciae** - Research and development grant from EU funds, from the Regional Competitiveness and Employment Objective (2010-2012) - member
2. **MedioGRID** - Research grant (CEEX) funded by the Romanian Ministry of Research (2006-2008) - member

1. INTRODUCTION

1.1. Multi-core Processors for Real-Time Systems

Not a very long time ago, the main way of facing the increasing demand for computational power consisted in processor clock acceleration. But higher clock speed results also in higher power consumption. One way out of this dilemma is believed to be *multi-core processors* or *chip multiprocessors*. Even if this trend affected at first the server and PC systems, multi-core processors will eventually be commonplace for many categories of devices. For example, nowadays mobile devices have become true mobile computation platforms that handle graphics, videos and many more applications, occasionally interrupted by a phone call.

The advent of the multi-core revolution can be placed around May 2004, when Intel canceled two single-core designs as they realized that these designs require a significant engineering effort to deal with the excessive heat dissipated by such chips. Instead, they started work on Intel Smithfield, the first dual-core Intel processor, released on the 25th of May 2005. Intel was also rushed by the need to respond to Advanced Micro Devices (AMD) plans with the dual-core Opteron processor, also launched in May 2005. Nowadays, processors with eights cores can be found in low-cost desktop systems while at the same time, prototypes with up to 48 cores are developed [97].

In the last years, this multi-core revolution affected also the world of embedded processors and many hardware producers spend considerable amounts of resources for designing such parallel architectures. An example is given by the ARM Cortex A15 MPCore processor consisting of up to four cores [7], designed for mobile phones and other embedded systems. Furthermore, on December 16, 2010, LG Electronics launched LG Optimus 2X, the first dual-core smartphone [6].

Once the computational power and energy saving problem is solved, for applications to fully benefit of the power provided by multi-cores, they must be redesigned and special care must be paid to scheduling and processor allocation issues of these applications. For throughput-oriented applications this area has been widely covered, but there remains an important class of applications – namely *real-time applications* – where processor allocation is still an obstacle, as these applications require not only high performance but also timing correctness. Without these, that occasional phone call would be missed.

The issue of ensuring timing correctness is strongly related to the real-time scheduling problem, which can be split in two subproblems:

(1) the *run-time scheduling* problem: find a schedule for the tasks of each application in the system such that all timing constraints are met,
(2) the *schedulability* problem: given a set of tasks, find if the tasks will meet their deadlines if executed on a given processor platform.

Both these problems have been thoroughly researched for uniprocessor systems, producing a considerably variety of publications and applications. However, there are still many open problems related to multiprocessor and multi-core scheduling. In this thesis, we focus on the schedulability problem of applications running on multi-core and identical multiprocessor platforms, in which all processors have the

same speed.

Another key aspect that appears when referring to real-time systems is that real-time applications must not be interfered by other applications sharing the same processing platform. The solution out of this dilemma is given by *virtualization*, which provides partitioning capabilities on top of multi-core processors.

Basically, virtualization ensures temporal and spatial isolation between different applications. In real-time scheduling theory, temporal isolation is ensured through hierarchical scheduling techniques. These techniques offer flexibility as they are able to enforce scheduling policies at runtime.

When choosing what kind of scheduler to use on a real-time multi-core/multprocessor system, there are two main options, each with its advantages and disadvantages. One can either use a *partitioned* approach and statically assign tasks to processors, scheduling each set of tasks using some uniprocessor algorithm, or he can use a *global* scheduler. Such a global scheduler assumes that each task can execute on any processor and can migrate from one processor to another. In this work we will devise new strategies for improving the global scheduling analysis for virtualized or hierarchical real-time systems, strategies that will significantly increase the number of detected schedulable tasksets.

1.2. Model-based Analysis of Real-Time Systems

Lately, model-driven engineering (MDE) has emerged as a major research area in software engineering as it promises to move the focus of the development process from writing source code to a more abstract world where the central point is the model of the system. From the model, the source code can be obtained to a chain of model transformations. Using MDE for real-time systems is still at the beginnings. One issue to be solved is how schedulability analysis can be encompassed within the models. Several solutions that bring closer these two domains have been proposed [106, 51, 149]. An analysis of these solutions showed that they are based either on classical scheduling theory, either use formalisms like timed automata [13]. In this work, we will also concentrate on the use of timed automata as a formalism for analyzing real-time schedulability.

Furthermore, multiprocessor/multi-core scheduling is the main focus in classical scheduling theory but until now it provided mostly results which lead to poor usage of the processor. In this context the use of formal methods for real-time scheduling is appealing. Moreover, verification of timing guarantees in hierarchical systems has been hardly studied.

1.3. Thesis Objectives

In order to be able to give timing guarantees in modern systems where real-time applications coexist with other performance demanding applications and, at the same time exploit efficiently the multi-core processor, accurate schedulability tests must be developed. Therefore, a rigorous study and analysis of the real-time scheduling theory for multiprocessor systems is required. Furthermore, as we aim to apply formal methods to solve the multi-core schedulability problem, we analyze the challenges that must be overcome and then look at how the problem has been tackled in related work.

One of the main objectives of the thesis is to propose a methodology for mod-

el-based real-time schedulability analysis. For this, we choose the timed automata formalism and transform the schedulability problem into a reachability one solved through model checking. The developed method addresses component-based systems, where the resource requirements of each component are specified in a service contract. The hierarchy of components in these systems is mapped to a scheduling hierarchy where each level is analyzed individually. The advantage of this approach is that schedulability can be analyzed incrementally, simplifying model checking. We give an exact schedulability analysis method for simple task models with periodic independent tasks, whereas in the classical theory only sufficient tests exist. Further, we also propose a method which provides a sufficient test for task models with precedence constraints, solving a problem that has not been tackled in classical scheduling theory.

Another objective of this thesis is to devise a new protocol for resource sharing in multi-core systems which will improve the state-of-the-art in the field. The protocol will address the problem of scheduling real-time tasks using a global preemptive fixed-priority algorithm with mutual exclusion constraints. At the same time, in order to determine if each task from an application meets its deadline, a schedulability test for the protocol will be provided.

Regarding resource sharing in hierarchical systems, another goal of this thesis is to propose a synchronization protocol that can be used in hierarchical multi-core systems, does not restrict application-level parallelism and can be applied when schedulers at all levels of the hierarchy use a global multiprocessor scheduling approach. Also, one of the objectives of this thesis will be to carry out a performance analysis of the proposed protocols.

1.4. Organization

The rest of this thesis is organized as follows.

Chapter 2. gives an overview of the multiprocessor real-time scheduling domain. The task models used mostly in classical real-time scheduling theory are presented and the main scheduling classes of scheduling algorithms are analyzed. Furthermore, we take a look at the anomalies that affect multiprocessors scheduling and the processor virtualization techniques engaged for ensuring temporal isolation between applications.

Chapter 3. is concerned with the challenges brought by the multi-core systems in the real-time schedulability analysis based on model checking and outlines how the solution proposed in this thesis improves on related work.

Chapter 4. describes the proposed scheduling analysis methods. First, a motivation of the usefulness of our methods is given. Further, detailed descriptions and analysis of the proposed methods are presented.

Chapter 5. proposes a method for generating multi-core time partitions in a two-level scheduling hierarchy. The chapter starts by giving the reasons why such a method is necessary. Further, the method is detailed and an empirical evaluation of it shows performance of the method.

Chapter 6. proposes a resource sharing protocol for periodic and sporadic task systems globally scheduled on a multiprocessor platform with fixed task priorities. The presented result improves over related works on the same topic. For the proposed protocol, we also give a schedulability test based on response time analysis. To prove the performance of the protocol we run a set of simulations, for various system configurations.

Chapter 7. handles the problem of resource sharing in virtualized or hierarchical real-time systems. We extend the protocol in Chapter 6. to cover the issues specific to hierarchical systems. In order to see if real-time constraints are met under the proposed protocol, we give a schedulability test for it and use this test to analyze the performance of the new protocol.

The final chapter concludes this thesis with a short summary, a list of the contributions of the thesis and a presentation of the problems that need to be tackled in the future.

2. SCHEDULABILITY ANALYSIS OF REAL-TIME MULTI-CORE SYSTEMS

Unlike other computing systems, real-time embedded systems must face timing and safety constraints, which means that the correctness of their behavior depends not only on performing well the operations they were designed for, but also on the time when those operations are performed. Examples of timed constrained system operations include processing an image in television between two successive frames or performing a control operation between successive rotations of an engine. Any error that may appear in the functioning of a flight control system for avionics or in computing the trajectory of an automobile is critical from a safety point of view.

However, real-time embedded systems include not only systems used in avionics or automobiles, but also devices like mobile phones or car navigation systems which also have features resembling more and more non-real-time systems. For example, on a mobile phone, along with real-time software, there may be various multimedia applications or the phone can have internet access which also posses security problems. Such security problems must not have any impact on the real-time components of the system. The solution to escape this dilemma comes from the use of virtualization techniques.

Guaranteeing correct real-time behavior and performance of a system is highly dependent on the efficiency of the scheduling algorithms used and the accuracy of the schedulability analysis methods backing up the algorithms. Multi-core processors closely resemble symmetric multiprocessors, therefore existing experience with multiprocessors real-time scheduling is valuable. The interest for multiprocessor real-time scheduling research became visible in the late 60', early 70'. In 1969, Liu [122] noted that

> "*few of the results obtained for single processors scheduling generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem*".

Almost ten years later, in 1978, the paper of Dhall and Liu [73] determined greatly the course of the research in this field for the next twenty years. They identified what is known in the literature as the "Dhall effect" (detailed in Section 2.1.3.) that appears in global multiprocessor scheduling. As a consequence research has focused mostly on partitioned scheduling approaches.

However, research in the multiprocessor scheduling field gained momentum only after the year 2000, when the fact that increasing processor performance only with higher clock speeds will soon become impossible and hence the need for increasing the number of processor cores as a solution for increasing processing speed.

This chapter covers the main issues related to multi-core/multiprocessor schedulability analysis. The first section defines the terminology and concepts used further in this thesis. A classification of multiprocessor algorithms is presented in Section 2.1. Next, Section 2.3. is devoted to presenting the main performance metrics used in classical schedulability analysis. As a set of scheduling anomalies have been identified for multiprocessor systems [22, 18, 113], we devote Section 2.2. to presenting these anomalies. The solutions proposed in classical real-time scheduling analysis for

enabling virtualization are discussed in Section 2.4. Section 2.5. analyzes current research results related to protocols for accessing mutually exclusive shared resource in multiprocessor real-time systems, as well as in virtualized real-time systems. The chapter ends with a set of concluding remarks on the topics included in it.

2.1. Concepts and Terminology

2.1.1. Task Models

The majority of the research in multiprocessor real-time scheduling assumes that the system has to deal with repeatedly executing processes modeled as tasks which generate jobs. Therefore a real-time task is the basic unit of work that must be executed by the system and each task is a sequence of one or more jobs, where each job is an executing process. The purpose of multiprocessor real-time scheduling is to plan the execution of a set of tasks belonging to a real-time application such that all tasks meet their timing constraints. The *taskset* is assumed to be static.

Each task has an associated *deadline* and the correct functioning of the system depends on whether all jobs of the task complete execution before their deadline. Also, each task is characterized by at least two other parameters - a *release time* and a worst case *execution requirement* (or *computation time*) - meaning that each job of the task must execute for at most an amount equal to task's execution requirement between its release time and its deadline. However, in the case of multiprocessor scheduling, a task executing for less than its worst case computation time can make the taskset unschedulable, and therefore an accurate task model must also consider the *best case computation time* for each task.

Most of the research in the multiprocessor real-time scheduling area is concentrated on two simple task models: the *periodic task model* and the *sporadic task model*.

The *periodic task model* was first defined by Liu and Layland in 1973 [123] and is still extensively used [19, 86]. In this model, each recurring process in the real-time system is considered a periodic task $\tau_i = (T_i, C_i, D_i)$, with the following meaning: *task* τ_i *with period* T_i *and worst case execution time* C_i *generates a job at each time instant* t *representing a multiple of* T_i *which needs at most* C_i *processing units and must complete in a period of time equal to* D_i. Therefore the real-time system is modeled as a collection of n periodic tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$.

Periodic tasksets can be *synchronous* if there is a time point when all tasks in the set are released simultaneously, or *asynchronous*, if task arrivals are always separated by a time offset.

If we lessen the requirement that the jobs of a task must start exactly at the beginning of the period and consider that T_i represents the minimum interval between two job releases, the model above becomes a *sporadic task model*. This model was formally described by Mok [135]. Each task is defined by three integers $\tau_i = (T_i, C_i, D_i)$ - the minimum separation between two consecutive jobs of τ_i, the worst case computation time and the deadline as a value relative to the job release time.

Further, in all cases for both task models it is assumed that a job may be released only at a time instant t such that a successive job release must not occur before $t + T_i$ time units.

Both models discussed above make the assumption that both tasks and jobs allow no parallelism. Allowing either type of parallelism means fulfilling the potential

of a multiprocessor or multi-core system. There are two levels at which parallelism would be possible [65]. First, in a system it could be allowed for multiple jobs of the same task to arrive simultaneously and each job could run on a different processor enabling thus *task parallelism*. Another possibility is constituted by *job parallelism*, where each job can be executed on several processors or cores at the same time.

Collete et al. [65] give a definition of a task model including job parallelism for sporadic tasks running on m identical processors. They consider a system of n sporadic tasks $\tau = (\tau_1, \tau_2, ..., \tau_n)$ where each task $\tau_i,\ 1 \leq i \leq n$ is defined by a period T_i, a worst-case execution deadline C_i and a m-tuple of real numbers $\Gamma = (\gamma_{i,1}, \gamma_{i,2}, ..., \gamma_{i,m})$ with the interpretation that if task τ_i executes for t time units on j processors it completes $\gamma_{i,j} \times t$ units of execution. All tasks have an implicit relative deadline equal with the task period. Their model also includes the limitation defined by Amdahl's law which states that the maximum degree of parallelism in any application is limited by the size of its serial components. Consequently, executing a job on k processors will not make the job run k/j times faster as executing it on j processors and also the number of processors on which a job may be executed in parallel is upper bounded. Once the bound is reached, using more processors leads to job performance degradation.

The deadline constraints on the tasks have been divided in three coarse categories:

- *implicit deadline*: tasks have deadlines equal to their periods ($D_i = T_i$),
- *constrained deadline*: tasks have deadlines smaller or equal to their periods ($D_i \leq T_i$),
- *arbitrary deadline*: there is no order relation enforced between task deadline and period.

From the task parameters defined above, several others were derived, all used extensively in the analysis of multiprocessor real-time scheduling:

- U_i - processor utilization factor, i.e. how much time spends the processor executing task τ_i:

$$U_i \stackrel{def}{=} \frac{C_i}{T_i},$$

- $load(\tau_i)$ - the computational demand of task τ_i:

$$load(\tau_i) = \frac{C_i}{D_i}.$$

- *nominal laxity (slack time)*: the maximum delay allowed before starting task τ_i assuming only this task is using the processor

$$l_i = D_i - C_i$$

- *residual laxity*: the maximum delay allowed for resuming the execution of task τ_i assuming only this task is using the processor

$$l_i(t) = (D_i - t) - C_i(t),\ 0 \leq C_i(t) \leq C_i$$

For the multiprocessor real-time system model to be complete we need a model of the multiprocessor platform accompanying the model of the tasks executing in the system. Scheduling theory distinguishes between three different kinds of multiprocessor machines [86]:

Identical parallel machines All processors are considered to have the same computing capacity. This is the most used model in the research of multiprocessor scheduling [141, 82, 65] and includes also multi-core systems.

Uniform parallel machines In this model, each processor has its own computing capacity s (i.e. if the computing capacities of all processors are equal this becomes the previous model). A job i with computation request p_i running on a processor with computing capacity s_j will complete in $p_{ij} = p_i/s_j$ units. This model can be considered more realistic since it is possible for some processors to reserve a certain capacity for executing non real-time tasks.

Unrelated parallel machines In this model the computing capacity of each processor depends on the job using it. Thus, a job i will need $p_{ij} = p_i/s_{ij}$ units to complete when executing on processor j.

The relation between these types of machines is depicted in Figure 2.1.

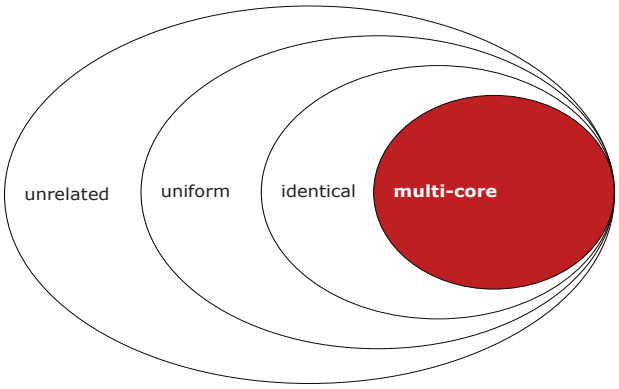


Figure 2.1: Multiprocessor machines taxonomy

2.1.2. Feasibility and Schedulability

A taskset $\tau = (\tau_1, .., \tau_n)$ is *feasible* with respect to a multiprocessor platform π if there exists at least one scheduling algorithm which can schedule all possible job sequences generated by the tasks in the system and no job misses its deadline. A task is *schedulable* if all its jobs complete their execution before their deadlines. A taskset is *schedulable* under a given scheduling algorithm if the algorithm provides a feasible schedule for the taskset. In other words, a taskset is schedulable if all its tasks are schedulable.

A scheduling algorithm is *optimal* with respect to a multiprocessor platform π and a specific task model M if it can schedule all tasksets which comply with the task model M and are feasible on the platform π.

A scheduling algorithm is *work conserving* if the algorithm never idles a processor while there is some job ready for execution and which may legally execute on that processor.

A *schedulability test* checks whether a given scheduling algorithm can provide a feasible schedule for a specific taskset. A schedulability test is said to be *sufficient* with respect to a scheduling algorithm and a system if all tasksets deemed schedulable by the test are indeed schedulable. A scheduling test is termed as *necessary* if all

tasksets which fail the schedulability test are indeed unschedulable. A schedulability test is *exact* if it is sufficient as well as necessary.

For a given taskset $\tau = (\tau_1, .., \tau_n)$ and multiprocessor platform π we can define the *feasibility interval* as *a finite interval such that if all tasks in the taskset meet their deadline in this interval then they will always meet their deadline* (no deadline miss will ever occur). The existence of a feasibility interval for a given task and system model is crucial for schedulability analysis since it simplifies it by allowing to perform the schedulability test only during that interval and being able to provide a valid verdict on the taskset schedulability.

2.1.3. Taxonomy of Multiprocessor Scheduling Algorithms

A multiprocessor scheduling algorithm determines which jobs should execute at the current time and on what processor. Therefore the scheduling algorithm must solve two problems:

- the *priority assignment* problem: determine an execution ordering of the jobs of the tasks, and
- the *processor allocation* problem: determine the processor on which a job must execute.

For the first problem several scheduling approaches are defined based on whether the schedulability of a system is performed statically or dynamically or whether it results in a plan according to which tasks are dispatched at run-time [144].

The most simple way to schedule tasks in a real-time system is through a static table with explicit start time and execution place for each task. This approach is known as *static table-driven scheduling* and although the resulting schedule is predictable it presents the drawbacks that it is inflexible since any change in the parameters of the tasks determines the reconstruction of the whole scheduling table and that no job preemption is allowed. The technique is applicable to periodic tasks for which the start and completion times can be identified using heuristic algorithms.

Current scheduling algorithms assign priorities to tasks and choose the task with the highest priority for dispatching next assuming that priorities were assigned such that the resulting schedule meets all deadlines. Depending on whether the priority of the jobs of a task can change from one job to another, we can distinguish fixed and dynamic priority algorithms. With *fixed priority-driven algorithms* a unique priority is assigned to each task and all jobs of the task will execute with that priority. Static priority assignment is appealing as once the task priority is assigned we do not have to reevaluate it. An example of fixed priority-driven algorithm is the Rate Monotonic (RM) scheduling [123].

When task priorities are assigned to individual jobs of a task we discuss about *dynamic priority scheduling* [150]. We can distinguish between two kinds of dynamic priority scheduling [58]: *fixed within a job*, or *fully dynamic*. With priorities fixed within a job, for every pair of tasks τ_i and τ_j, if at some moment t job J_i of task τ_i has higher priority than job J_j of task τ_j than J_i will always have higher priority than J_j, but it is possible that for some other pair of jobs J_i' of task τ_i and J_j' of task τ_j, J_j' to have higher priority than J_i'. An algorithm that falls in this category is the Earliest Deadline First (EDF) scheduling [123]. In the fully dynamic approach, the relative priorities of two jobs can change over time: if at moment t for a job pair (J_i, J_j) corresponding to the task pair (τ_i, τ_j), J_i has higher priority than J_j, it is possible that at moment t' job J_j will have higher priority than J_i. The Least Laxity First (LLF) [135] algorithm can be included in this class.

Table 2.1: A parallel between advantages and disadvantages of partitioned and global approaches for multiprocessor real-time scheduling

	Partitioned	**Global**
Advantages	• a missed deadline affects only tasks allocated to the same processor • no penalties due to task migration (lower context switch costs, lower cache miss costs) • any of the well-known uniprocessor scheduling algorithms can be used for tasks allocated to a processor	• better resource utilization as most of the times tasks execute for less than their worst case execution time • it is possible to have fewer context switches / preemptions than in the partitioned case [21] • more appropriate for open systems when tasks can come at any time
Disadvantages	• allocation of tasks to processors is a NP-hard problem • if tasks can enter and leave the system dynamically, it is necessary to reallocate tasks to processors	• optimal scheduling algorithms exist only for periodic task models • maintaining a single lengthy global queue available to all processors may be cumbersome

If we refer to the way the processor allocation problem is solved, scheduling algorithms can also be classified as *partitioned* or *global*. In partitioned scheduling, each processor is scheduled independently and each task is assigned to one processor and it is allowed to execute only on that processor. In global scheduling, tasks compete for all processors. Global scheduling can be further classified according to when migration is allowed [58]:

- task level migration: different jobs of a task can execute on any of the available processors but once a job has started on a processor it can execute only on that processor,
- job level migration: a job can migrate from one processor to another, with the limitation that the job is not permitted to run in parallel.

For many years, the global scheduling strategies have received little attention, mainly because it was believed that they suffer from the "Dhall effect" [73], a scheduling paradox which makes that tasksets with low processor utilization are unschedulable on a multiprocessor platform. More specifically, in [73] Dhall and Liu showed that for a set of $m+1$ periodic tasks scheduled on m processors using global EDF the utilization bound for schedulable tasksets is just $1+\epsilon$ for an arbitrary small ϵ. This "Dhall effect" led to the conclusion that global multiprocessor scheduling is inferior to partitioned multiprocessor scheduling. Moreover, it has been proved that no global online scheduling algorithm is optimal for scheduling tasks with distinct deadlines on multiprocessor systems [100]. Only in late 1990' and early 2000 it was shown that algorithms like EDF or LLF which provide pessimistic evaluations of the schedulability tests, can be improved if "resource augmentation" is used [141, 86] (i.e. the processing speed of available resources is increased). The effect of this is twofold. First it was proved that the "Dhall effect" has impact only on tasksets with certain particularities which may never appear in real systems (e.g. tasksets with m

low utilization tasks and one task with utilization close to 1). Secondly, it changed the course of multiprocessor schedulability analysis and the majority of the research focused on providing taskset utilization bounds dependent on the maximum task utilization [19, 45, 25, 26, 82, 28]. Today, with the advent of multi-core processors, the interest in the global scheduling algorithms has increased considerably [16, 17, 55]. Table 2.1 draws a parallel between the advantages and disadvantages of the partitioned and global multiprocessor real-time scheduling algorithms. Global scheduling can be difficult to apply on any multiprocessor architecture due to migration costs but experiments have shown that on some architectures these costs can be smaller than preemption costs [35], which affect both global and partitioned scheduling. On the other hand, modern real-time systems are open systems, where tasks are created and destroyed dynamically, and global scheduling is the best approach to follow in such cases.

Recently, in order to cope with the disadvantages of both partitioned and global algorithms, namely to achieve better resource usage with fewer migrations, a third class of algorithms appeared: the *hybrid* ones. In the hybrid approach it is possible either to split a task and partition its jobs between processors [20] or to allocate some tasks similar to the partitioned approach, while a subset of the tasks are scheduled in a global fashion [105, 111, 132]. Another mixed approach, which aims to alleviate the drawbacks of the partitioned and global approaches, is *clustered* scheduling [56]. In this case, the multiprocessor platform is partitioned into clusters of processors. The taskset is statically divided in task subsets, with each subset statically allocated to a cluster. Within each cluster, the tasks are scheduled globally. This approach is especially suited for multi-core systems were cores can be clustered around the different levels of shared caches. A study [36] performed recently on an implementation of partitioned, global and clustered EDF in a real-time Linux extension called $LITMUS^{RT}$ [57, 92], showed that the clustered approach performs best for soft real-time tasks for any platform size and is suitable for scheduling hard real-time tasks on small and medium-sized platforms.

Figure 2.2 sums up the possible relationships between the presented multiprocessors scheduling taxonomies. The work in this thesis focuses on global scheduling with job level migration and addresses both dynamic and fixed-priority tasksets (the grayed boxes in Figure 2.2).

An analysis [58] of the nine classes of global and partitioned algorithms based on the tasksets that can be scheduled by each class, has shown that global dynamic priority scheduling with job-level migration dominates all other classes. This means that any taskset that is deemed schedulable by an algorithm in any of the other classes is also schedulable by algorithms in this class, but there are also tasksets deemed schedulable by algorithms in this class but which are not schedulable according to other algorithms. Algorithms in the fixed-priority class are incomparable, meaning that tasksets schedulable according to some algorithm in one of these classes may be unschedulable according to some other fixed-priority algorithm. Also, the three partitioned classes are incomparable.

Moreover, while for partitioned algorithms, online optimal scheduling algorithms exist for both periodic and sporadic tasksets, the class of global algorithms lacks such online optimal algorithms. The impossibility of finding an optimal online scheduling algorithm for periodic tasksets with multiple deadlines has been proven first by Hong and Leung in 1992 [100]. This result was recently generalized for sporadic task sets [85]. Only for periodic tasks optimal global dynamic scheduling algorithms are known: Pfair [29], LLREF [62] or Deadline Partitioning Fair (DP-FAIR) [118]. Although these algorithms offer a processor utilization bound of 100%, they are not priority-

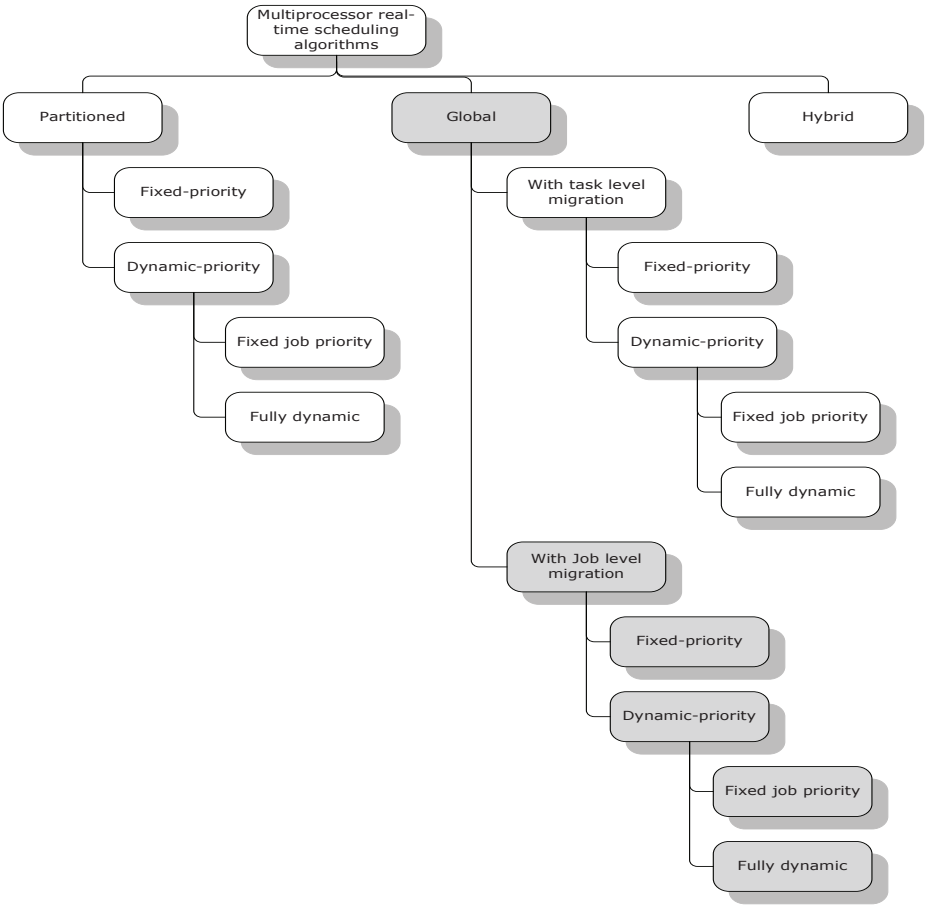


Figure 2.2: Multiprocessor real-time scheduling algorithms taxonomy

based and incur significant runtime overhead. Moreover, both Pfair and LLREF are not work-conserving algorithms.

Table 2.2 gives an overview of existing global and partitioned schedulability tests for sporadic tasksets scheduled on multiprocessor platforms. The schedulability tests are referred by the corresponding priority assignment policies. It must be noted that, because partitioned multiprocessor scheduling can be reduced to uniprocessor

Table 2.2: The main schedulability tests for different classes of multiprocessor real-time scheduling considering only sporadic tasksets

	Partitioned	**Global**
Fixed priority	RM [123] (exact) DM [123] (exact)	RM-US[ζ] [19] (sufficient) TkC [21] (sufficient) DM [123] (sufficient)
Fully dynamic	EDF (exact)	EDF [28, 31, 27], EDF-US[ζ] [155], EDF-DS[ζ] [43], EDZL [115] (sufficient)

scheduling, there exist exact tests for this class of algorithms. However, for global scheduling only *sufficient* schedulability tests are known until now.

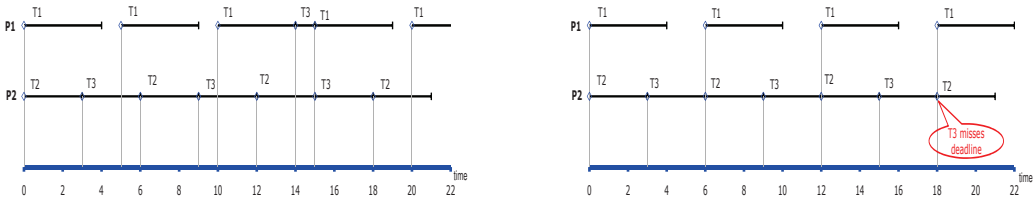
2.2. Multiprocessor Scheduling Anomalies

It is very important to stress that for multiprocessor scheduling, unlike for the uniprocessor one, apparently positive changes in the parameters of the taskset to be scheduled can transform a schedulable taskset into an unschedulable one. For example, one would expect that an increase in the period of a task leading to a lower processor utilization for that task, would improve schedulability, while in fact it can make it unschedulable. This kind of phenomena are called *scheduling anomalies*. Based on the "positive change" that can occur on the taskset, several anomalies have been identified [18, 22].

Period anomalies: The positive effect of increasing the period of a task is the decrease of the processor utilization for the task in cause. However, on the negative side, this change will modify the arrival times of the task. This leads to a different distribution over time of processor load and can determine a missed deadline of a lower priority task or of the same task.

Anomaly 1. *[22] For any fixed-priority preemptive global multiprocessor scheduling, there exist schedulable tasksets such that if the period of a task τ_i increases, either a task τ_j with lower priority (see Example 1) or the same task τ_i will be unschedulable (see Example 2).*

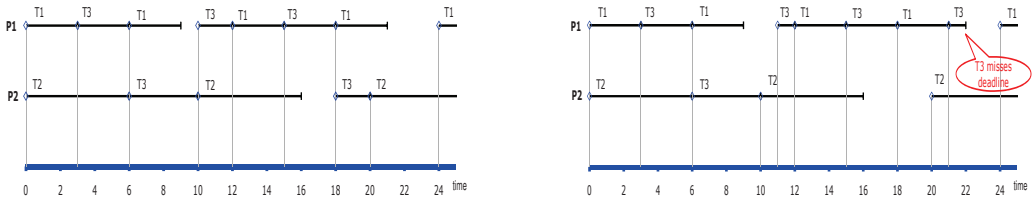
Example 1. *Consider the following taskset $\tau = (\tau_1, \tau_2, \tau_3)$ scheduled using RM: $\tau_1 = (T_1 = 5, C_1 = 4, D_1 = 5)$, $\tau_2 = (T_2 = 6, C_2 = 3, D_2 = 6)$ and $\tau_3 = (T_3 = 18, C_3 = 10, D_3 = 18)$. If we increase T_1 with one unit the taskset becomes unschedulable (see Figure 2.3).*



(a) Task set schedulable (b) Task set unschedulable

Figure 2.3: Period anomaly in global fixed-priority scheduling: increasing the period of τ_1 with 1 makes τ_3 unschedulable.

Example 2. *Consider the following taskset $\tau = (\tau_1, \tau_2, \tau_3)$ scheduled using RM: $\tau_1 = (T_1 = 6, C_1 = 3, D_1 = 6)$, $\tau_2 = (T_2 = 10, C_2 = 6, D_2 = 10)$ and $\tau_3 = (T_3 = 10, C_3 = 7, D_3 = 10)$. If we increase T_3 with one unit task τ_3 becomes unschedulable (see Figure 2.4).*



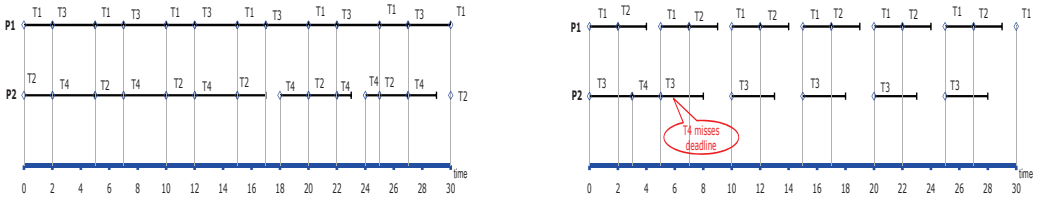
(a) Task set schedulable (b) Task set unschedulable

Figure 2.4: Period anomaly in global fixed-priority scheduling: increasing the period of τ_3 with 1 makes τ_3 unschedulable.

Priority ordering anomalies: A low priority task τ_i can be delayed only by higher priority tasks. One would expect that this interference of higher priority tasks is the same no matter what the priority ordering of these tasks is. This assumption is true in the uniprocessor scenarios. However, in multiprocessor scheduling, different priority orderings give different schedules for the lower priority task.

Anomaly 2. *[22] For any fixed-priority preemptive global multiprocessor scheduling, there exist tasksets for which the response time of a task depends not only on the periods and execution requirements of its higher priority tasks, but also on the priority ordering of those tasks (see Example 3).*

Example 3. *Consider the following taskset* $\tau = (\tau_1, \tau_2, \tau_3, \tau_4)$ *scheduled using RM:* $\tau_1 = (T_1 = 5, C_1 = 2, D_1 = 5)$, $\tau_2 = (T_2 = 5, C_2 = 2, D_2 = 5)$, $\tau_3 = (T_3 = 5, C_3 = 3, D_3 = 5)$ *and* $\tau_4 = (T_4 = 6, C_4 = 3, D_4 = 6)$. *If we consider the priority ordering is* $\tau_1, \tau_2, \tau_3, \tau_4$ *with* τ_1 *having the highest priority, the taskset is schedulable (see Figure 2.5(a)). If we switch the priorities of tasks* τ_2 *and* τ_3*, task* τ_4 *becomes unschedulable (see Figure 2.5(b)).*



(a) Task set schedulable (b) Task set unschedulable

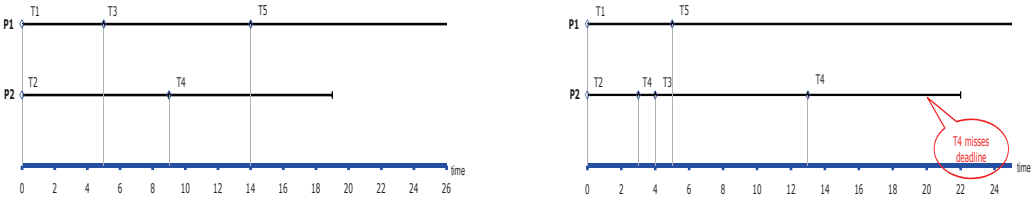
Figure 2.5: Priority anomaly in global fixed-priority scheduling: switching between the priorities of τ_2 and τ_3 makes τ_4 unschedulable.

Execution time anomalies: The positive effect of decreasing the execution time of a task is the decrease of the processor utilization for the task in cause. However, on the negative side, this change may modify the allocation of tasks to processors. This leads to a different distribution over time of processor load and can determine a missed deadline of a lower priority task.

Anomaly 3. *[91] If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, then increasing the number of processors, or reducing computation times can increase the schedule length (see Example 4).*

Example 4. *Consider the following taskset* $\tau = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5)$ *scheduled using DM with parameters given in Table 2.3. If we consider the priority ordering is* $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$ *with* τ_2 *executing for 9 time units the task set is schedulable (see Figure 2.6(a)). If we decrease the execution time of task* τ_2 *to 3 time units then the taskset becomes unschedulable (see Figure 2.6(b)). We assume task preemption is possible but migration is not allowed.*

Absence of critical instance: In uniprocessor scheduling, the *critical instant* is when a task arrives simultaneously with its higher priority tasks. This instant represents the worst case execution scenario for uniprocessor real-time scheduling. However, for the multiprocessor case no worst case job arrival pattern could have been identified [113, 26]. The critical instant effect can be seen in Figure 2.4(b) where, on the first invocation all tasks meet their deadlines, but in the second one, task τ_3 misses its deadline.



(a) Task set schedulable

(b) Task set unschedulable

Figure 2.6: Execution time anomaly: decreasing the execution time of τ_2 makes τ_4 unschedulable.

Table 2.3: Taskset to highlight the execution time multiprocessor scheduling anomaly

Task	Release time	Computation time	Deadline
τ_1	0	5	6
τ_2	0	[3,9]	15
τ_3	4	9	18
τ_4	0	10	20
τ_5	5	50	100

2.3. Performance Metrics

In order to evaluate the effectiveness of a scheduling algorithm or a schedulability analysis method, some performance metrics are needed. Two categories of such metrics can be identified: *theoretical* and *empirical* measures.

Theoretical metrics

M1. Utilization bounds In 1974, Horn [101] observed that $U(\tau) \leq m$ ($U(\tau)$ is the worst case utilization of a taskset) is a sufficient and necessary condition for the feasibility of implicit-deadline periodic tasksets. Horn's condition implies that if the condition is met, then there exists *some* scheduling algorithm which can plan the taskset. For a specific algorithm A the bound U_A on the taskset utilization is usually lower:

$$U(\tau) \leq U_A \tag{2.1}$$

This condition says that all tasksets with utilization lower or equal to U_A are schedulable with algorithm A. The condition is only sufficient, not also necessary, which means that there may be tasksets deemed unschedulable by A which are in fact schedulable [19, 25].

M2. Approximation ratio Another way to prove the efficiency of an algorithm is to compare it with an optimal algorithm. This comparison can be made from several points of view. For example, one could estimate the efficiency of an algorithm based on the number of processors it uses compared to the number used in the optimal algorithm [73]. If m_o is the number of processors used by the optimal algorithm when scheduling a taskset τ and m_A is the number of processors used by algorithm A, then the approximation ratio is defined as:

$$\mathcal{R}_A = \lim_{m_o \to \infty} (\max_{\forall \tau} (\frac{m_a}{m_o})) \tag{2.2}$$

Note that the smaller the value $\mathcal{R}_A \geq 1$, the better is the algorithm.

M3. Resource augmentation Instead of comparing an algorithm A with the optimal

algorithm based on the number of processors required, this metric considers the increase of processor speed that would be necessary (assuming task execution times decrease linearly with the increase of processor speed) to make a taskset schedulable with algorithm A [30]. The resource augmentation r_A of an algorithm A is defined as the maximum factor by which the speed of m processors must be increased such that all taskset are schedulable with A:

$$r_A = \max_{\forall m,\, \forall \tau} (r_A(\tau)) \tag{2.3}$$

Note that a smaller $r_A \geq 1$ means a better algorithm.

The theoretical metrics presented above are based on some pathological classes of tasksets, which may never occur in some applications. Conversely, empirical measures can help understanding the performances of an algorithm in real-life scenarios.

Empirical measures Different scheduling algorithms can be evaluated experimentally by comparing the number of randomly generated tasksets that each of them deems schedulable. Ideally, this number would have to be compared with the number of tasksets schedulable by an optimal algorithm but for many types of task models and classes of algorithms, such an optimal algorithm is not known yet. Through experimentation, the effects of different system parameters on the scheduling efficiency can be determined easily. Such parameters include number of tasks, number of processors, taskset utilization, various ranges for task parameters.

2.4. Virtualization Techniques in Real-Time Scheduling

Currently, real-time research puts considerable efforts on designing and implementing *open real-time environments* [70]. Such systems allow building systems from independently designed, implemented and validated applications, which must execute concurrently on a shared platform. A key requirement for these open systems is that any application that meets its timing constraints when running in isolation, must also meet its timing constraints when running on the shared platform. This requirement is called *temporal isolation*.

Applications in a real-time system can consist of tasks with a broad range of timing requirements. Typically these tasks can be classified as:

- *hard* real-time tasks: it is considered a fatal error if the task does not complete before its deadline,
- *soft* real-time tasks: the task should complete before its deadline but occasionally missing it is not critical,
- *non-real-time* tasks: tasks do not have a deadline associated with them.

Temporal isolation ensures that all three types of tasks can coexist on a single system without jeopardizing each other.

The typical approach for providing such temporal isolation is the *resource reservation* paradigm introduced in [133]. Under this paradigm, the processing capacity provided by the multiprocessor or multi-core platform is divided into a set of resource partitions, effectively forming a smaller number of *virtual processors* to which application tasks are allocated.

The advantages of using a design approach based on virtual processors are twofold. First, any violation of a timing constraint in an application executing on the shared platform will not affect any other application on the platform. Second, decisions of whether the system can guarantee the correct behavior of an application

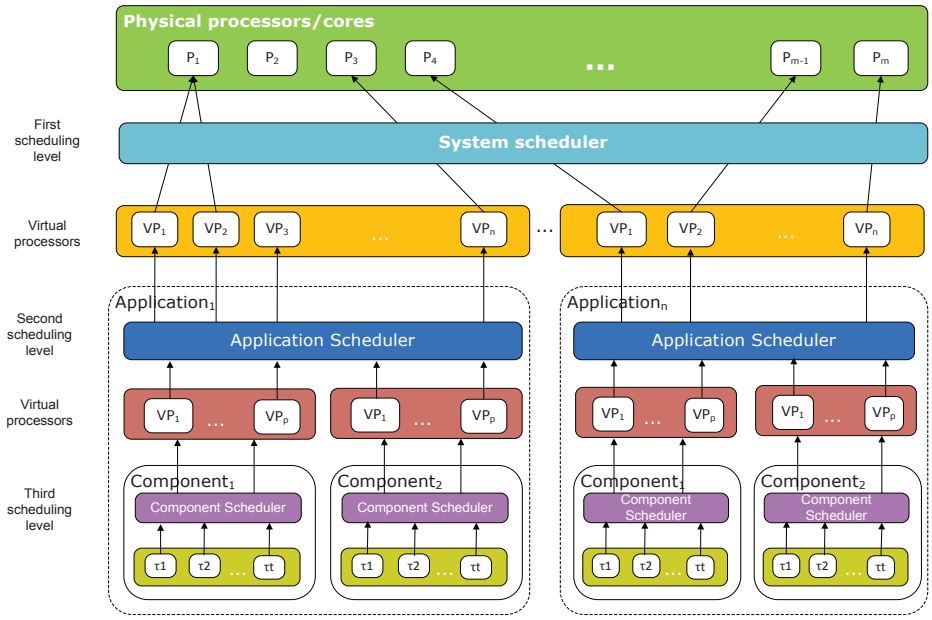


Figure 2.7: A three-level scheduling hierarchy

can be taken based only on the requirements of the application and on the allocated virtual processors, independently of the other applications.

Usage of resource reservation techniques is appropriate for systems composed of applications with hard-, soft- and non-real-time constraints [52], as through temporal isolation, one can be sure that hard deadlines will not be endangered by any of the other kinds of tasks.

Real-time scheduling supports resource reservation schemes through the use of hierarchical schedulers with two or more levels. On the top level, a scheduler allocates resources to each application on the shared platform and, on another level, each application has a local scheduler responsible for scheduling the tasks of the application on the virtual processors. Furthermore, as another trend in real-time embedded systems is building applications from independently designed and implemented components, for each of the components there could be a distinct scheduler for the component's tasks, introducing several supplementary scheduling levels. Such a scheduling hierarchy with three levels is depicted in Figure 2.7.

2.4.1. Virtual Resource Models

A resource partition is typically modeled as a pair (Q_i, P_i), meaning that the resource partition will provide Q_i units of execution every P_i units of time, otherwise said, the virtual processors have a bandwidth $U_i = Q_i/P_i$. Bandwidth reservation strategies may be accomplished through the use of two categories of resource models: *time partitions* and *bandwidth or execution time servers*.

Time Partitions

In a uniprocessor system, a time partition is implemented as a fixed-length major time frame composed of several scheduling windows. A scheduling window is defined by its offset to the beginning of the major time frame and by its length. The scheduling scheme of the major time frame repeats during the execution of the system such that all scheduling windows are essentially periodic (see Figure 2.8). The application associated with a time partition gains access to the processor whenever one of the scheduling windows becomes active and is preempted when the window terminates [104].

A time partition can also contain scheduling window sets, where the offset of the set is the offset of the first scheduling window in the set and the period of the windows in the set is a divisor of the length of the major time frame.

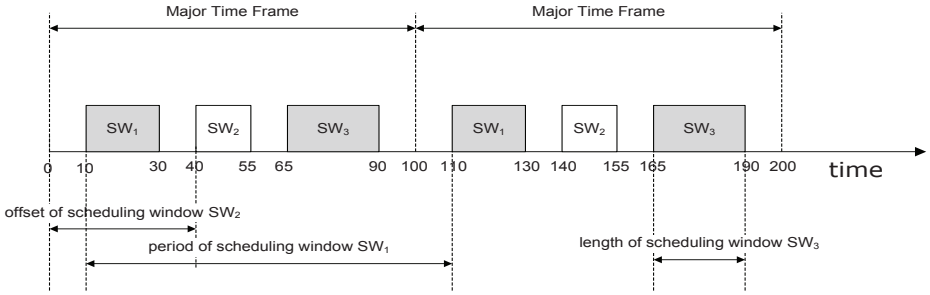


Figure 2.8: Uniprocessor time partitions: partition A contains the gray scheduling windows SW_1 and SW_3 and its major major time is 100 and partition B contains the white scheduling window SW_2 and its major time frame is also 100.

In a multiprocessor system, we can define a time partition assuming there is a major time frame for each processor, but frames on all processors will have equal length and will be synchronized. The scheduling windows of frames on different processors can have different parameters and the time partition can contain scheduling windows on any processor [60]. Such a time partitioning scheme is used in PikeOS [104], a proprietary operating system. Figure 2.9 illustrates an example of two multiprocessor time partitions, both having the length of the major time frame equal to 100.

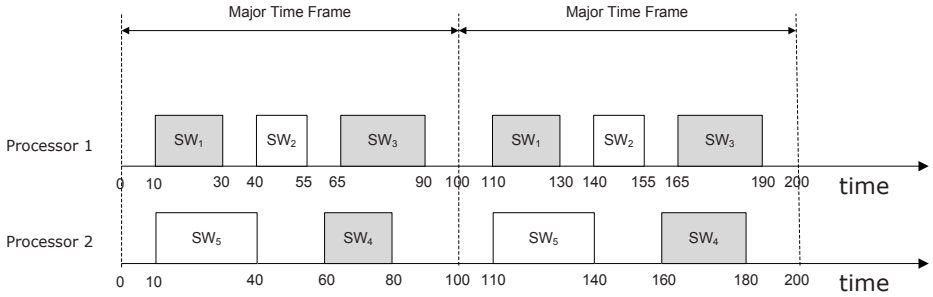


Figure 2.9: Multiprocessor time partitions: partition A contains the gray scheduling windows SW_1, SW_3 and SW_4 and its major major time is 100 and partition B contains the white scheduling windows SW_2 and SW_5 and its major time frame is also 100.

One of the disadvantages of time partitioning is its inefficient resource usage. This comes as a consequence of the fact that the time partitions are defined off-line, based on worst case application analysis, and its configuration cannot be changed at run-time. They are appropriate for use mostly in a hard real-time context as they guarantee that deadlines are meet even in worst case conditions. Another drawback, deriving from their static structure, is that they can only be scheduled using a non-preemptive scheduler.

Bandwidth servers

In real-time scheduling theory, a *bandwidth server* or *execution time server* is defined as an abstract entity used by a scheduler to reserve a fraction of processor time to a particular task or set of tasks. Each server S_k is characterized by a pair of parameters (P_k, Q_k) where P_k is the period of the reservation and by Q_k we denote the reserved execution time per period. For each server we define the utilization factor $U_k = Q_k/P_k$ representing the fraction of CPU-time reserved by server σ_k.

Initially, servers were used for scheduling soft- and non-real-time tasks along with tasks having hard deadlines, but they can also be used to provide temporal isolation in hierarchical systems as each application can be encapsulated in one or several servers. When the server is selected for execution, the application scheduler will choose one of the tasks in the application and that task is executed. If the application consists of several components, each with its own scheduler, then the task selected for execution by the application scheduler may actually represent another server.

While there are many design solutions for servers in uniprocessor systems, only a few results deal with the multiprocessor/multi-core case. The higher schedulability bound of multiprocessor dynamic scheduling algorithms influenced also the course of the research in this area.

The Multiprocessor Constant Bandwidth Server (M-CBS) [32, 33] is an algorithm based on the principles of the Constant Bandwidth Server (CBS) for the uniprocessor case. In CBS the server is characterized by a pair of parameters - (Q_s, P_s) - where Q_s is the server's maximum budget and P_s is its period (the server will execute at most Q_s time units in every interval of length P_s). The server is also characterized by its current budget and a dynamic deadline. The CBS server was designed for serving aperiodic tasks along with hard real-time periodic tasks without jeopardizing their deadlines. Every aperiodic task is assigned to a server task with the constraint that at most one task can be assigned to a server task. The task will have associated a deadline equal to the server's current deadline. The server's budget varies at runtime. The initial budget is set to Q_s and each time an aperiodic task is served, the execution time of the task is decreased from the server's current budget. When the current budget reaches zero or becomes negative it is recharged to the maximum value Q_s and the deadline is postponed by P_s. Server tasks are scheduled along with other kinds of tasks using EDF scheduling. In M-CBS a real-time system of $\sigma_1, \sigma_2, ..., \sigma_n$ servers are scheduled on m processors using an EDF algorithm. Each server σ_s is characterized by its processor share U_s and its period P_s. The priority of each server σ_s is given by a variable deadline D_s. The algorithm also keeps track of how much of server σ_s's current bandwidth has been consumed. When the whole bandwidth has been consumed the deadline D_s is incremented by P_s changing also the priority of the server. M-CBS chooses for execution m servers that have jobs awaiting for execution and have higher priorities. Due to the deadline-postponing approach used by this kind of server, the EDF algorithm used together with M-CBS is work-conserving, as

no processor will be idle while there are jobs ready for execution. On the other side, the same approach brings also a drawback as the job assigned for execution may be delayed indefinitely many times.

Another bandwidth server solution for the multiprocessor case is presented in [34]. They extend one of the most efficient algorithms for scheduling aperiodic tasks along with hard real-time tasks on uniprocessors machines, namely the Total Bandwidth Server (TBS) and introduce the Multiprocessor Total Bandwidth Server (M-TBS) algorithm. For each aperiodic task the worst-case execution time and the deadline are considered known only at its arrival time. The M-TBS schedules the hard real-time tasks and the servers using the EDF algorithm and handles aperiodic jobs through a server. The set of aperiodic jobs $J_1, J_2, ...$ with $J_i = (A_i, C_i),\ i \geq 1$, where A_i represents the job arrival time and C_i is the execution requirement, is scheduled by the server in the order of jobs arrival. The algorithm assigns to each aperiodic job a deadline which considers the overall aperiodic server load, the job computation requirement and the deadline associated with the previously arrived aperiodic job. After that the job can be scheduled by the EDF scheduler responsible for scheduling the hard real-time jobs. Further, the algorithm is extended for the case when the aperiodic jobs present real-time constraints, i.e. response time. In this situation the server performs an admission control test and if it is not possible to guarantee the required response time, the job is not accepted. The main advantage of M-CBS is that it fully exploits all capacity available on the multiprocessor platform.

One of the most complex scheduling solutions for multi-core platforms is introduced in [52]. A feasible scheduling algorithm for modern multi-core systems must bear hard sporadic real-time tasks along with soft sporadic real-time tasks and best-effort aperiodic jobs and for all these must guarantee deadlines or minimum deadline tardiness or minimum response times respectively, all with maximum efficiency of processors utilization. Again, they refer to a server approach for solving these problems. Basically, tasks are partitioned among "server tasks" which schedule their constituent tasks and, at another level, server tasks are scheduled on available processors using a global EDF algorithm. With each processors in the system a server for hard real-time tasks is associated. These servers have higher priority than any other server in the system and dispatch jobs using an EDF algorithm. Secondly, for all soft real-time tasks a single migratory server is created and a number (equal to the number of processors) of migratory servers will handle best-effort jobs. Best-effort jobs are placed in a global queue and when a server dedicated to serving this type of jobs is scheduled it will service jobs from the queue until either the queue empties or the server exhausts its capacity. For a better use of processor, hard real-time jobs that finish earlier than expected donate the unused capacity to a global queue. This solution addresses only systems where the number of hard real-time tasks is small due to the need to statically assign them to servers and considers that all jobs are independent. A server-based approach has also been used in [158] and [157] for scheduling applications with different criticality levels on a modern smartphone.

Periodic Resource Model

Besides time partitions and bandwidth servers, several other periodic resource models have been considered in the context of uniprocessor systems (e.g. bounded-delay resource model [136] or the periodic resource model [153]). As shown in [77] simply replicating these resource models to multiprocessor systems is inflexible and leads to wasted processor utilization. Therefore, a new model [152, 77] was introduced, a Multiprocessor Periodic Resource (MPR) model, where the contribution of each pro-

cessor to an application resource supply is not the same. Their work addresses a multi-level containment hierarchy where components have hard real-time timing requirements and consist of sporadic tasks. The MPR model is represented as a triplet (Π, Θ, m') meaning that the resource provides Θ units of computational power on a multiprocessor platform with m' processor every Π time units. They also provide an optimal condition for component schedulability under the global EDF scheduling policy and the MPR model based on resource supply bound function and component workload. Even if the algorithm is optimal, it is quite pessimistic. Moreover they introduce a technique for transforming a MPR model into a set of periodic tasks which are scheduled by the parent component, but since all these tasks have the same period, driven by the smallest period in any application task, the obtained taskset will result in wasted computing capacity. Moreover, the procedure for computing Θ uses the highest possible workload such that all application tasks meet their deadlines and the system can get to this workload only when the maximum possible number of tasks are running in parallel. A more optimist algorithm and model could guarantee all timing constraints with a smaller Θ.

In [116], the problem of scheduling multi-level component hierarchies on multiprocessor resource partitions is also considered but is mostly restricted to soft real-time sporadic tasks and assumes that only a small number of tasks have hard timing constraints. They manage to reduce the resource usage by allowing minimum parallelism and using first the maximum utilized processors and only a small fraction of a less utilized processor. As the authors themselves show, it is possible for some hard deadline to be missed and, consequently, their scheme is more appropriate for soft real-time tasks.

The resource model proposed in [152, 77] is highly dependent on the multiprocessor platform, and, consequently any application designed using such a model of the underlying platform is tightly coupled with the modeled platform. A more general interface of the resource layer is given in [48]. The Multi-Supply Function introduced in [48] describes the exact amount of resources provided to the application. This model gives a solution for the pessimistic approach in [152, 77] and is applicable even when the physical multiprocessor platform is already allocated to other applications and, consequently, is not fully available to the currently analyzed application. However, the schedulability tests presented for this model give only sufficient conditions and are quite pessimistic. The pessimism originates from the fact that the possibility of task migration is not well covered by the model. This flaw was corrected by the Parallel Supply Function introduced in [47].

2.4.2. Contract-based Real-Time Scheduling

In the last years, real-time embedded software development has focused more and more on building flexible and extensible applications. Component-based software systems achieve these objectives by gluing individually designed, developed and tested software components, each component having different timing requirements. Therefore, when building such a component-based system one must ensure that components can coexist without jeopardizing each other's execution.

Starting from a hierarchical scheduling scheme, Harbour has introduced the concept of service contracts [95]. In Harbour's model, every application or application component may have a set of service contracts describing its minimum resource requirements. This contract is acting like an interface of the component or application. The goal of this interface specification is to abstract out and encapsulate the salient features of the components' resource requirements. The system uses this informa-

tion during admission control, to determine whether the component can be supported concurrently with other already admitted components; for admitted components, this information is also used by the open environment during run-time to make scheduling decisions. Therefore, these contracts are used in online or off-line negotiations to determine if the resource requirements can be guaranteed or not.

Research is undertaken also for extending the service contract model for component-based multiprocessor real-time systems. Chang et al. [60, 160] proposed a two-level resource contract model. First, each application has a contract specifying the resources to be reserved for its execution. This is called an *external contract*. Next, every component of the application has its own contract, called *internal contract*, describing the portion of the resources specified in the external contract that must be distributed to the component. Each component consists of one or more tasks which may require parallel execution. Internal contracts are mapped to abstract servers which are further divided in execution time sub-servers in order to support parallel execution of the components. In this case the sub-servers are handled as simple periodic tasks. On the other hand, external contracts are mapped to multiprocessor time partitions [104] (see Figure 2.10). As each application will be mapped to a separate time partition, a specific scheduling policy may be associated with it.

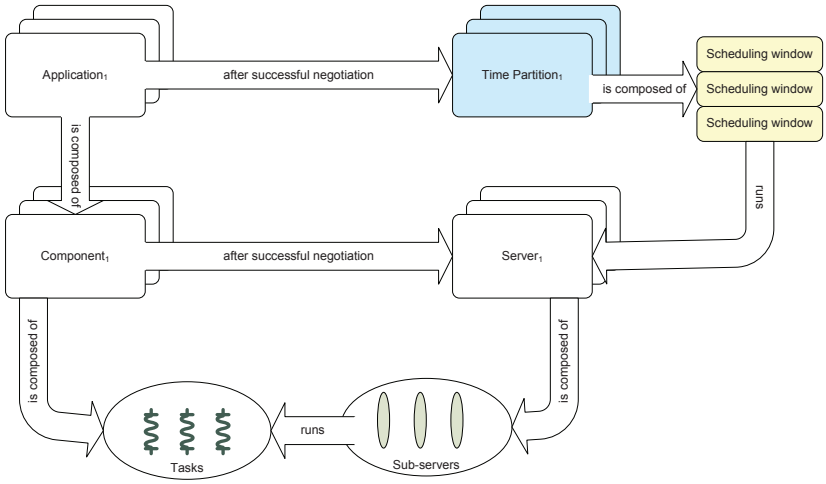


Figure 2.10: Contracts and hierarchical scheduling [60]

The schedulability analysis of the sub-servers and of the tasks in Chang's model, is based on a strategy introduced in [26], extended for usage in a hierarchical scheduling scheme and a fixed-priority algorithm. The outline of the strategy is as follows:

- consider a critical duration from the arrival to the deadline of a server, at the end of which a deadline is missed,
- compute the total workload of higher or equal priority servers during the critical duration,
- using this workload, define a *necessary* condition for the server to miss its deadline,
- derive an upper bound on the maximum interference of other servers on the analyzed server, such that the server meets its deadline,

- establish a *necessary* un-schedulability test as a inequality between the previously computed workload and maximum interference,
- negate the inequality and get a *sufficient* schedulability test.

2.5. Resource Sharing

Most of real-time scheduling research focused on applications consisting of independent tasks. However, real applications include tasks that share resources like shared data structures or I/O devices that have to be accessed in mutual exclusion, such that only a single task uses the resource at all times.

Static and dynamic priority-driven scheduling algorithms are susceptible to priority inversions due to blocking. When a high priority task blocks waiting for a resource held by a lower priority task, if the low priority task is repeatedly preempted by tasks with intermediate priority, it will not be able to complete and release the resource, causing the high priority task to remain blocked for a very long time.

For uniprocessor real-time systems, there is an extensive set of scheduling theories handling the priority inversion problem. The family of PIPs [151] handle priority inversion by temporarily changing task priorities: if at least one high priority task is blocked waiting for a low priority task to release a resource, the low priority task will execute with the highest priority of all tasks blocked by it. However, PIP does not prevent deadlock. The Priority Ceiling Protocol (PCP) [151] solves this problem by associating a priority ceiling with each shared resource. The priority ceiling is the highest priority of any task that may access the resource and behaves like a lock required for accessing the resource. PCP works similar to PIP but in order to prevent deadlocks, a task may access a resource only if its priority is greater than the highest priority ceiling of all resources locked by other active tasks in the system. This implies that any job of a task will block at most once. An alternative to PCP is the Stack Resource Policy (SRP), a protocol proposed by Baker et al. [24].

One of the first proposed resource sharing protocols for multiprocessor real-time systems was the Multiprocessor Priority Ceiling Protocol (M-PCP) [143], a variant of PCP for partitioned multiprocessor scheduling with fixed task priorities. M-PCP, as well as all the other protocols for partitioned multiprocessor scheduling, makes distinction between resources shared by tasks assigned to different processors (called global resources) and resources used only by tasks on a single processor (called local resources). For local resources, the uniprocessor PCP is applied, while for global resources the priority ceiling is computed based on task priorities on all processors (i.e. task priorities are assigned independently on each processor) and the resource is granted similar to PCP but using this priority ceiling. Later, Chen and Tripathi [61] extended M-PCP for systems with dynamic task priorities but the protocol can be applied only for periodic tasks and nested resource accesses are allowed only for resources of the same type (local or global). Gai et al. [87] introduced the Multiprocessor Stack Resource Policy (M-SRP) by extending the SRP for multiprocessors. As with M-PCP, in M-SRP tasks are partitioned between processors, but in M-SRP a blocked task spins while waiting for a resource request to be granted and this may lead to inefficient use of the processors. Moreover, globally shared resource accesses cannot be nested and their critical sections are executed non-preemptively.

One of the first contributions on resource sharing in real-time systems scheduled under preemptive global EDF belongs to Devi et al. and is described in [71]. However, they do not focus on finding a new protocol for task synchronization, but rather assume that lock requests are granted in FIFO order with critical sections ex-

ecuting non-preemptively and extend global EDF scheduling to account for non-preemptive critical sections. The approach is suitable only for short, non-nested resource accesses. Moreover, because tasks waiting for a resource to be granted, spin or implement waiting with retry-loops, the parallelism provided by a multiprocessor platform is not fully exploited.

The first protocol suitable for both global and partitioned scheduling approaches is Flexible Multiprocessor Locking Protocol (FMLP), proposed by Block et al. in [50]. FMLP uses EDF for assigning priorities to jobs, which also bounds the interference on high priority tasks from the lower priority ones. Here, deadlock is avoided by grouping resources for which nested requests can be issued. Resources are considered to be long or short based on how long they can be held. The resources are grouped based on the following principle: if there is a job that issues a request for resource R_1 nested within a request for resource R_2 then R_1 and R_2 are in the same group. Resources in a group are either all short or all long. When a job needs to access a resource it will lock the whole group of the resource. Further, short resource access is controlled using spin locks and critical sections are non-preemptive, while long shared resources are guarded by semaphores and granted according to PIP. The advantage of this synchronization scheme is that it allows unrestricted critical sections nesting. However, a schedulability test is not given for the protocol and this complicates the problem of finding out if a given set of tasks is schedulable under FMLP or not.

The first schedulability test for a global multiprocessor synchronization protocol is given by Easwaran and Andersson in [74]. They propose the Parallel Priority Ceiling Protocol (P-PCP) and develop schedulability tests for it and for global PIP. P-PCP uses a fixed-priority preemptive scheduling policy and uses a set of taskset-level configuration parameters α_i (one for each task) to control when a task is allowed to enter a critical section. Based on the values of the α_i parameters, one can either improve the efficiency of the processor usage or reduce the lower priority interference. However, lower priority interference is reduced only for the class of *reasonable priority assignments* (i.e. which assumes higher priority tasks have lower deadlines than lower priority tasks, similar to EDF or DM) and the protocol lacks a quantitative analysis which would have lent some insight into its efficiency.

Another approach for bounding the blocking time suffered by any high priority task is proposed by the Bounded Blocking with High Parallelism (BHP) [75] protocol. BHP is designed for nested resource accesses and controls tightly the execution of critical sections. Basically, whenever a task requests access to a resource and the resource is available, the request is granted only if during the time the resource will be in use, there is no possibility for a higher priority task to be prevented from executing due to priority inversions. Unfortunately, the time-complexity of BHP is rather large and this makes its implementation rather difficult.

2.5.1. Resource Sharing in Virtualized Systems

Resource sharing and task synchronization in virtualized or in Hierarchical Scheduling Frameworks (HSFs) has been studied extensively for uniprocessor platforms. Part of the existing results are extensions of the basic uniprocessor resource sharing protocols. Consequently, the SRP protocol has been extended for HSFs by Davis and Burns in [68] where the Hierarchical Stack Resource Policy (HSRP) is proposed. HSRP can be applied for applications scheduled using fixed-priority preemptive algorithms and allows mutually exclusive access to resources shared either locally within the same application, or globally, between different applications in the system. The problem of server budget depletion inside global critical sections is handled by adding extra

budget to the server until the critical sections end. To see if all tasks in an application meet their deadlines, Davis and Burns also give a schedulability test for HSRP, based on computing the worst case response time of each server and task in the system.

Another protocol based on SRP is Subsystem Integration and Resource Allocation Policy (SIRAP), proposed in [39], with an improved schedulability test in [38]. SIRAP is applicable for applications consisting of periodic tasks and, unlike HSRP which allows a server to overrun while the task inside accesses a global resource, SIRAP uses a skipping mechanism, preventing a task to enter a critical section unless enough server capacity is available for its execution.

Both HSRP and SIRAP require prior knowledge of critical sections lengths. This limitation is dropped in [121] where the Bandwidth Inheritance (BWI) protocol is introduced. However, BWI comes with another limitation as for each task a distinct server is used. Later, Bertogna et al. [43] proposed a new synchronization protocol for HSFs where it is not required to know a-priori the length of the critical sections but, instead this length is estimated at runtime. However, global critical sections are executed non-preemptively and only when there is enough server budget to complete them. The possibility to execute global critical sections with preemptions allowed and an accurate schedulability analysis are presented in [46].

The Bounded-delay Resource Open Environment (BROE) protocol [83] handles resource sharing in a hierarchical framework where each application is executed within a CBS [10]. BROE is suitable for open environments. Each application is admitted in the system based on its interface specifying its maximum processor demand translated in server speed, its maximum delay tolerance and the resource holding times for each global resource. Access to global resources is managed actually through SRP and, like SIRAP, BROE also uses a skipping mechanism along with application deadline postponing to prevent server depletion. A limitation of the protocol is that servers can be scheduled only using the EDF algorithm. Moreover, the protocol cannot guarantee hard deadlines.

Nemati et al. extended FMLP for hierarchical frameworks in [137, 139]. However, the first level of the framework consists just of virtual processors with the same capacity as the physical ones, the only scheduling problem at this level consisting only of mapping virtual processors to actual ones. Furthermore, resource sharing is handled only within a component and not within different components.

Nemati et al. also proposed another synchronization protocol for multiprocessor HSFs, namely Multiprocessor Hierarchical Synchronization Protocol (MHSP) [138]. The protocol is based on SRP and can be used with either global or partitioned scheduling. However, instead of handling separately each application, MHSP groups all tasks that are directly or indirectly dependent under a single component to which a single server is assigned. Hence, no actual resource sharing happens between different servers and resource sharing within each component can be handled using uniprocessor SRP.

The only known protocol which allows inter-application resource sharing in HSFs is the Multiprocessor Bandwidth Inheritance (M-BWI) protocol [78], an extension of the BWI protocol. M-BWI can be applied for both global and partitioned scheduling algorithms with either dynamic or static priorities. It allows co-existence of hard real-time, soft real-time and best-effort tasks, but for hard real-time tasks requires a-priori knowledge of task parameters. One of the limitations of M-BWI is that each task is assigned to a separate server. Furthermore, tasks waiting for a resource are waked up in FIFO order which may cause large delays to some tasks. This aspect makes the protocol less suitable for systems consisting of mostly hard real-time tasks.

2.6. Concluding Remarks

This chapter aimed at providing an introduction to some of the complex topics of multiprocessor real-time scheduling. The main task models analyzed in current research, namely the periodic and sporadic task models, were described and the most important concepts were introduced. Next, a taxonomy of the multiprocessor scheduling algorithms was presented. For each class in this taxonomy, an analysis of the current algorithms showed that for a very important class, the global algorithms, online optimal algorithms exist only for periodic tasks. Moreover, for sporadic tasks only sufficient schedulability tests are known. The following topic is that of the anomalies that emerge during scheduling of tasks on multiple processors. These anomalies burden the schedulability analysis for multiprocessors since it is very hard to identify a worst execution case and more task parameters must be considered for an accurate analysis.

Section 2.4. outlined the scheduling techniques for ensuring temporal isolation of multiple real-time applications running on a shared platform. Important resource models, like time partitions, execution time servers and some other current periodic resource models were briefly introduced. While time partitions are most suited for highly critical applications like avionics, the latest models try to mould on the application designer needs and to ease design by giving an abstraction of a multi-core platform. Section 2.4.2. introduced the concept of application contract as an interface specifying the resource requirements of the application. These contracts are used later in a sufficient schedulability test to see if the application can be executed on a resource partition represented by one of the presented resource models. It is important to note that, for all presented virtual resource platforms, most schedulability algorithms known in classical real-time multiprocessor theory are either only sufficient or are quite pessimistic. Because of these limitations, it is important to study the feasibility of other approaches for schedulability analysis than the ones presented in this chapter.

Lastly, Section 2.5. presented the protocols used for handling mutual exclusive access to shared resources. First, the protocols for single-level multiprocessor systems were analyzed. This analysis highlighted the imbalance between the amount of research undergone for developing protocols for partitioned versus global multiprocessor scheduling. Next, the issue of resource sharing was covered for hierarchical or virtualized systems. In this case, the analysis revealed that only one actual protocol exists for resource sharing in multiprocessor HSFs.

3. MODEL-BASED DESIGN OF REAL-TIME SYSTEMS

When building real-time systems it is vital to ensure that the timing constraints are satisfied by the system under construction. In this thesis we focus on the rigorous verification of timing requirements of multi-core real-time systems by using model checking and schedulability analysis. Examining the timing behavior of a system requires construction of a model of such behavior. The model is then checked for safety as we want to make sure that no undesirable state can be reached by the system. Second, the feasibility of the model is checked in order to assert that all deadlines are met.

As we have shown in Chapter 2., the analytical methods used for multiprocessor schedulability analysis provide only sufficient conditions, which lead to inherent pessimism. During the recent years, model checking emerged as an attractive approach to schedulability analysis providing absolute guarantees: if a system is deemed schedulable through model checking then is guaranteed that no deadlines will be missed during system execution. For single processor systems, several schedulability analysis tools [15, 64] are well-established. However, these tools are now challenged by the rapid spread of multi-core processors in the real-time systems realm. This chapter focuses on the challenges brought by the multi-core systems in the real-time schedulability analysis based on model checking and on how the solution proposed in this thesis improves on related work.

The chapter is organized as follows: Section 3.1. outlines the main challenges faced by real-time formal schedulability analysis in the context of multi-core systems. Furthermore, in Section 3.2., a presentation of the results obtained until now using different formalisms applied for real-time schedulability analysis is given. As this thesis focuses on the use of timed automata [13] for schedulability analysis, Section 3.3. presents the syntax and semantics of timed automata and Section 3.4. gives an overview of existing timed automata frameworks for schedulability analysis. In the end of the chapter, a few concluding remarks on the presented topics are made.

3.1. Formal Verification Challenges

Analyzing the schedulability of a set of tasks running a multiprocessor platform faces several challenges:

(1) *State space explosion* - The factor that limits mostly the applicability of formal verification and model checking is the often excessive size of the state space. The state space size generally grows exponentially with the size of the problem. This exponential growth of the state space is referred to as the state-space explosion problem. For continuous (dense) time models the state space is infinite because of the real-valued clocks. The key idea that facilitates model checking on such models is to introduce some equivalence relation between states, which leads to a finite number of groups of states. Assuming that all clocks in the model increase at the same constant rate, the states in a group can be described using linear inequations which can be represented as matrices [41]. The state space

explosion problem appears also in models with discrete time semantics, although the state space in this case is finite.

(2) *Number of clocks* - The number of state groups generated while transforming a continuous time model into a system with a finite number of states grows exponentially with the number of clocks in the model and the number of constants with which clocks are compared. Consequently, in order to build a scalable system model, one must address this issue by maintaining a minimum number of clocks and constants used in clock constraints.

(3) *Clock rate* - Accurate modeling of multiprocessor systems requires the possibility to model the different speeds at which the processors may work. Linear Hybrid Automata may be used to model such systems as the dynamic evolution of variables in these is specified in each location by a differential equation. The problem of these models is that reachability is undecidable and schedulability analysis based on them can be only approximative [12].

(4) *Preemption modeling* - Preemptive scheduling policies, where the execution of a task can be suspended and resumed at a later time, require the possibility to stop and resume clocks. For dense-time models, one solution for modeling such clocks is the stopwatch automata formalism [59], a subclass of Linear Hybrid Automata. As the reachability problem is undecidable for Linear Hybrid Automata, it is also undecidable for stopwatch automata.

3.2. Formal Modeling Approaches

Formal modeling and verification of time constrained embedded software requires appropriate consideration of the time features employed by each formalism. In this section, we take a look at the most important time dependant formalisms used for real-time systems verification with focus on their applicability for schedulability analysis. Almost all these formalisms have been obtained as extensions of untimed ones.

The correct functioning of a real-time embedded system depends on qualitative as well as quantitative properties. A formalism widely used for verifying qualitative properties, like absence of deadlock or eventual occurrence of some event, is represented by finite-state automata and temporal logic. *Timed automata* [13] extend finite-state automata with real-valued clocks and enable specification of quantitative timing properties relating to occurrences of events. Due to their simplicity, several verification tools for timed automata have been developed to model and verify real-time systems, among them UPPAAL [112], Kronos [161] or HyTech [98].

Model-based schedulability analysis using timed automata has been introduced in [81] where timed automata are used for modeling task arrival patterns. In timed automata models, schedulability analysis is reduced to reachability analysis and, therefore, proving that a system is schedulable is reduced to showing that a given state of the timed automata system model is reachable or not. As a consequence, the ability to check schedulability of a system depends on whether the reachability problem is decidable or not. For uniprocessor systems, the schedulability checking problem for non-preemptive tasks is decidable [81, 109]. In the area of non-preemptive scheduling, timed automata have been used for job-shop scheduling [8] or for schedulability and compatibility checking of real-time objects [102]. For preemptive tasks with fixed execution time, scheduled on uniprocessor systems using any scheduling policy, schedulability is decidable by translating it into a reachability problem for another class of automata called *timed automata with subtraction* [81, 80, 79]. Unfor-

tunately, the number of clocks used in the model is proportional with the number of tasks, which influences the scalability of the model. An alternative would be to model preemptive tasks assuming a dense time model using stopwatch automata which can be stopped and resumed but the reachability of these automata is undecidable [59] in the general case [99]. Moreover, it is demonstrated that the schedulability problem is undecidable if the following conditions hold simultaneously [109, 79]:
(1) the execution times of tasks are intervals,
(2) the precise finishing time of a task influences the release of another task (feedback), and
(3) a task can be preempted by another task.

Partitioned multiprocessor scheduling can be seen as a superclass of uniprocessor scheduling and, consequently, the above result on undecidability is also true for partitioned multiprocessor schedulability. On the positive side, the schedulability of partitioned multiprocessor systems remains decidable for the following two cases [108]:
(1) tasks have variable execution time and the finishing time of a task influences the release of some other task but a non-preemptive scheduler is used,
(2) a preemptive scheduler is used and feedback on task execution is allowed but tasks have fixed execution requirements.

On the other side, schedulability is not decidable if tasks have variable execution times and a preemptive scheduling policy is used.

In global multiprocessor scheduling a task can be preempted from one processor and resume execution on any available processor. The solution used in [81, 80, 79] for keeping track of current execution time of the task cannot be applied anymore, and the only possibility would be to use stopwatches. Nevertheless, reachability for stopwatch automata is undecidable [99]. To address this issue a discretized preemption scheme can be used instead, since any computer control system is essentially discrete (e.g. for a CPU, the cycle time of the CPU can be seen as the discrete time step) and therefore it can be modeled using a discrete time assumption.

A special class of timed automata models which can also be used for schedulability analysis is the *parametric timed automata*. The methods for fixed-priority preemptive uniprocessor schedulability analysis introduced in [80] were extended using parametric timed automata with the goal of representing and analyzing the region of the task parameter space that corresponds to feasible schedules [162, 63].

Petri nets were developed as an untimed formalism for specifying concurrent systems. In the Petri net model the dynamics of the system is modeled by moving tokens. A Petri net consists of four components: places, transitions, directed arcs and tokens. The arcs can connect only a place to a transition or a transition to a place. Each place can contain zero or several tokens. The assignment of tokens to the places is called marking. Given a marking, a transition is enabled if it has at least a token in each of its input places. An enabled transition can fire consuming a token from each of its input places and adding a token in each of its output places. Over the time, several timed versions of Petri nets have been proposed differing from one another in the association of components with time constructs:

- *Timed Petri Nets* are derived from Petri nets by associating a finite firing time to each transition [145].
- *Time Petri Nets (TPN)* are Petri nets with labels: two values of time expressed as real numbers, x and y, are associated with each transition where $x < y$. x is the delay after which an enabled transition can fire and y is the deadline by which to fire the enabled transition [134]. TPNs are dense time models.

A number of authors propose extensions of time Petri nets especially designed to account for scheduling issues like task preemption:

Scheduling-TPN [146, 119] is an extension of TPN for analysis of task scheduling over a multiprocessor platforms where tasks are associated with places and each place has associated two supplementary parameters: one for the processor on which the task shall execute and one for the task priority. However, the Scheduling-TPNs introduced [146] can be used only for partitioned multiprocessor preemptive or non-preemptive scheduling with statically assigned priorities. The Scheduling-TPNs were extended in [120] to include the possibility of modeling dynamic scheduling policies like EDF. Automated support for analysis and visual editing of Scheduling-TPNs is given by the ROMEO tool [89].

Preemptive-TPN [53] also extends TPN and, in order to ease schedulability analysis, associates the set of requested resources and access priorities with transitions instead of places. Just like Scheduling-TPNs it can handle only fixed-priority, partitioned scheduling policies. Analysis methods for Preemptive-TPNs are implemented by the ORIS tool [159], which supports visual editing and interactive animation of Preemptive-TPN models, symbolic state space enumeration and interactive trace analysis.

Inhibitor Hyperarc TPN [147] introduce special inhibitor arcs that control the progress of transitions. Each transition has an associated stopwatch which can be paused and resumed by using inhibitor hyperarcs. An inhibitor arc between a place and a transition means that the transition can only be fired if the place is unmarked. Inhibitor Hyperarc TPNs are a superclass of Scheduling-TPNs and Preemptive-TPNs and consequently are also appropriate for modeling real-time tasks scheduling.

It has been shown that reachability of all above types of TPNs is undecidable [42]. For all types of TPNs exact computation of the state space is expensive and consequently state space over-approximation methods are available. These methods produce a linear hybrid automaton, bisimilar to the TPN, and exact analysis is performed on the automaton using available model checkers. Another disadvantage of TPNs is that it cannot model dynamic task creation as the number of tasks is set in advance and encompassed in the net. Even so, TPNs are still attractive due to their intuitive graphical representation and the wealth of available tools [96], although relatively new, with limited features.

The timing behavior of a system can be modeled also with the help of a process algebra. A process algebra is a concise language for describing the possible execution steps of computer processes. The *Algebra of Communicating Shared Resources (ACSR)* [115, 114] is a process algebra enhanced with the notion of time, such that it enables the analysis of real-time systems. ACSR is a discrete real-time process algebra and provides a set of timed operators that can be used to bound the execution time of a sequence of actions, to delay the execution of the sequence by a number of time units, and to timeout while waiting for specific actions to occur.

One way to check that the system satisfies some safety properties using the ACSR approach is to verify that a design specification of the system is correct with respect to a requirements specification by showing that the two processes representing respectively these two specifications are equivalent. To determine equivalence of the two specifications in ACSR, the behaviors of the system's processes are first translated

Table 3.1: Summary of multiprocessor real-time scheduling related features of timed automata (TA), TPN and ACSR formalism

	Priority assignment		Processor allocation		Pre-emption	Time model		Decidability
	static	dynamic	partitioned	global		continuous	discrete	
TA	√	√	√	√	√	√		√ (only partitioned)
TPN	√	√	√		√	√		×
ACSR + ACSR-VP	√	√	√		√		√	√

into a prioritized labeled transition system which is basically a state space graph used also in timed automata and which also suffers from the state-explosion problem.

Compared to other formalisms, like timed automata or Petri nets, writing and understanding the system specification in ACSR seem to be more difficult. ACSR like most real-time process algebra employs an implicit global clock which means that time progresses for all processes simultaneous and is not possible to model distributed systems with proper clocks advancing at different rates. Another major drawback is that ACSR cannot handle dynamic priorities of tasks and can handle only a discrete time model. However, ACSR is still attractive for its ability to express compactly several timing constraints which have to be explicitly written in other formalisms.

An extension of ACSR, called Algebra of Communicating Shared Resources with Value Passing (ACSR-VP) [40, 110], a process algebra with value-passing and parameterized processes, is able to model real-time systems with variable timing attributes and dynamic priorities. Using ACSR-VP it is possible to model preemptive or non-preemptive, periodic or aperiodic, independent or dependent task systems running on a uni- or multi-processor platform, scheduled by various static-priority and dynamic-priority scheduling disciplines. Through schedulability analysis of these systems one can determine values for the system parameters that make the system schedulable [110].

Table 3.1 presents a summary of the features each of the above discussed formalisms presents regarding real-time multiprocessor schedulability analysis.

In addition to the formal approaches for schedulability analysis presented above, several works explore the possibility of formalizing the real-time schedulability problem using some other, not so well established, formalisms.

Timed modules [14], a modular modeling language for continuous time systems, was used for encoding the functional and timing behavior of real-time tasks accessing shared resources [148]. Schedulability of real-time systems modeled using timed modules can be verified using model checking of Alternate-time Temporal Logic formulas which has been proved to be decidable. Using timed modules, static and dynamic priorities can be encoded for both partitioned and global multiprocessor scheduling strategies. However it is not possible to cope accurately with task preemption.

Currently, model-driven engineering is one of the major research areas in software engineering, but for it to be fully accepted by the real-time community, a fair amount of research is still required. A modeling language suitable for schedulability analysis must recognize elements like processes, shared resources or scheduling policy and ensure some consistency between different views of the system in order to guarantee that the model used for schedulability analysis is consistent with the one used for code generation. A proof-of-concept modeling tool is presented in[51], but the schedulability analysis employed by the tool is based on analytical methods like

the ones resented in Chapter 2. and, consequently, inherits their drawbacks. Another approach, based on Message Sequence Charts (MSC) is introduced in [103]. MSC specifications denote an execution scenario of the system and capture the ordering of the tasks. The schedulability analysis in [103] is based on annotating each message or event in an MSC with lower and upper bounds on their execution times and then, using these bounds, the response times of various events is checked against their deadlines.

After the examination of above presented time-dependent formalisms we can conclude that the main advantage that schedulability analysis based on formal verification has over the analytical methods presented in the previous chapter is that it supports description and analysis of complex tasking models running under preemptive or non-preemptive scheduling. In particular, the task model can be periodic, sporadic and aperiodic, with nondeterministic execution times, with semaphore synchronization and precedence relations deriving from interprocess communication. Furthermore, for some task models like the periodic one, the multiprocessor schedulability analysis gives exact answers while analytical methods provide, in most cases, only sufficient conditions. However the applicability of all the formal techniques is limited by the state space explosion problem.

3.3. Timed Automata Preliminaries

After the analysis on the formal verification approaches for real-time schedulability analysis in the previous section, we decided that the formalism that most suites our modeling needs is timed automata. Motivated by this reason, in this section we give basic descriptions and definitions related to the timed automata used in our work.

Formal syntax. Assume a finite set of real-valued clocks $\mathcal{C}$ and $\mathcal{B}(\mathcal{C})$ the set of constraints on the clocks in $\mathcal{C}$. The clock constraints (guards) are conjunctions of expressions of the form $x \bowtie N$ and $x - y \bowtie N$ where $x, y \in \mathcal{C}$, $N \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton over the set of clocks $\mathcal{C}$ is a tuple $\langle L, l_0, \Sigma, \mathcal{C}, I, E \rangle$ where

- L is a set of finite locations,
- l_0 is the initial location,
- Σ is a set of actions,
- $\mathcal{C}$ is the set of clock variables,
- $I : L \rightarrow \mathcal{B}(\mathcal{C})$ associates invariants to locations,
- $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of transitions, where transition $\langle l, g, a, r, l' \rangle$ from location l to location l', labeled with action a is executed only if guard g is true and resets clocks in $r \subseteq \mathcal{C}$.

All timed automata models presented in this thesis are based on the UPPAAL [112] model of timed automata which is extended with constructs such as constants, integers, committed and urgent constraints on locations, networks of timed automata and events transmitted between automata. An *urgent location* is similar to a location with all incoming transitions resetting a clock x and having associated an invariant $x \leq 0$ (i.e. time cannot pass while the automaton is in an urgent location).

Semantics. For a timed automaton we can define a clock valuation function $v : \mathcal{C} \rightarrow \mathbb{R}_+$ assigning positive real values to clocks in $\mathcal{C}$. A state s in the timed automaton is a pair (l, v) where $l \in L$ and v is a clock valuation. The automaton can stay in state s as long as the invariant associated to l is true or can execute transitions outgoing from l when the guard of these transitions is true. Therefore, two types of transitions can be defined:

- delay transitions: $(l, v) \xrightarrow{d} (l, v')$ where $v'(x) = v(x) + d, \forall x \in \mathcal{C}$ and v' preserves the

invariant of location l,

- action transitions: $(l, v) \xrightarrow{a} (l', v')$ if there exists a transition $\langle l, g, a, r, l' \rangle \in E$ and guard g is true for clock valuation v and v' is obtained from v by resetting all clocks in $r \subseteq \mathcal{C}$ and leaving all others unchanged.

In UPPAAL, each timed automaton has an associated set $\mathcal{V}$ of integer variables. These variables can be included in guards and can be updated when transitions are taken. This modifies the above presented semantics as follows. Besides location and clock assignments, a state s in a UPPAAL timed automaton also includes the current values of the variables, i.e. s is a triple (l, v, u) where $u : \mathcal{V} \to \mathbb{Z}$ is a variable assignment function. This also changes the semantics of action transitions as transition $(l, v, u) \xrightarrow{a} (l', v', u')$ requires additionally for variable assignments u and u' to be considered in evaluating guard g.

Networks of timed automata. A network of n timed automata $A_i = \langle L_i, l_i^0, \Sigma, \mathcal{C}, I_i, E_i \rangle$, $1 \leq i \leq n$ over a common set of clocks and actions is a parallel composition of A_i, describing a timed automaton obtained from its component automata. Semantically, the network of timed automata requires joint execution of delay transitions and synchronization over complementary action transitions.

For networks of timed automata, UPPAAL introduces the concept of committed locations. A committed location is more restrictive than an urgent location, as a state containing a committed location cannot delay and the next transition of the system must involve an outgoing edge from one of the committed locations in the state.

3.4. Timed Automata Frameworks

This section compares the proposed model-based schedulability analysis framework for component-based systems with existing timed automata based frameworks and describes how the proposed model-based analysis framework improves on these frameworks.

One of the first tools that makes use of real-time model checking for schedulability analysis is TIMES [15]. TIMES uses timed automata to model the tasks and the dependencies between them using a time-triggered architecture and verifies schedulability using the UPPAAL engine. However, until now the tool only offers support for analyzing uniprocessor systems. This thesis considers the problem of using formal methods for real-time multiprocessor hierarchical schedulability analysis.

TAXYS [64] produces a timed automata model capturing the temporal behavior of a whole application, including the external environment. The model is generated from the application program annotated with timing constraints and afterwards, model checking is done in Kronos. However, TAXYS' features are limited as the framework considers that the program consists of a single, non-preemptable thread.

Formal verification of component-based systems is addressed by several frameworks for various purposes. The Save Integrated Development Environment (Save-IDE) [149] offers support not only for design of component based systems, but also allows specification of the behavior of each component using timed automata, transformation of each component to tasks and setup of execution parameters, like priority and periodicity. Using UPPAAL and the timed automata models, it is possible to check if the components satisfy their requirements specified as formulas in a subset of timed CTL. In contrast to the work in this thesis, the verification features of the IDE do not allow specification of component-level scheduling strategies based only on component interfaces.

Ke et al. [106] also propose a methodology for formal verification of the timing

and reactive behavior of component-based systems. Unlike the work presented in this thesis, their approach assumes that tasks associated to a component execute on a single processor and each task is modeled by a separate timed automaton. In contrast, in this thesis the timed automata network which models a component uses a different approach in which a single automaton is used for all tasks of the component and the tasks are scheduled on a multiprocessor resource partition.

The global multiprocessor schedulability analysis using model-checking has been investigated for tasks with static priorities in [93]. The models in [93] allow restricted and full migration of task instances and use the discrete time semantics. Every task is modeled separately and the schedulability of tasks is checked in decreasing order of their priority which limits the applicability of the analysis to static scheduling policies. This also implies that for a task set with N tasks, model checking has to be performed N times in order to determine the schedulability of the entire set. With this approach a maximal number of $N+1$ clocks are necessary for a task set of size N. Unlike this model checking solution, this thesis addresses both static and dynamic scheduling policies. Moreover, the solution in this thesis requires just a single run of the model checking for the entire task set using a single clock in a setting with resources that are not continuously available and multiple levels of scheduling.

The MOVES analysis framework [140] proves the ability to apply UPPAAL to verify schedulability for real-time applications executing on Multiprocessor System-on-Chip (MPSoC) platforms where tasks are statically partitioned across processing elements. The applications can consist of periodic, sporadic and aperiodic tasks with hard, firm or soft deadlines and non-deterministic execution times, in an interval between a best and worst case. The framework uses the discrete time model and simulates different clock frequencies on processing elements by specifying the execution times of tasks as number of clock cycles and the other task parameters (i.e. period, deadline) as seconds. Each application is modeled as a task graph, i.e. a directed acyclic graph of tasks where edges indicate causal dependencies and, each task in the application is modeled as a distinct timed automaton. In contrast, this thesis assumes global multiprocessor scheduling is used and all tasks of a component or application are modeled as a single timed automaton.

The open-source DREAM [5, 129] model-based verification framework allows schedulability analysis of event-driven distributed real-time systems. The computational model in DREAM defines tasks, timers, event channels and schedulers. Tasks are triggered either by a timer or external aperiodic events and tasks communicate among themselves by means of an event channel. Each of the elements of the computational model is modeled using a timed automaton and schedulability analysis is transformed in reachability checking using the UPPAAL model checker. The schedulers in the DREAM framework assume that a fixed-priority preemptive scheduling policy is used. Just like the work in this thesis, a discrete time semantics is used for modeling preemption but, compared to our work, a partitioned scheduling scheme is used and task migration between processors is not allowed. Also, the processors are fully available to the tasks under analysis. DREAM also introduces model checking for schedulability analysis of preemptive event-driven asynchronous real-time systems with execution intervals [130]. Starting from the approach in DREAM, we propose a method for schedulability analysis of hierarchical based multiprocessor systems, where the tasks can migrate from processor to another, issue that is not addressed at all in DREAM.

3.5. Concluding Remarks

In this chapter the main issues related to multiprocessor schedulability analysis based on formal verification methods were addressed. First of all, there are several challenges posed by these methods, challenges which may limit their broad acceptance. The greatest challenge that must be overcome is the state space explosion and, although it can not be completely beaten for problems like schedulability analysis, it can be diminished by following some guiding rules like reducing the number of clocks in the model. A better and more general solution would be to improve the model checker in terms of less memory usage which would increase the size of systems that can feasibly be verified. Another challenge, specific to schedulability analysis, is modeling task preemption and it seems that the only method which guarantees to terminate and give a feasibility verdict is based on the use of a discrete time model instead of a dense one.

Furthermore, the most used formalisms for verifying real-time schedulability were analyzed and compared based on the multiprocessor scheduling features they possess. The analysis shows that, compared to Petri nets and ACSR, timed automata are the most suited for modeling real-time systems using a dense time model. However, this time model can be used only for scheduling partitioned multiprocessor systems, while for global scheduling, reachability on timed automata models is decidable only if discrete time semantics are employed. The same discrete time model is used by ACSR, but this process algebra cannot model global scheduling strategies. Despite of all their weaknesses, the presented formalisms allow simple analysis of different task models, inter-task interactions and timing constraints, while classical analytical methods for multiprocessor scheduling analysis are mostly limited to independent periodic or sporadic tasks.

During the recent years several scheduling frameworks based on timed automata have been developed. While some of them are dedicated to uniprocessor real-time systems, other handle the multiprocessor case assuming a partitioned scheduling policy. Few of them are dedicated to component-based systems but those do not aim multiprocessor hierarchical schedulability analysis, like the method proposed in this thesis.

4. MODEL CHECKING FOR VIRTUALIZED REAL-TIME MULTI-CORE SYSTEMS

4.1. Motivation

In the recent years, real-time embedded software development has focused more and more on building flexible and extensible applications. Component-based software systems achieve these objectives by gluing individually designed, developed and tested software components, each component having different timing requirements. Therefore, when building such a component-based system one must ensure that components can coexist without jeopardizing each other's execution.

For single processor systems, Harbour [95] proposed a solution for temporal isolation of component-based applications based on service contracts. *Contract-based scheduling* is a hierarchical scheduling scheme in which every application or application component may have a set of service contracts describing its minimum resource requirements. These contracts are used in online or offline negotiations to determine if the resource requirements can be guaranteed or not. The contract-based scheduling model was extended for multiprocessor systems by Chang et al. [60].

In the contract-based scheduling solution for multiprocessors, each application has a contract with the underlying operating system specifying the application's resource requirements. Furthermore, each component of the application also has a contract. The contracts of the components specify how to distribute the resources requested by the parent application among the components. The multi-core/multiprocessor time partitions (see Section 2.4.1.) are used to support the contracts for the application level, while components contracts are facilitated by execution time servers. While scheduling windows are bound to a processor or core, servers can migrate from one processor to another. Consequently, the contract-based scheduling model falls in the class of hierarchical multiprocessor global scheduling algorithms with preemption and full migration.

The analysis in [60] starts from the minimum computation time provided during a time interval by the time partition to the execution time servers and, respectively by the servers to the tasks of the components. This is used as an upper bound on the maximum workload of the servers and tasks such that the system is schedulable. The workload of the servers and tasks is computed assuming a fixed scheduling policy is employed. The result of the analysis gives only sufficient and quite pessimist schedulability conditions. Other drawbacks of the analysis are the assumption that tasks are independent and the limitation of the scheduling policy to the fixed priority class.

This chapter describes how we apply timed automata theory to analyze the schedulability of real-time component-based applications running on multi-core platforms. The resource requirements of each application or application component are specified in a service contract resulting a hierarchy of contracts like the one used in contract-based scheduling. We use model checking and transform the schedulability analysis problem into reachability checking on a timed automata model of the service contracts.

Because the reachability problem for timed automata with dense time se-

mantics is undecidable for our schedulability problem, we use a discrete time model. We do not consider this a limitation since on a real processing element, execution of instructions is carried out in clock cycles. In our model of the execution platform, time is divided into clock cycles as well and each task takes a number of clock cycles to complete. Between clock cycles, the operating systems only runs, if there is any change (ready or finished tasks), and schedules a task to run in the next clock cycle. The overhead time for the operating systems decisions on the execution platform is assumed be zero.

4.2. Exact Schedulability Analysis for Independent Tasks

4.2.1. Problem Formulation

The analytical or utilization-bound schedulability tests for multi-core contract-based scheduling give only sufficient and quite pessimistic conditions and, as a consequence, components, which may in fact be schedulable by the system, will be rejected. This can lead to poor utilization of the multi-core platform. Moreover, the analytical methods assume only a rather simple periodic or sporadic task model and are bound to a rather specific scheduling policy. Any change in any of these hypothesis requires resumption of the analysis process and results in a new testing methodology, completely different from the previous.

In view of these drawbacks of analytical methods, we consider that a more flexible methodology is required for the analysis of contract-based scheduling, providing more accurate results than analytical methods. In this section, we propose an exact schedulability method for hierarchical multi-core global scheduling with fixed or dynamic task priorities and preemption enabled by using model checking and transforming the schedulability problem into a reachability problem on a timed automata model [126]. Besides the quite flexible way of dealing with different scheduling policies, the proposed method has the additional benefit of being easy extensible in order to handle different types of task interactions, as we will show in Section 4.3.

Using model checking on timed automata models for global multiprocessor preemptive scheduling has to overcome a series of challenges:

(1) *Preemption and migration modeling*: Schedulability analysis using timed automata has been applied successfully for non-preemptive scheduling policies. However, in timed automata models as defined in [13] time elapses at the same rate for all components and therefore they cannot be used for preemptive scheduling policies where execution of tasks can be suspended and resumed later. One possible solution, proposed in [81, 80], is to use a subclass of timed automata, Bounded Timed Automata with Subtraction. Unfortunately, this solution can be used only for uniprocessor or partitioned multiprocessor preemptive scheduling because it can not handle task migration. Stopwatch automata [59], a subclass of Linear Hybrid Automata, have been proposed as a solution for modeling preemptable tasks, but only approximative methods for reachability checking are known for these automata [59].

(2) *Decidability*: This problem refers to whether it is possible to say if a certain state in the timed automata model is reachable or not. Assuming a dense time model, the schedulability checking problem has been shown to be decidable for non-preemptive tasks. Preemptive uniprocessor scheduling is decidable only if tasks have either variable execution times and there is no dependence between them or tasks have precedence constraints between them but fixed execution times [109, 79].

This result is partially true for partitioned multiprocessor scheduling in the sense that schedulability is not decidable if tasks have variable execution times and a preemptive scheduling policy is used. Using stopwatch automata for modeling task preemption is undecidable [99], no matter the scheduling policy. For global preemptive multiprocessor scheduling, schedulability is decidable only if a discrete time model is used.

(3) *Scalability*: The major source of skepticism in adopting model checking for scheduling analysis is its rather limited scalability. Until now, even assuming a simple task model, only systems with few tasks have been successfully tested.

In this section we propose an exact schedulability method for hierarchical multi-core global scheduling policies. We use the timed automata formalism for our method and in order to overcome the preemption and decidability challenges, the model employs a discrete time semantics. In order to improve the scalability of our model, we make use of only one continuous clock and use a minimal number of constraints for that clock.

Our exact schedulability analysis method is based on a model and that model has its own assumptions. We start in this section with a simple model, which we extend in the next section to become closer to a realistic system. In this section of the thesis, we will make the following assumptions:

a1. The characteristics of the tasks (arrival times, periods and execution times) and of the resources (offset, budget, replenishment period, length or scheduling window period) are given as requirements to the scheduling algorithm and do not change at scheduling time.

a2. All tasks are periodic.

a3. The scheduling algorithm has succeeded only if all task deadlines are met. If a task misses its deadline it is considered a failure.

a4. The only shared resource in the system is the processor.

a5. The speed of the multi-core processor does not change.

a6. A task cannot execute on two or more cores simultaneously, and a core cannot execute two or more tasks simultaneously.

a7. Preemption is permitted at any time but we do not consider any overheads.

a8. Task and server migration is allowed at any time and no overhead is associated with migration.

a9. Components of an application are independent of each other. They share only the time partition supplied to the parent application.

a10. The priorities of the tasks can be fixed or dynamically assigned.

a11. The tasks of each component are independent, that is tasks do not share any resource other than processor and the arrival time of any task is not conditioned by the end of another task.

a12. The execution time of a task is a constant and does not variate within an upper and lower bound.

4.2.2. Contract-based Scheduling Model

This section presents the formal model of the service contracts. As explained in Section 2.4.2. there are two levels of such contracts. The first level specifies the resource requirements of a single application while the second level describes the requirements of each individual component of the application. Corresponding to the two levels of contracts there are two scheduling levels. At the upper level, each component of an application has a scheduler for scheduling its tasks, while at the lower level there is an application scheduler which manages the servers associated with each component of the application. This hierarchical contract-based scheduling model is depicted in Figure 4.1. Each of the blocks in the figure are detailed in the following subsections.

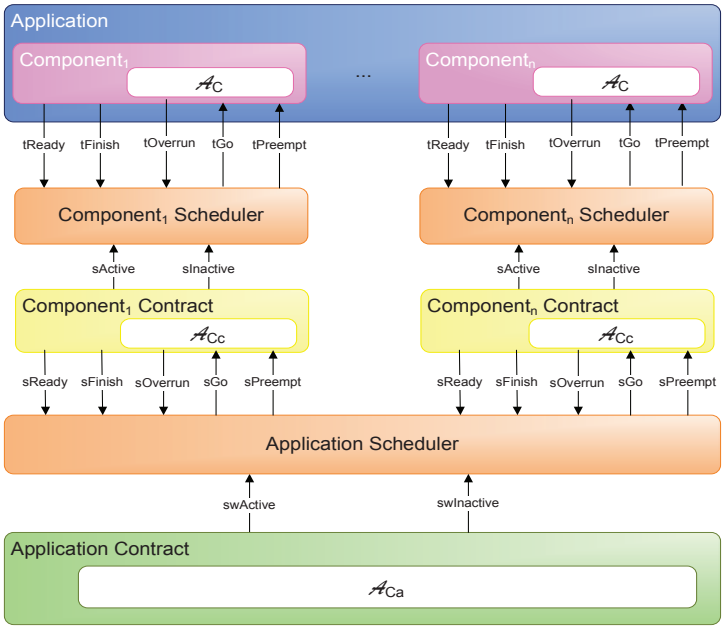


Figure 4.1: Contract-based scheduling model

Component Contracts

A component consists of a set of tasks, which may execute in parallel. Each task is characterized by a worst case execution time, a relative deadline, a period and is independent of the other tasks.

Definition 1 (Component)**.** *A component C consists of a finite set of n tasks $\mathcal{T}$ and a timed automaton $\mathcal{A}_C$ where:*

- *a component task $\tau_i \in \mathcal{T}$ is a tuple $\tau_i = (C_i, T_i, O_i, D_i)$, with C_i being the worst case execution time of the task, T_i the inter-arrival time between different instances of the same task, O_i the first release of the task and D_i is the deadline of the task where $C_i \leq D_i \leq T_i$,*
- *tasks may execute in parallel and are independent of each other,*

- *$\mathcal{A}_C$ models the execution of tasks in set $\mathcal{T}$ by taking transitions labeled with actions $tReady_i$, $tFinish_i$ and $tOverrun_i$, $\forall\, 1 \leq i \leq n$ representing the release and ending of task τ_i, and actions tGo_i and $tPreempt_i$ through which the component scheduler notifies task execution start/restart and suspension.*

The tasks of the component will be executed according to a component specific scheduling policy implemented by a scheduler associated with the component. The parameters of the tasks along with the task arrival pattern determine the resource requirements for the component. These resource requirements can be supported using one or more execution time servers, depending whether the tasks must execute in parallel or not. The period, deadline and budget of the servers associated with a component are specified in the component contract.

A server is defined by a tuple (Q, P, O) where Q is the capacity of the server, P is its replenishment period (i.e. the server becomes active every P time units) and O is the time of its first release. Each server may also have a deadline equal to its period. It is assumed there is a finite set of servers $\mathcal{S}$ containing the servers for all the components of an application.

Definition 2 (Component contract)**.** *A component contract $\mathcal{C}_C$ supported by a set of n_s execution servers $\mathcal{S}_C \subseteq \mathcal{S}$ is a timed automaton $\mathcal{A}_{Cc}$ over the set of actions $\Sigma_{\mathcal{C}}$ such that:*

- *$\mathcal{A}_{Cc}$ specifies the activation pattern of servers $\sigma_i \in \mathcal{S}_C$, $1 \leq i \leq n_s$.*
- *$\Sigma_{\mathcal{C}}$ is split in two sets:*
  - *output actions: $\Sigma_{\mathcal{C}}^O = \{sReady_i, sFinish_i, sOverrun_i, sActive_i, sInactive_i \mid 1 \leq i \leq n_s\}$*
  - *input actions: $\Sigma_{\mathcal{C}}^I = \{sGo_i, sPreempt_i \mid 1 \leq i \leq n_s\}$.*

The $\mathcal{A}_{Cc}$ automaton sends the output action $sReady_i$ to the scheduler associated with the application as soon as server σ_i is ready for execution and sends $sFinish_i$ or $sOverrun_i$ to the same scheduler to notify it that the server has finished its execution, respectively missed its deadline. As a response to its actions $\mathcal{A}_{Cc}$ can receive from the application scheduler sGo_i, telling it that server σ_i can start its execution, or $sPreempt_i$ which results in server σ_i being suspended from execution until the next sGo_i action. Actions $sActive_i$ and $sInactive_i$ are used to announce the component scheduler that server σ_i has consumed all its budget, respectively that it has replenished its budget and can be used again to execute tasks.

Application Contracts

As proposed in [60] the application contracts are supported by a multi-processor time partition model. Each application is associated with a time partition which has a local scheduler to execute the execution time servers assigned to the components of the application.

In a single processor system a time partition is implemented as a fixed-length major time frame composed of several scheduling windows. A scheduling window is defined by an offset to the beginning of the major time frame and by its length. The scheduling scheme of the major time frame repeats periodically during the execution of the system such that all scheduling windows are periodic. In a uniform multiprocessor or multi-core system, we assume there is a major time frame for each processor, but frames on all processors will have equal length and will be synchronized. The scheduling windows of frames on different processors can be different.

From the above specification we derive next a formal definition of the multiprocessor time partition.

Definition 3 (Time partition)**.** *A time partition $\mathcal{TP}$ in a uniform multiprocessor or multi-core system is described by a set of major time frames $\{\mathcal{F}_i \mid 1 \leq i \leq m, length(\mathcal{F}_i) = L\}$, one for each of the m processors/cores in the system, where $\mathcal{F}_i$ is a set of scheduling windows with periods that are an exact divisor of L.*

In our setting the time partition is used to facilitate application contracts. In a simple scenario, the application contract could specify a few pairs of period and length values which upon successful negotiation of the contract could be mapped to a set of scheduling windows.

Definition 4 (Application contract)**.** *An application contract $\mathcal{C}_\mathcal{A}$ is a pair $(\mathcal{TP}, \mathcal{A}_{Ca})$ where:*

- *$\mathcal{TP}$ is the multiprocessor time partition supporting the contract, and*
- *$\mathcal{A}_{Ca}$ is a timed automaton over the action set Σ_{SW} modeling the scheduling scheme of the major time frame:*
  - *$\Sigma_{SW} = \{swActive_k, swInactive_k\}$, where k is a scheduling window in $\mathcal{TP}$.*
  - *action $swActive_k$ signals to the application scheduler that the scheduling window k is now active, while $swInactive_k$ signals its deactivation.*

4.2.3. Timed Automata Model

As shown in the previous section both component and application levels include three automata - one for generating tasks or servers according to a given release pattern, one for generating the resource partitions (servers or scheduling windows) on which the tasks and servers, respectively shall be executing and one for scheduling. Notice that servers can be both schedulable entities (i.e. when referring to the application scheduler) and resources (i.e. for the component scheduler). For this reason in the rest of this section they are referred simply as tasks and, respectively as resources. Also, this plurality of roles implies that the timed automaton generating tasks for the application level is the same with the one generating resources for the component level. Therefore, this automaton can be deduced immediately from the task generator and the resource generator automaton types. The rest of the section is dedicated to given detailed descriptions of each of the three types of automata. In addition to the three types of automata, the model also includes a *Timer* automaton (see Figure 4.2) which uses a single continuous clock t and each time this clock ticks sends a *tick* signal to the task generator and the resource generator automata.

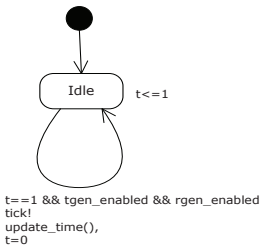


Figure 4.2: The Timer automaton

We first introduce some notations. Let $C(i)$, $T(i)$, $D(i)$, $R(i)$ and $E(i)$ denote the worst case execution time, the period, the deadline, the next release time and the current execution time, respectively for each task τ_i. For each task τ_i it is defined a

status variable $status(i)$ that is initialized to *idle* meaning that a task instance has not been released yet. The value $status(i) = ready$ is used to denote that a task instance of τ_i is ready for execution (i.e. it has just been released or was preempted). Let $status(i) = running$ stand for the fact that a task instance of τ_i is currently running on one of the active resources. To denote that an instance of task τ_i has finished or has missed its deadline we use $status(i) = finish$ and $status(i) = overrun$, respectively.

Task Generator Automaton

Model checking of preemptive scheduling algorithms could be done using a stopwatch model but it has been proved that schedulability of these models is undecidable [59]. Therefore, in order to address task preemption a discrete time formalism is adopted for the model proposed in this paper. This leads to a limitation as all task parameters (i.e. worst case execution time, period, deadline, release time) must have integer values.

In order to be able to determine the actual execution time of a task, a variable $E(i)$ is used for keeping track of the time task τ_i has executed since its last release. Each time the task is released $E(i)$ is set to 0 while $R(i)$ is set to the time of its next release. When the task generator automaton receives a *tick* signal from the *Timer* automaton it increases $E(i)$ for tasks with $status(i) = running$ and decreases $R(i)$ for all tasks with a value MIN representing the minimum between the time for the next release of a task or of a resource and the time for the next termination of a task or deactivation of a resource. In other words, $E(i)$ acts like a discrete clock which can be suspended and resumed.

Instead of using a task generator for releasing all n tasks of a component according to some pattern, it would have been possible to define a timed automaton for each of the n tasks, each automaton with a clock, leading to a total of n clocks. Since the state space of timed automata grows exponentially with the number of clocks in the model, the approach taken in this paper is superior to this one.

Figure 4.3 shows the main locations and transitions in the task generator automaton, leaving out some self-loop transitions. All white locations in the figure have the semantics that the system cannot delay in those locations and the next transition must involve an outgoing edge from one of them.

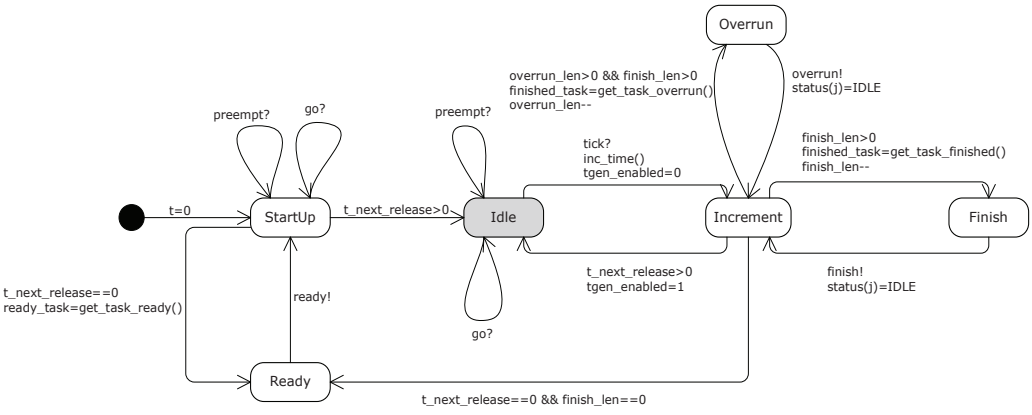


Figure 4.3: The task generator automaton

The task generator automaton uses a variable $t\_next\_release$ to remember the time until the next task is released. At start-up this variable is initialized with the smallest $R(i)$ and, if after that, $t\_next\_release = 0$ the automaton goes to the *Ready* location and selects a task τ_i for which $R(i) = 0$, updates $t\_next\_release$, sets the shared variable $ready\_task = i$ and sends the $ready$ signal to the scheduler automaton. Once $t\_next\_release$ becomes greater than 0, the generator moves to the *Idle* location where it waits for the next tick of the *Timer*. When the *tick* signal arrives the transition to the *Increment* location is taken and $inc\_time()$ updates the values $status(i)$, $E(i)$ and $R(i)$ as follows:

- for all tasks τ_i with $status(i) = running$, $E(i) = E(i) + MIN$ and if $E(i) = C(i)$ then $status(i) = finished$,
- for all tasks τ_i $R(i) = R(i) - MIN$ and $t\_next\_release = min(R(i))$,
- for all tasks τ_i running or ready for execution with $E(i) < C(i)$ and $T(i) - D(i) = R(i)$ sets $status(i) = overrun$.

Next, for all tasks τ_j that have finished, the variable $finished\_task$ is set to j and the $finished$ signal is sent to the scheduler which will free the resources used by these tasks. If any task τ_j has missed its deadline an $overrun$ signal notifies the scheduler which as a result will go to an *Error* location. After signaling all task finish events the generator checks to see if there is any task ready for execution and goes back to the *Ready* location.

Resource Generator Automaton

The task generator automaton presented above can be used to generate servers which act as resources for the component level. By adding just two signals - *active* and *inactive* - to notify the scheduler about the availability of the resources the task generator automaton becomes a resource generator automaton with the property that those resources are preemptable. If resources are not preemptable (i.e. the scheduling windows of a time partition) the resource generator automaton is a simplified version of the task generator.

Figure 4.4 presents the non-preemptive version of the resource generator automaton. The automaton keeps a discrete clock $RE(k)$ for each resource r_k. Also $RR(k)$ is used to remember the time until the next activation of resource r_k and two variables named $r\_next\_release$ and r_next_finish hold the time until the next resource activation and, respectively deactivation. When resource r_k is activated $RE(k) = L(k)$ where by $L(k)$ we denote the length of the resource's activation period. At every *tick* signal received from the *Timer*, for all active resources r_k, $RE(k)$ is decreased with the value MIN and variables $r\_next\_release$ and r_next_finish are also decreased with the same value. When $r\_next\_release$ reaches 0 all resources r_k with $RR(k) = 0$ are activated. If r_next_finish becomes 0 than all resources r_k with $RE(k) = 0$ are deactivated.

Scheduler Automaton

As it can be seen from the definitions in the previous sections, the component scheduler and the application scheduler have rather similar behavior. Both of them must schedule a set of periodic tasks/servers with deadlines less or equal to their period. The component tasks are scheduled on execution time servers which may be active or inactive. It is possible for two or more servers to be active simultaneously which implies that two or more tasks may run in parallel. For the application scheduler the tasks to be scheduled are actually the servers used by the component scheduler as

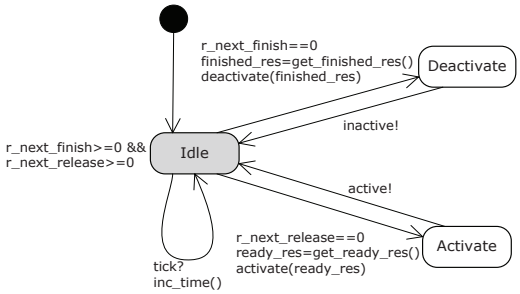


Figure 4.4: The non-preemptive resource generator automaton

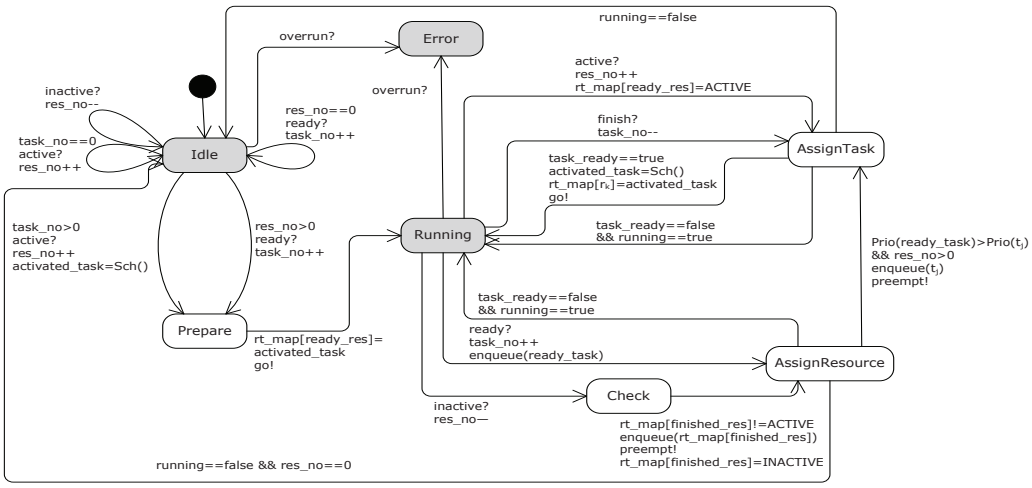


Figure 4.5: The scheduler automaton

resources. The servers are scheduled for execution on the scheduling windows of a time partition. The scheduling windows represent the resources allocated to the application by the system. As more scheduling windows can be active simultaneously, parallel execution of the servers is also possible.

A scheduler automaton for a service (i.e. application or component) contract has the following characteristics:

- has a queue holding the tasks ready for execution,
- implements a preemptive scheduling policy *Sch* representing a sorting function for the task queue,
- maintains a map between active resources (servers or scheduling windows) and tasks using those resources, and
- has an *Error* location which is reached when a task misses its deadline.

To record the status of a resource, let $rt\_map(j)$ be a map where $rt\_map(j) = inactive$ denotes that resource j is inactive, $rt\_map(j) = active$ means that resource j is active but no task is executing on it, and $rt\_map(j) = i$ denotes that resource j is active and is currently used by task τ_i.

Figure 4.5 shows the scheduler automaton. The locations of the automaton have the following interpretations:

1. *Idle* - denotes the situation when no task is ready for execution or no resources are active,
2. *Prepare* - a task has been released and a resource is active after a period during which either there were no tasks to schedule or no active resources,
3. *Running* - at least one task is currently executing,
4. *AssignTask* - a task has just finished and as a result an active resource can be used to schedule another ready task,
5. *AssignResource* - a task has just been released or a resource has just become inactive leaving its assigned task with no resource on which to execute; consequently the task has to be enqueued and if it has the highest priority in the queue according to *Sch* then an active resource is assigned to it,
6. *Check* - a resource has become inactive,
7. *Error* - the task set is not schedulable with *Sch*.

The scheduler enters the *Idle* location when either there are no ready tasks, no active resources or both of these conditions hold. As long as new tasks are released for execution but there are no active resources on which the tasks to be executed (i.e. $task\_no > 0 \; and \; res\_no = 0$) or as long as there are available resources but no ready tasks (i.e. $task\_no = 0 \; and \; res\_no > 0$) the scheduler stays in the *Idle* location. If the scheduler receives a $ready$ signal meaning that task τ_{ready_task} has been released and $res\_no > 0$ the scheduler goes to the *Prepare* location. Leaving the *Prepare* location for the *Running* location, it assigns the task to one of the active resources by setting $rt\_map(j) = ready\_task$, sets the variable $activated\_task = ready\_task$ and sends a go signal to announce the task generator automaton that task $\tau_{activated\_task}$ is running. After the scheduler has reached the *Running* location, it will leave this location if one of the following situations happen:

- the resource r_k becomes active (signaled by the $active$ signal and $ready\_res = k$): this is marked by updating $rt\_map[k] = ACTIVE$ on the transition to the *AssignTask* location. If tasks are ready for execution than the scheduler will assign the highest priority task τ_j to resource r_k by setting $rt\_map[k] = j$ and will notify the task generator with the signal go on a transition back to the *Running* location.
- a new task τ_i has been released (signaled by the $ready$ signal and $ready\_task = i$): the task is enqueued by setting $status(i)$ to $ready$ on the transition to the *AssignResource* location. If task τ_i is the highest priority released task and there are active resources, then τ_i must start executing. If there is a free active resource then task τ_i is assigned to it, otherwise the lowest priority task is chosen from the running tasks, preempted and the automaton goes to the *AssignTask* location. On the transition from *AssignTask* to *Running* the resource is assigned to τ_i and a go signal is sent to the task generator to notify it that task τ_i has started running.
- the resource r_k becomes inactive (signaled by the $inactive$ signal and $finished\_res = k$): this is marked by updating $rt\_map[k] = INACTIVE$ on the transition to the *Check* location. If the deactivated resource was free and there are still running tasks but no tasks in the queue then the transition back to *Running* location is taken. If a task τ_i was using resource r_k then the scheduler must set $status(i) = ready$ and go to *AssignResource* location. Should the resource r_k be the last active resource the scheduler would simply preempt task τ_i and go back to the *Idle* location, otherwise an active resource is searched analog to the situation when a new task is released.
- the task τ_j finishes (signaled by the $finish$ signal and $finished\_task = j$): the resource used until now by τ_j can be assigned to the highest priority task waiting

in the queue, if there is such a task.
- the task τ_i misses its deadline (signaled by $overrun$): the scheduler automaton goes into the *Error* location.

4.2.4. Performance Evaluation

This section presents an evaluation of the performance and scalability of model checking the contract-based scheduling model with independent tasks. The experiments were run on a machine with Intel Core 2 Quad 2.40 GHz processor and 4 GB RAM running Ubuntu. The analysis of the model was automated using UPPAAL and the utility program memtime [3] was used for measuring the model checking time and memory usage. Although the proposed model addresses scheduling at two levels, namely task level and server level, experiments were conducted only for the server level as we consider the analysis of the task level is just a replica of the server level due to the similarities between the two levels. In all experiments, to verify schedulability we checked if property *A[] not Error* holds meaning that the Error location is never reached.

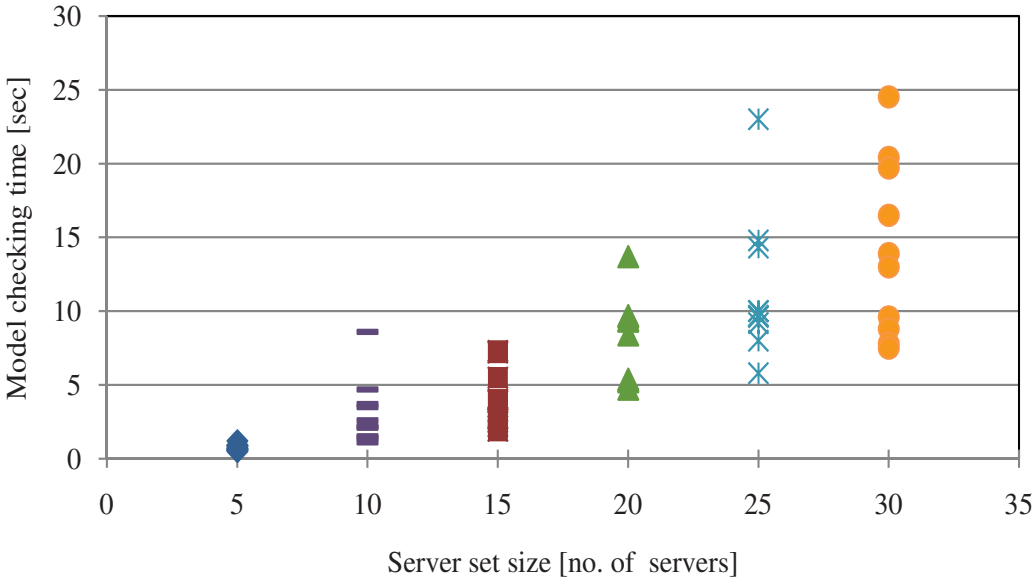
We measured the performance of our method with two metrics: *scalability* of the model checking process in terms of running time and peak memory usage and *accuracy* of the schedulability analysis decisions when compared with the analytical method of Chang et al. [60].

First, the model-based schedulability analysis was compared to the method based on schedulability bounds proposed in [60]. In the experiment 200 server sets with 30 servers and 200 server sets with 20 servers were used. Each server period was chosen randomly in the interval [10, 200], server utilization was a randomly generated number between 1 and 4.5 and server offset was set to 0 for all tasks. All servers had deadline equal to their period. The time partition used in the experiment had 9 scheduling windows with a total utilization of 4.5 and a major frame length equal to 50. The priorities of the servers were assigned according to the RM policy. In the experiment, from the 200 server sets of size 30, the model-based analysis accepted 182, while the classical method accepted only 83. For the server sets with 20 servers, the model-based method accepted 155 sets and the classical method accepted 75 sets (see Table 4.1). It can be seen that the results of the classical method are clearly pessimistic as it rejected a large number of server sets that are actually schedulable.

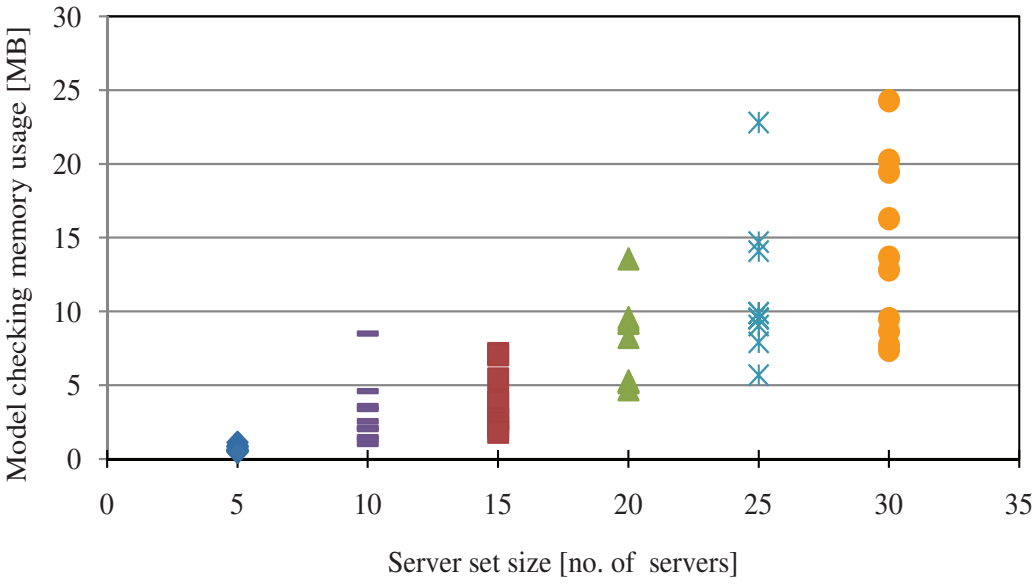
In order to observe the behavior of the model for different number of application servers we have used randomly generated sets of servers with periods in the range $[10, 100]$ and utilizations (i.e. $budget/period$) generated with a uniform distribution in the range $[0.05, 1]$. The offset of each server was set to a value equal to the period multiplied with a randomly generated number in the interval $[0, 0.3]$. Also, the servers sets were accommodated by a time partition with 9 scheduling windows and a total utilization of 4.5. Figure 4.6 shows how the model checking time and memory usage

Table 4.1: Comparison of schedulability analysis results using the analytical method in [60] and the proposed method using reachability checking

	No. of schedulable servers	
	20 servers/set	30 servers/set
Analytical method [60]	75	83
Reachability checking	155	182

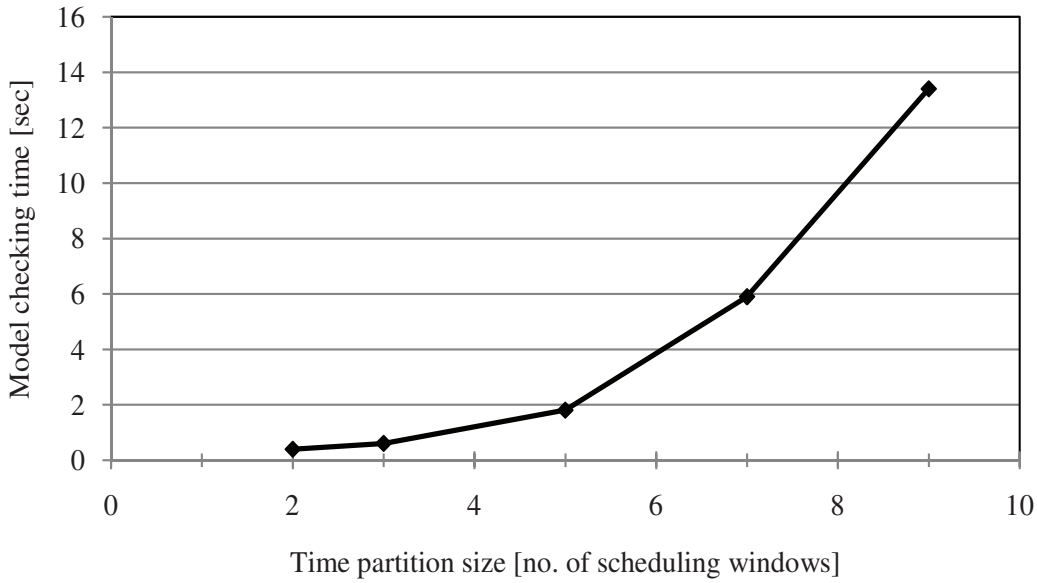


(a) Model checking time

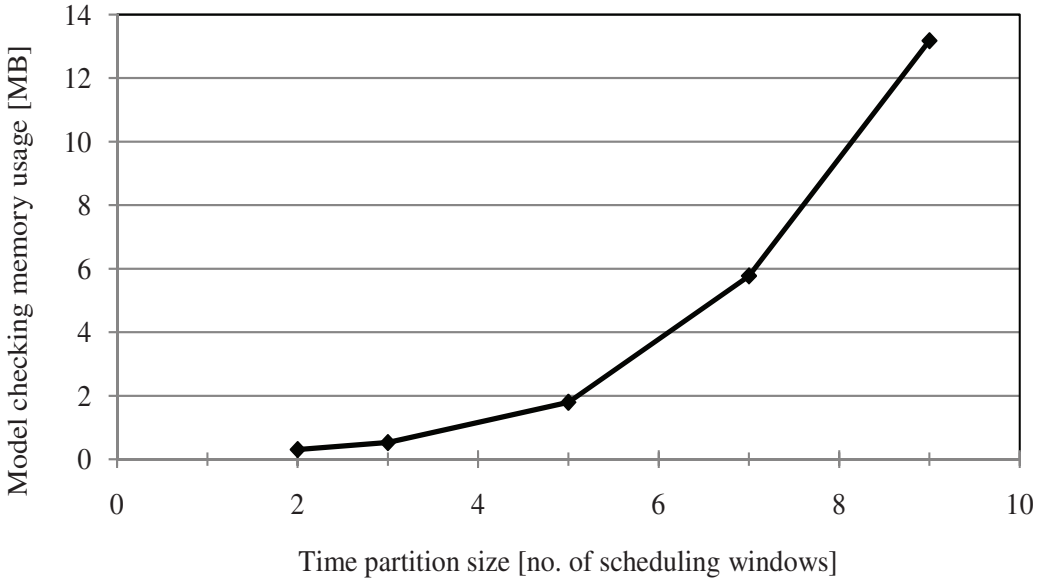


(b) Model checking memory usage

Figure 4.6: Influence of server set size on model checking performance



(a) Model checking time



(b) Model checking memory usage

Figure 4.7: Influence of time partition size on model checking performance

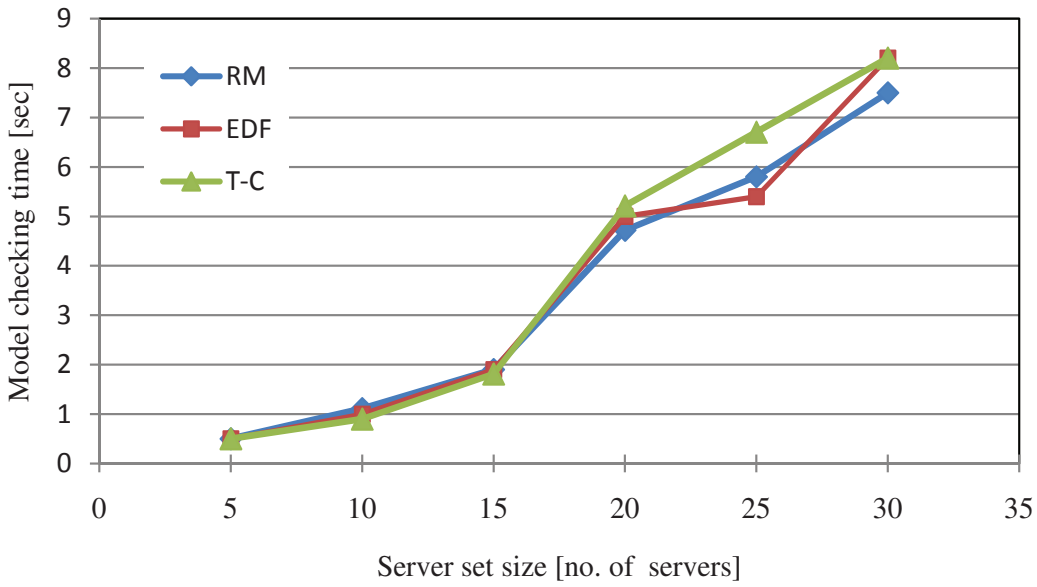


Figure 4.8: Influence of scheduling policy on model checking time

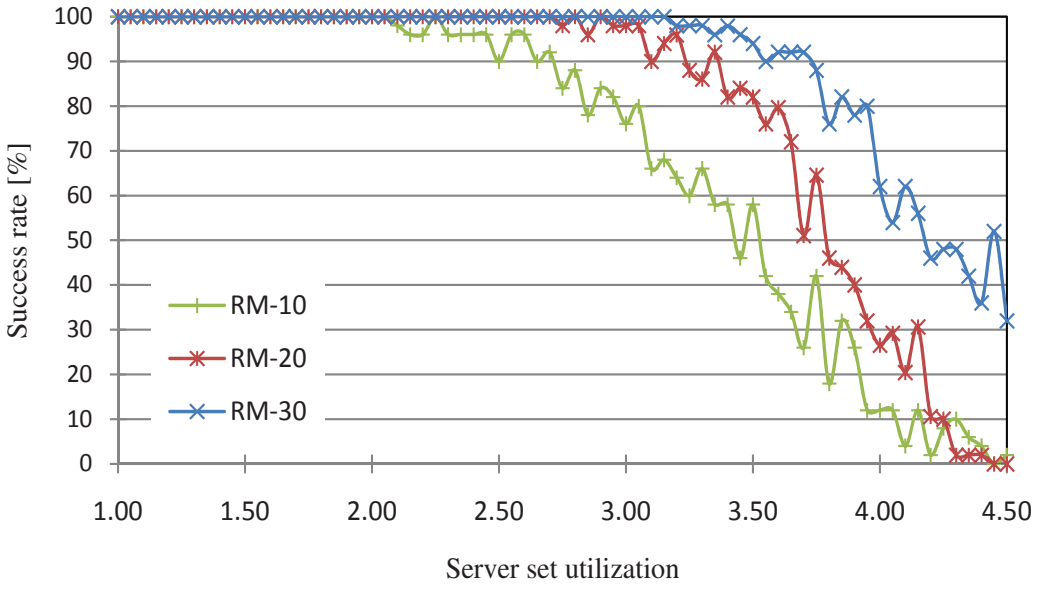


Figure 4.9: Schedulability of server sets

increase with the number of servers in the set. Also it can be noticed that for the same size of the server set the performance of the model checking can vary between rather larger limits (e.g. for sets of 30 servers the model checking time grows from 7 seconds to approximatively 25 seconds). This is due to the size of the hyper-period of the server sets, larger the hyper-period larger the model checking time and memory consumption.

Next, we analyzed the scalability and performance of model checking when the number of scheduling windows in the time partition accommodating the servers varies. For this, sets with 25 servers each and parameters in the same limits as for the previous experiment were generated and time partitions with 2, 3, 5, 7 and 9 scheduling windows were tested. In Figure 4.7 it can be seen that both the time for checking the model and the memory usage grow with the number of scheduling windows in the time partition.

In the previous two experiments the server sets were scheduled using the RM priority scheduling policy. The goal of our next experiment was to determine the impact of the scheduling policy on the model checking time and peak memory usage. The same time partition configuration as in the first experiment was used and sets of 5, 10, 15, 20, 25 and 30 servers were scheduled using both the RM, the EDF and the (T-C) (i.e. the higher the difference between the period and the budget of a server the lower its priority) scheduling policies. As can be seen in Figure 4.8 the scheduling policy has little influence on the performance of the model checking.

In the last experiment we are interested in seeing what is the influence of the server set utilization on the schedulability analysis. We have used the same time partition as in the second experiment with a total utilization of 4.5 and server sets of 10, 20 and 30 tasks with utilizations between 1 and 4.5 scheduled using the RM policy. Figure 4.9 depicts the number of schedulable server sets identified by our analysis. It can be noticed that even if the total utilization of a server set is maximal with respect to the available resources, our analysis is able to determine its schedulability, which is a clear advantage over the pessimist schedulability bounds presented in [60].

4.3. Approximative Schedulability Analysis for Task Graphs

4.3.1. Problem Formulation

Providing formal guarantees on the schedulability of component-based systems running on multi-core platforms becomes even more important if these components consist of interacting tasks, represented as task graphs, and if each component can have a different scheduling strategy. Until now, no analytical method for schedulability analysis of such systems has been developed.

Stopwatch automata [59] have been proposed for modeling of preemptive tasks with or without precedence constraints between them [9], but reachability of composition of these automata is decidable only for a single initialized stopwatch automaton [99] and undecidable [107] in the rest of the cases. Moreover, it has been shown that many preemptive multiprocessor scheduling algorithms suffer of scheduling anomalies when there is a change in their execution time [91]. Therefore, an accurate analysis of systems using these algorithms must consider variable task execution times. Moreover, it has been demonstrated in [108] that the schedulability analysis problem for multi-processor systems is undecidable if (1) the tasks are scheduled using a preemptive scheduling strategy, and (2) the tasks have execution times ranging over a continuous time interval.

We propose a formal method for checking the schedulability of real-time component-based applications running on multi-core platforms. Although the analysis method is based on the same essential idea as the work in [130], there are several significant differences. First, we assume a hierarchical scheduling model which means that the execution of tasks is constrained by the availability of the temporal partitions or execution servers and we consider also this resource aspect in our method. Second, unlike [130], task migration is allowed in our model. Last, instead of modeling just the tasks of an application individually, we model a whole scheduling level. We assume that the multi-core contract-based scheduling model described above is used and we adapt the previous timed automata model to encompass a new task model allowing task precedence constraints. The timed automata model is actually the model of a level in a multi-core scheduling hierarchy. The proposed method is an approximation method for schedulability verification and uses a discrete time formalism, while at the same time is able to capture continuous task execution times by approximating the stopwatch automata model. We show that our model approximates the stopwatch model by proving that the formal language accepted by the timed automata model is included in the language accepted by the stopwatch automata and we evaluate the approximation errors. Also we show how our model can be applied iteratively to check the entire scheduling hierarchy [127].

Besides the changed task model, the analysis method presented in this section considers that all service contracts, either for an application or for a component, are mapped to a set of execution time servers which mark the limits of the resources allocated to the components. This is not a major change, since any scheduling window in a time partition can be seen as a non-preemptable periodic server.

The model used in the approximative schedulability analysis method is based on a set of assumptions:

a1. The characteristics of the tasks (arrival times, periods and execution times) and of the execution time servers (offset, budget and replenishment period) are given as requirements to the scheduling algorithm and do not change them at scheduling time.

a2. All tasks are periodic.

a3. The scheduling algorithm has succeeded only if all task deadlines are met. If a task misses its deadline it is considered a failure.

a4. The only shared resource in the system is the processor.

a5. The speed of the multi-core processor does not change.

a6. A task cannot execute on two or more cores simultaneously, and a core cannot execute two or more tasks simultaneously.

a7. Preemption is permitted at any time but we do not consider any overheads.

a8. Task and server migration is allowed at any time and no overhead is associated with migration.

a9. Components of an application are independent of each other. They share only the time partition supplied to the parent application.

a10. The tasks of each component are grouped in so called Multi-threaded Tasks (MTTs). MTTs are independent, that is MTTs do not share any resource other than the servers supplied to the component and the arrival time of any MTT is not conditioned by the end of another MTT.

a11. The tasks within a MTT can be independent or there can be precedence constraints defined between them. That is, the release of a task may be dependent on the finishing time of another task.

a12. All tasks within a MTT have the same period and deadline.

a13. The execution time of a task in a MTT is defined as a continuous interval between an upper and lower bound.

a14. Tasks have fixed priorities.

4.3.2. System Model

According to the assumptions in the Section 4.3.1. we give new definitions to components and component contracts.

Definition 5 (Component)**.** *A component C consists of a finite set $\mathcal{MT}$ of n MTTs where:*

- *a MTT $\Theta_i \in \mathcal{MT}$, with $1 \leq i \leq n$, is a tuple $\Theta_i = (\mathcal{T}_i, T_i, D_i, R_i)$, where $\mathcal{T}_i$ is the set of t_i tasks in the MTT, T_i represents the inter-arrival time between different instances of the same MTT, D_i is the deadline by which all tasks in $\mathcal{T}_i$ should finish and R_i represents the time of the first release of Θ_i,*
- *each task $\tau_j \in \mathcal{T}_i$, $1 \leq j \leq t_i$, is characterized by a tuple $(bcet_j, wcet_j, prio_j)$ where $bcet_j$ and $wcet_j$ are integer values that specify the limits of the continuous execution interval of task τ_j with $0 \leq bcet_j \leq wcet_j$ and $prio_j$ is the priority of the task.*

All numeric parameters in Definition 5 are considered integer numbers.

The tasks belonging to each component are scheduled separately using a component-specific preemptive scheduling policy. Therefore, when building an application based on such components one must ensure that the tasks of each component are schedulable independent of the execution of any other component in the application.

An execution time server in a multi-core system, as considered here, is characterized by a tuple (Q, P) meaning that the component will receive Q units of execution every P units of time. Additionally, we consider a third parameter O representing the time when the server is first released. It is assumed there is a finite set of servers $\mathcal{S}$ containing the servers for all components of an application.

In terms of the timed automata formalism we define a component contract as follows:

Definition 6 (Component contract)**.** *A component contract $\mathcal{C}_C$ providing a set of n_s execution servers $\mathcal{S}_C \subseteq \mathcal{S}$ is a timed automaton $\mathcal{A}_{\mathcal{C}c}$ over the set of actions $\Sigma_{\mathcal{C}_\mathcal{C}}$ such that:*

- *$\mathcal{A}_{\mathcal{C}c}$ specifies the activation pattern of servers $\sigma_i = (Q_i, P_i, O_i) \in \mathcal{S}_C$, $1 \leq i \leq n_s$,*
- *$\Sigma_{\mathcal{C}_\mathcal{C}} = \{active, inactive\}$, where action $active$ signals to the component scheduler that a server $\sigma_i \in \mathcal{S}_C$ has just become active (i.e. the processor is now available to be used by the component), while $inactive$ signals deactivation of the server.*

We consider that parameters Q_i, P_i and O_i have integer values.

The component also has associated a scheduler which will schedule for execution the tasks of the MTTs in a component according to a preemptive scheduling policy. We consider that a global scheduling policy is used and as such a task can run on any processing unit. As a consequence, task migration may occur whenever a task is preempted or suspended. The scheduler of the component is modeled by a timed automaton with the following characteristics:

- has a queue holding the tasks ready for execution,
- implements a preemptive scheduling policy Sch representing a sorting function for the task queue,
- maintains a map between active execution servers and tasks using the servers, and
- has an Error location which is reached when a task misses its deadline.

A component consisting of n MTTs could have been modeled also using a timed automaton for each of the tasks of the MTTs, each automaton with its own clock. Since the state space of timed automata models grows exponentially with the number of clocks in the model, we decided to build a single timed automaton which models the execution patterns of all n MTT and reduce the number of clocks to one as it will be shown in the following subsection. Moreover, each task could have been modeled using stopwatch automata but the reachability analysis of composed stopwatch automata is undecidable [107]. The same observation applies for modeling the component as a single stopwatch automaton and consequently, we propose an approximation of a stopwatch model using timed automata with discrete clocks to keep track of the execution time of each task.

4.3.3. Timed Automata Model

In the timed automata formalism we consider that a component of a real-time application is the network of timed automata obtained through parallel composition of the automaton which models the execution pattern of the MTTs of the component, the component scheduler automaton and the timed automaton modeling the activation and deactivation patterns of the execution time servers (i.e. the $\mathcal{A}_{Cc}$ automaton in Definition 6). In what follows we will use the names *Task Generator (TGT)* to denote the timed automaton which models the MTTs' execution and *Server Generator (SG)* for the one modeling the servers. Apart from these, the network also includes a *Timer* automaton (see Figure 4.2) which uses a single continuous clock t and each time this clock ticks the automaton sends a *tick* signal to the TGT and the SG automata.

Before explaining in more detail the timed automata model we introduce some notations. For each MTT Θ_i we use a variable $R(i)$ to hold the time of the next release of Θ_i. In order to determine the actual execution time of each task τ_j, we use a variable $E(j)$ to keep track of the time task τ_j has executed since its last release. Basically, $E(j)$ acts like a discrete clock which can be suspended and resumed. Also, for each task τ_j a variable $status(j)$ indicates its current status and is initialized to *idle* meaning that a task instance has not been released yet. The value $status(j) = ready$ is used to denote that a task instance of τ_j is ready for execution (i.e. it has just been released or was preempted). If τ_j is waiting for one of its predecessor tasks to finish then $status(j) = waiting$. If an instance of task τ_j is running then $status(j) = running$. A task τ_j which has executed for $bcet_j$ time units but for less than $wcet_j$ units will have $status(j) = can\_stop$. To denote that an instance of task τ_j has finished or has missed its deadline we use $status(j) = finished$ and $status(j) = overrun$, respectively. The discrete clock $E(j)$ keeps track of the overall time for which $status(j) = running$.

The TGT automaton presented in Figure 4.10 uses a variable $task\_next\_release$ to remember the time until one of its MTTs must be released and the variable $task\_next\_finish$ to keep the earliest time when one of the currently running tasks should finish. At start-up, $task\_next\_release$ is initialized to $min(R(i))$, $i = 1, 2, .., n$ and the automaton goes to the *Ready* location. From this location, if $task\_next\_release = 0$ and there is at least one task τ_j with $status(j) = idle$ (i.e. TGT must release a task), the automaton executes the transition with the guard $must\_release = true$ and in function $get\_ready\_task()$ elects the task τ_j, $j = 1, 2, .., t_i$, for which $status(j) = idle$, updates $task\_next\_release$ (only if this is the first task τ_j in the MTT that is released in the current period) and $task\_next\_finish$, sets a shared variable $task\_ready$ to $(i-1) \cdot n + j$ (the global identifier of task τ_j of the MTT Θ_j) and then sends the $ready$ signal to the scheduler automaton which will read the $task\_ready$ variable and will add task τ_j to its queue. The process is repeated until $task\_next\_release$ becomes greater than 0 and there are no idle tasks for any MTT that has just been released. At this point the automaton goes to the *Idle* location where it waits for a *tick* signal from the *Timer* automaton. On each release of the MTT Θ_i, $R(i)$ is postponed with T_i.

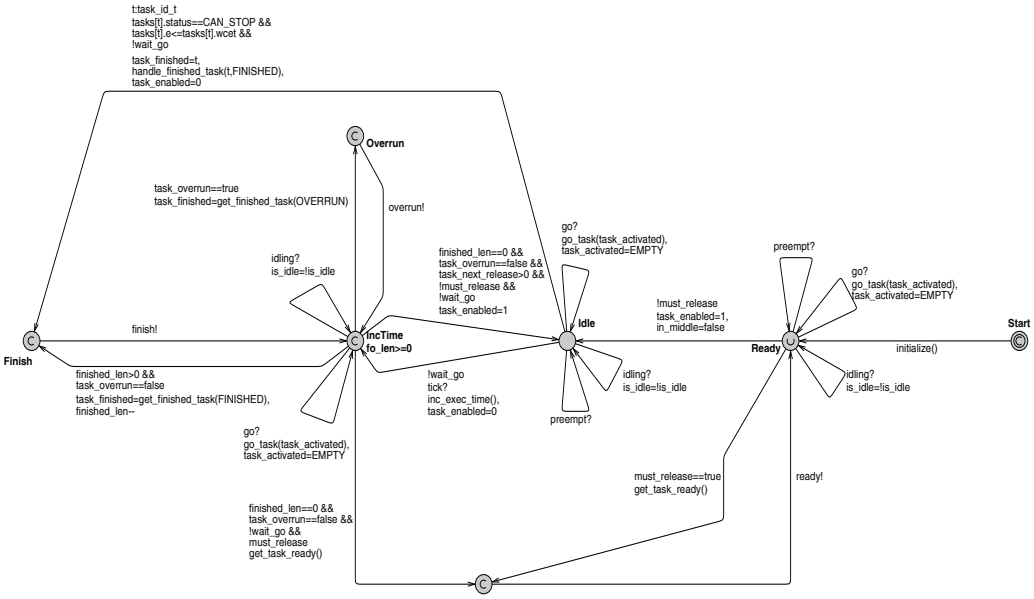


Figure 4.10: The Task Generator (TGT) for MTTs timed automaton

The SG automaton presented in Figure 4.11 works in a similar way with the distinction that generated active servers are continuous (not preemptable). For each server σ_k, $1 \leq k \leq n_s$, there is a discrete clock $RE(k)$ analogous to $E(j)$ and a variable $RR(k)$ is used to keep the time until the next activation of σ_k. Two additional variables, $server\_next\_release$ and $server\_next\_finish$ hold the time until the earliest start time of a server σ_k and the earliest finish time, respectively. When $server\_next\_release = 0$ a processing unit becomes available for the component (i.e. some σ_k starts) and the SG takes the transition guarded with $server\_next\_release = 0$. In function $get\_sever\_ready()$ SG determines the server σ_k which became active, updates $server\_next\_release$ and $server\_next\_finish$ and sets a shared variable $server\_ready$ to k. Afterwards, the $active$

signal is sent to the scheduler of the component to announce the activation of server σ_k. Also, for the server that just started, $RR(k)$ is set to P_k. When σ_k finishes and the processing unit is no longer available, the automaton takes the transition guarded with $server\_next\_finish = 0$ and similar to the previous scenario sets the shared variable $server\_finished = k$ and sends the $inactive$ signal to the scheduler.

On every tick of the timer, TGT leaves the *Idle* location and goes to the *IncTime* location. During this transition, in function $inc\_exec\_time()$, the current execution time $E(j)$ of all tasks τ_j running (with status set to $running$ or can_stop) at that time are increased with a value MIN representing the minimum between $task\_next\_release$, $task\_next\_finish$, $server\_next\_release$ and $server\_next\_finish$. If, as a result of this update, there are tasks for which $E(j)$ reached $bcet_j$ then we set $status(j) = can\_stop$ and if $E(j) = wcet_j$ then the task has finished its execution, $status(j)$ becomes $finished$ and a variable $finished\_len$ counting the finished tasks is incremented. At the same time, we identify any task τ_j that missed its deadline and set $status(j) = overrun$. Also, as time passes the time $R(i)$ of the next release of each MTT Θ_i is decreased with MIN and the values $task\_next\_release$ and $task\_next\_finish$ are updated. When the SG receives the *tick* signal from the *Timer*, in function $update\_times()$, increases the current activation length $RE(k)$ of all active servers σ_k with MIN and decreases $RR(k)$ of all servers with the same value. Also the values of the variables $server\_next\_release$ and $server\_next\_finish$ are updated.

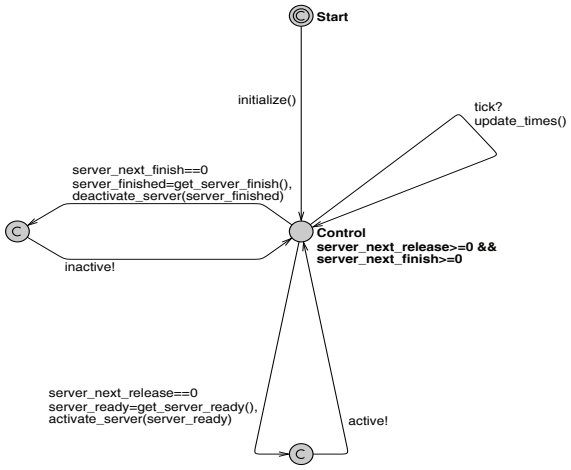


Figure 4.11: The Server Generator (SG) timed automaton

If some $R(i)$ reaches 0 then a new instance of the MTT is released (TGT sends the *ready* signal to the scheduler as explained before). When the scheduler (see Figure 4.12) receives notification of a new task being released, checks if a server on which to schedule the task is available and, if so, sends the go signal to the TGT automaton and sets the entry in its server-task map accordingly. If the priority of the newly released task is higher than the priority of one of the running tasks and no active servers are idle, the scheduler will preempt the lower priority task and will give the server to the higher priority task. If no server is available or the server is deactivated while a task is running on it, the task is either scheduled on another server (if its priority allows it) or is queued. On every tick the TGT automaton searches for all tasks τ_j that finished their execution or that missed their deadline and sends $finish$ or

$overrun$ signals to the scheduler. If the server used by a finished task is still active and there are ready tasks waiting in the scheduler's queue, a new task is started and the go signal is sent to TGT. Moreover, when an active server σ_k finishes, the scheduler will attempt to reschedule the task that was using the server on some other free server available to the component. If no active server is free, then the lowest priority running task may be preempted. Between all automata, data (e.g. task identifier or resource identifier) is transmitted using shared variables.

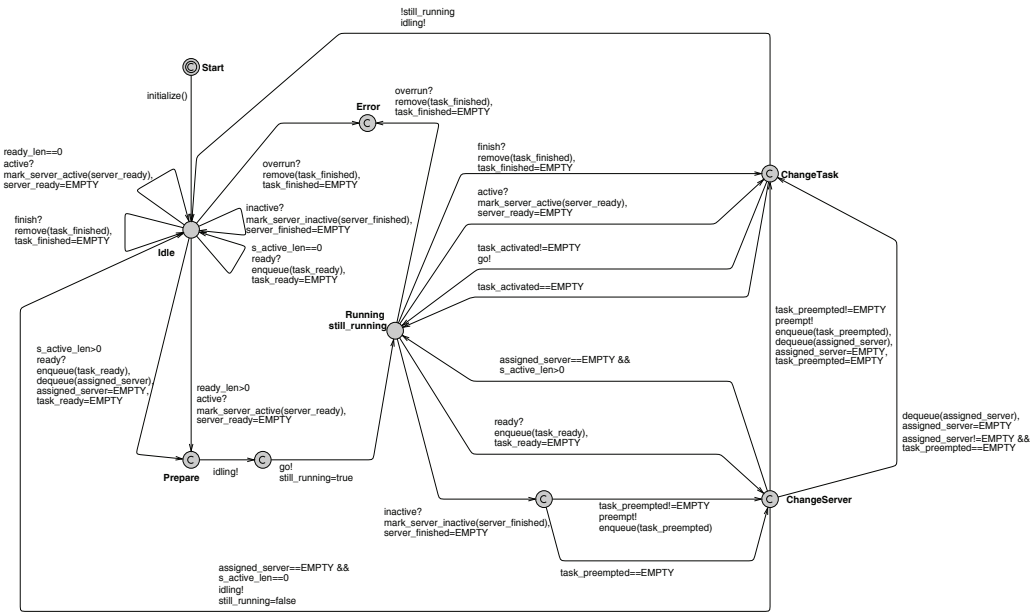


Figure 4.12: Timed automaton model for the scheduler of component composed of several MTTs.

In order to be able to capture task execution intervals in continuous time, when the TGT automaton is in the *Idle* location, if there is at least one task τ_j with $status(j) = can\_stop$, the automaton may decide non-deterministically to finish the task. A remark that must be made is that whenever there is at least one task with status can_stop the value of MIN is set to 1. This implies that at the next *tick* signal, the discrete clocks presented above are increased with a single time unit. If a task τ_j finishes at some fraction of the time unit, another task τ_l that was previously preempted or is ready to be released may take the place of τ_j. However, because in this case we cannot keep track in the discrete clock $E(l)$ of the time task τ_l is executing until the first tick after it has been started/restarted, the value in $E(l)$ is only an approximation of the real execution time of τ_l. Although, this approach represents just an approximation model of the real system, we will show in the next section that the model preserves the properties of the system and any component that is found schedulable with the proposed model is indeed schedulable.

It is important to notice that once each component of an application is proved to be schedulable, by using reachability analysis on our model we can also check the schedulability of the entire application as follows. Each execution time server σ_k is basically a periodic task with hard deadlines and fixed execution requirement Q_k. Therefore it can be considered as an MTT consisting of a single task with $bcet =$

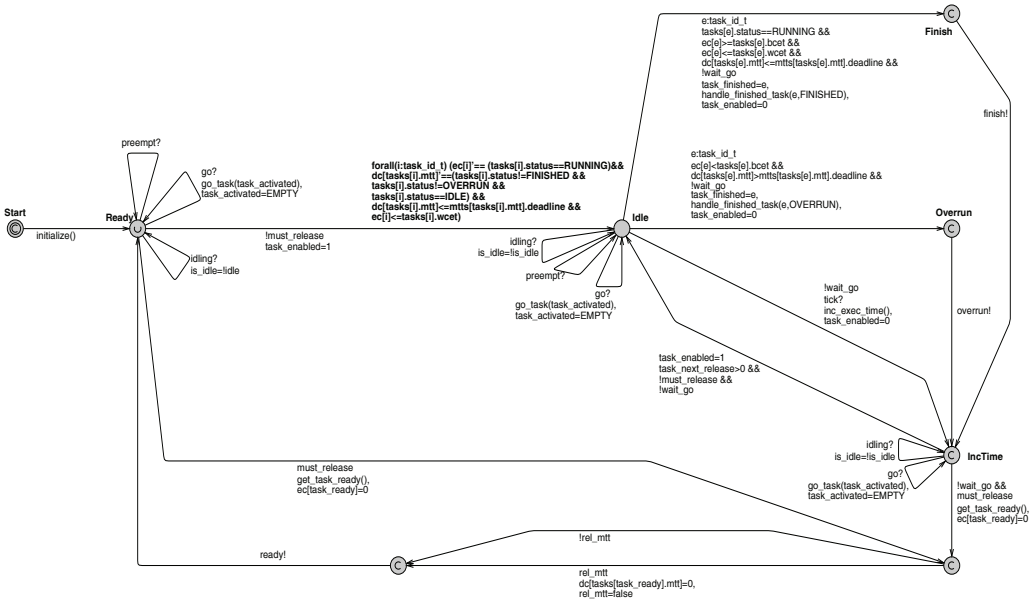


Figure 4.13: Stopwatch automata model of a real-time component.

$wcet$ and the whole application can be seen as just another component with its own scheduler and whose MTTs are the execution servers corresponding to the service contracts of its components. If we consider that the application also has a service contract mapped to another set of execution servers, we can again check the proposed model by changing only the parameters of the MTTs and of the execution servers to reflect the new scheduling level represented by the parent application.

4.3.4. Stopwatch Automata as a Model for Real-Time Components

Stopwatch automata [59] can be defined as timed automata for which clocks can be stopped and later resumed with the same value. These clocks are called stopwatches and provide a simple way for modeling preemptive real-time tasks. Syntactically, a Stopwatch automaton (SWA) is a tuple $\langle L, l_0, \Sigma, \mathcal{C}, I, E, A\rangle$ where $L, l_0, \Sigma, \mathcal{C}, I, E$ have the same meaning as for timed automata (see Section 3.3.) and $A : L \times \mathcal{C} \rightarrow \{0, 1\}$ is a function that defines the rates of clocks $c_i \in \mathcal{C}$ in locations as differential functions $\dot{v}(c_i) = k_i$ where $k_i \in \{0, 1\}$.

From a semantical point of view, the element that distinguishes the SWA from the timed automaton is the clock valuation function $v : \mathcal{C} \rightarrow \mathbb{R}_+$ assigning positive real values to clocks in $\mathcal{C}$. In a SWA the value of a clock variable during a delay transition $(l, v) \xrightarrow{d} (l, v')$ is updated to $v'(c_i) = v(c_i) + A(l, c_i) \cdot d, \forall c_i \in \mathcal{C}$.

In our case, since the tasks belonging to the MTTs of a component are scheduled using a preemptive scheduling policy we could have chosen to model the component using the SWA in Figure 4.13. The execution time of each task in each MTT is represented as a stopwatch clock ec_j, $\forall j \in \{1, 2, ..., n \cdot t_i\}$ with $1 \leq i \leq n$. With each MTT we associate a clock dc_i, $\forall i \in \{1, 2, .., n\}$ which will keep track of the MTT's deadline.

In the stopwatch version of our components we only need to replace the TGT automaton with a Task Generator Stopwatch automaton (TGS). The TGS presen-

ted in Figure 4.13 still uses the variable $task\_next\_release$ to remember the time until one of its MTTs must be released but it is not necessary to keep the variable $task\_next\_finish$. When the system starts, $task\_next\_release$ is initialized to $min(R(i))$, $i = 1, 2, .., n$ and the automaton goes to the *Ready* location. From this location, it can either go the *Idle* location if there are no tasks ready for release or, if there is at least one task τ_j with $status(j) = idle$, the automaton executes the transition with the guard $must\_release = true$ and, in function $get\_ready\_task()$ elects the task τ_j, $j = 1, 2, .., t_i$, which is idle, updates $task\_next\_release$, puts the task global identifier $(i-1) \cdot n + j$ in the shared variable $task\_ready$ and sends the $ready$ signal to the scheduler automaton. The scheduler will read the $task\_ready$ variable and will add task τ_j to its queue. On each release of the MTT Θ_i, $R(i)$ is postponed with T_i. After all tasks that are ready to start are released, the stopwatch automaton goes to the *Idle* location where it waits for a *tick* signal from the *Timer* automaton or for a running task τ_j to finish. Another event which may take the automaton out of the *Idle* location is a missed deadline of any MTT. The rates of the clocks ec_j and dc_i are specified in the guard of the *Idle* location: $dc_i' = 1$ for all MTTs which contain at least one task that is not finished yet, otherwise $dc_i' = 0$ and $ec_j' = 1$ for all tasks that have $status(j) = running$ but $ec_j' = 0$ for the other tasks.

In the stopwatch automaton the execution time of each task τ_j is measured by stopwatch ec_j started at the release of the task, when the automaton sends the *ready* signal to the scheduler automaton while the variable $task\_ready = j$, until the task finishes and the *finish* signal is sent with variable $task\_finished = j$. ec_j does not include the time while the task was preempted. Therefore, for any task τ_j belonging to the multi-threaded task Θ_i the following constraints should be true such that we can say that τ_j has not missed its deadline:

$$0 \leq ec_j \leq wcet_j,\ 0 \leq ec_j \leq dc_i \tag{4.1}$$

Definition 7. *A multi-threaded task $\Theta_i = (\mathcal{T}_i, T_i, D_i, R_i)$ is schedulable iff all its tasks $\tau_j = (bcet_j, wcet_j, prio_j) \in \mathcal{T}_i$ finish execution before the deadline of Θ_i: $dc_i \leq D_i$ when $ec_j = wcet_j$, $\forall j \in \{1, 2, ..., t_i\}$.*

Definition 8. *A component is schedulable iff all its multi-threaded tasks are schedulable.*

The set of actions of the TGS is $\Sigma = \{ready, go, preempt, finish, overrun, idling, tick\}$. The *idling* signal is sent by the scheduler automaton when goes in or out of the *Idle* location. TGS stays in the *Idle* location as long as either there is no servers active or there are no ready tasks to be scheduled or both of these conditions are true. The *go* and *preempt* signals are controlled also by the scheduler. The TGS will send *ready* for every new release of a task instance and *finish* at its end.

A timed word over the alphabet Σ is a pair (ρ, θ) where $\rho = \rho_1, \rho_2, ..$ is an infinite sequence of events in Σ and $\theta = \theta_1, \theta_2, ..$ is a timed sequence denoting the timestamps of the events in ρ. A timed language over Σ is a set of timed words over Σ. The timed language $L(S)$ accepted by the stopwatch automaton is the union of the timed languages $L_j(S)$ where the words in each language $L_j(S)$ refer to valid event sequences generated during the execution of task τ_j. We consider that $L(S) = \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq t_i} L_j(S)$ because the semantics of task related events in Σ are established only in correspondence with a shared variable indicating the task to which the event refers. The untimed words in all $L_j(S)$, and consequently in $L(S)$, are described by the following regular expression:

$$E_S = (ready, go, (preempt, go)^*, finish) \tag{4.2}$$

In our case the timestamps of all events $\{ready,go,preempt,finish\}$ acceptable by the TGS have to be less than the deadline of the MTT containing the task for which the event appeared (i.e. the task is indicated in a shared variable). This implies that the time a task τ_j in Θ_i spends in the $ready$ state, denoted from now as T_{ready_j}, must be lower or equal than $D_i - wcet_j - T_{wait_j}$, where T_{wait_j} is the time the task has to wait for its predecessors to finish.

4.3.5. Approximation of Components using Timed Automata

In this section we show what are the approximation errors implied by the proposed method for schedulability analysis of real-time component-based applications. We also prove that any component declared schedulable by our method it would also be declared schedulable by the stopwatch model. We do this by showing that the language $L(T)$ of TGT is also accepted by TGS.

The alphabet of the TGT is similar to the one of the stopwatch automaton. The *tick* and *idling* events in Σ are not directly related to the execution of the task and only help in modeling the discrete time. These two signals are kept in TGS since the activation time of the servers is still measured with discrete clocks (i.e. we see them as non-preemptable tasks with integer parameters). Keeping the discrete clocks also in TGS, instead of replacing them with real clocks, has no influence on the accuracy of the model and on the schedulability analysis. Therefore, in what follows we will refer mostly to the task-related events in Σ: $ready, go, preempt, finish$ ($overrun$ means the system is not schedulable and, as we analyze the conditions under which the system is schedulable, we assume this event does not appear).

For the proposed discrete time model we have chosen to consider a time unit equal to 1. The approximation errors in our model arise from the following situations:
(1) a task τ_j starts its execution at some subdivision of the time unit and
(2) a task τ_j resumes its execution (after it has been preempted) at some subdivision of the time unit.
Since we cannot measure the time from the start/restart point of τ_j until the beginning of the next time unit, the subunit of execution time will not be reflected by $E(j)$.

A task τ_j can be preempted either when a higher priority task τ_l becomes ready for execution or the server used by task τ_j is deactivated and there is no other active and free server. Activation and deactivation of servers is observed by TGT only through task preemptions and resumptions. As for all MTTs the release time is an integer value, a higher priority task τ_l can be released for execution while τ_j is executing only at discrete moments. If τ_l does not depend on any task it means that, in this case, τ_l can preempt τ_j only at discrete moments of time and the clocks $E(j)$ and $E(l)$ will behave just like the clocks ec_j and ec_l. If τ_l depends on some task τ_k and τ_k finishes at some time between two consecutive discrete moments then this makes it possible for τ_l to preempt τ_j. In the later case, the clocks $E(j)$ and $E(l)$ will not be increased at the next discrete time point. If task τ_j resumes is execution during the same fraction of time in which it was preempted, clock $E(j)$ will still not be increased as in this case τ_j's role is similar to that of τ_l in the previous case. This implies that no matter how many times the task is preempted between two successive discrete time points, in the computation of $E(j)$ this will have the same effect as a single preemption. Since all tasks of an MTT Θ_i have the same deadline D_i and are released at the same time then for all tasks in Θ_i at most m_j preemptions may influence the value of the discrete clock $E(*)$ without a deadline miss occurring, where

$$1 \leq m_j \leq D_i - T_{wait_j}, m_j \in \mathbb{N} \tag{4.3}$$

The alphabet of TGT is the alphabet of the TGS, namely $\Sigma = \{ready,go,preempt, finish,overrun,idling,tick\}$. Just like for the stopwatch automaton, in this case also we are interested only in the events in Σ related to task execution. Therefore, for each task τ_j, the timed automata has to accept timed words following the syntax of the untimed regular expression:

$$E_T = (ready, go, (preempt, go)^*, finish) \tag{4.4}$$

We see that $E_T = E_S$.

In what follows we use v_{sw_j} to denote the valuation of the stopwatch clock ec_j in TGS and v_t for the valuation of the continuous clock in TGT, where $v_t \in [0, 1]$. We also consider a valuation $v_j = E(j) + v_t$ for each task τ_j. This helps in measuring the approximation error of the proposed model. Note that v_j is an approximation of v_{sw_j}. As at most m_j preemptions of a task τ_j can influence the value of $E(j)$ it results that $v_{sw_j} - v_j \leq m_j$. Also, we use v_{dc_i} to denote the valuation of the clock dc_i in TGS and we consider the valuation $v_{\Theta_i} = T_i - R(i) + v_t$ which measures the time since the release of the MTT Θ_i. The valuation v_{Θ_i} will grow with the same slope as the valuation v_{dc_i} and consequently $v_{\Theta_i} = v_{dc_i}$ at any time between the release of Θ_i and its finish. Also, we consider $v_{\tau_j} = T_i - R(i) - T_{wait_j} + v_t$ which measures the time since the start of task τ_j. At all time instants between the start of τ_j until its deadline $v_{\tau_j} \leq v_{\Theta_i}$ and both v_{τ_j} and v_{Θ_i} grow with unitary slope.

In order to establish the relationship between TGT and TGS, we must compare the timed words that follow the syntax of E_T and E_S. We assume the timestamps of these words analyzed in relation to the valuations v_{Θ_i} for TGT and v_{dc_i} for TGS, respectively, are the same.

Theorem 1. *For any timed word that follows the syntax of E_T and E_S and simulates the execution trace of a task τ_j on both TGT and TGS automata, $v_{sw_j} - m_j \leq v_j \leq v_{sw_j}$ holds from the release of the task until its ending, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof. Both TGS and TGT receive events related to a specific task in the same order and with the same timestamps (related to v_{Θ_i} and v_{dc_i}, respectively). Whenever the TGT receives a *go* signal it sets the status of the task to *running* and when it receives the *preempt* signal the status of the task is set to *ready*. The behavior of the TGS upon receiving the *go* and *preempt* events is the same.

When the status of the task is *running*, v_{sw_j} grows with slope 1 and stays constant when the task has status *ready*. The valuation v_j also stays constant while the task is in the *ready* state and grows with slope 1 when the task is running due to the v_t component. If the task is preempted only at discrete time points (e.g. when the server that the task was using finished its available execution units) then at the end of the task $v_j = v_{sw_j}$. However, if a task preemption happens between two successive distinct time points, the valuation v_t is not added to v_j, whereas v_{sw_j} will contain also this fraction of time unit, and therefore $v_j \leq v_{sw_j}$. Since the v_j can stay constant for at most m_j times, each time by not being increased with at most 1 time unit, it follows that $v_{sw_j} - m_j \leq v_j$. □

The inequality in Theorem 1 shows that during the simulation of the same word on both the TGT and TGS automata, it will take at least the same amount of time for v_j to reach a specific value as it will take to v_{sw_j}, but it could also take longer.

Next we analyze possible timestamps of the *finish* event and show that all tasks that finish before their deadlines in TGT also finish before their deadlines in TGS. As this event is related to the guards in the automata that contain the best

case execution time $bcet_j$ of a task τ_j and its worst case execution time $wcet_j$ we will determine what is the relation between the actual best execution time t_{bcet_j} (the value of v_{τ_j} when $v_j = bcet_j$) and the actual worst execution time t_{wcet_j} (the value of v_{τ_j} when $v_j = wcet_j$) and the valuations $v_j = bcet_j$ and $v_j = wcet_j$. Note that v_j does not include those fractions of time units that we cannot measure in the TGT.

Theorem 2. *For any timed word that follows the syntax of E_T and simulates the execution trace of a task τ_j on TGT, if $v_j = bcet_j$ then $t_{bcet_j} \geq bcet_j$, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof. If task τ_j is not preempted during its execution then $t_{bcet_j} = bcet_j$. If τ_j is preempted only at discrete time points then $0 < t_{bcet_j} - bcet_j \leq T_{ready_j} + m_j$. If the task is preempted between two consecutive discrete time points then the valuation v_j when the task will resume its execution will not contain the subunit of time that it had executed before it was preempted and only an integer number of time units. In contrast, t_{bcet_j} will contain those time fractions and the preemption time and, therefore it will reach $bcet_j$ faster than v_j which means that $t_{bcet_j} > bcet_j$. □

Theorem 3. *For any timed word that follows the syntax of E_T and simulates the execution trace of a task τ_j on TGT, if $v_j = wcet_j$ then $t_{wcet_j} \geq wcet_j$, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof. If task τ_j is not preempted during its execution then $t_{wcet_j} = wcet_j$. If τ_j is preempted only at discrete time points then $0 < t_{wcet_j} - wcet_j \leq T_{ready_j} + m_j$. If the task is preempted between two consecutive discrete time points then the valuation v_j when the task will resume its execution will not contain the subunit of time that it had executed before it was preempted and only an integer number of time units. In contrast, t_{wcet} will contain those time fractions and the preemption time and, therefore by the time v_j will reach $wcet_j$ t_{wcet_j} will be greater than $wcet_j$. □

From theorems 2 and 3 it follows that, if a task finishes its execution before its deadline in TGT ($t_{wcet_j} \leq D_i$ or $t_{bcet_j} \leq D_i$), it will always meet its deadline in TGS (i.e. in the TGT the task executes for a longer time that in TGS and is still schedulable) which means that if the task is proven schedulable in TGT then it will also be schedulable in TGS.

For a task τ_j belonging to a MTT Θ_i to be schedulable it is required for it to finish before its deadline. If T_{ready_j} is the time for which the task has the *ready* status then we say that the task is schedulable if the following condition is satisfied:

$$T_{ready_j} + m_j \leq D_i - wcet_j - T_{wait_j} \tag{4.5}$$

The condition 4.5 says that task τ_j can be preempted for at most $D_i - wcet_j - T_{wait_j} - m_j$ before its deadline. The m_j term appears due to the imprecision of the model.

Next we prove that TGS accepts the timed language over Σ that TGT accepts. Specifically we prove that $L(T) \subseteq L(S)$ by checking the intersection $L(T) \bigcap \overline{L(S)} = \emptyset$. We have already shown that the syntax of the timed language $L(S) = \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq t_i} L_j(S)$, where $L_j(S)$ is the language with words referring to valid event sequences generated during the execution of task τ_j. By analogy, $L(T) = \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq t_i} L_j(T)$. Therefore, if $L_j(T) \subseteq L_j(S)$, $\forall 1 \leq j \leq t_i$ and $\forall 1 \leq i \leq n$ then also $L(T) \subseteq L(S)$. With this objective we prove that $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$, $\forall 1 \leq j \leq t_i$ and $\forall 1 \leq i \leq n$. For a task τ_j to be schedulable all words in $L_j(T)$ must satisfy condition 4.5. Similarly, all words in $L_j(S)$ must satisfy the condition $T_{ready_j} \leq D_i - wcet_j - T_{sw\_wait_j}$, where $T_{sw\_wait_j}$ is the task

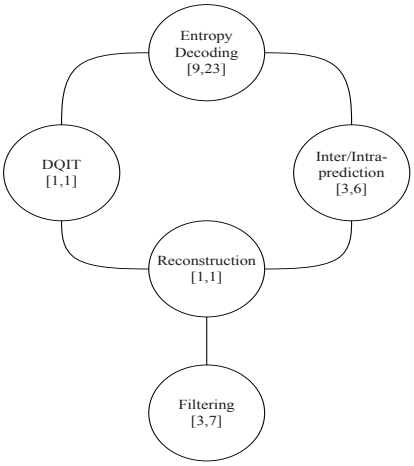


Figure 4.14: The MTT for slice decoding.

waiting time in TGS and $T_{sw\_wait_j} \leq T_{wait_j}$. Then the condition $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$ becomes $(T_{ready_j} + m_j \leq D_i - wcet_j - T_{wait_j}$ and $T_{ready_j} > D_i - wcet_j - T_{sw\_wait_j})$. As from condition 4.3 we know that $m_j \geq 1$ and T_{ready_j} cannot be at the same time higher and smaller than a value, it follows $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$ is true for any task τ_j and, therefore $L(T) \bigcap \overline{L(S)} = \emptyset$ holds.

4.3.6. Case Study for the H.264 Decoder

Using the proposed method for schedulability analysis, in this section we present a series of experiments in which we apply the proposed schedulability method to analyze the multimedia H.264 decoder [2]. The decoder is modeled as a component.

We use the H.264 decoder for decoding 176×144 square pixel Quarter Common Intermediate Format (QCIF) video intended for portable multimedia devices. In the encoded video, each video frame can be divided in slices and each slice can be further split in blocks of 16×16 pixels, called macro-blocks (MB). The decoding process can be applied to several slices in parallel and consists of several steps which can be mapped to the tasks of a MTT, one MTT for each slice. In the first stage of the process, numerical values are recovered from the binary codes of the compressed video (Entropy Decoding). Since a part of the data in the encoded video was computed through prediction, in the next stage (Dequantization and Inverse Transform DQIT) the differences between the predicted data and the real data are recovered. Next, in the motion compensation (or Inter-prediction) or Intra-prediction stage, each MB in the frame is decoded based on predicted data from previous frames or other MBs in the frame. Finally, the MBs of each slice are put together (Reconstruction) and a filtering stage is applied to improve quality of the decoded slice. Correspondingly, each MTT will have five tasks, one for each of the above stages. The MTT corresponding to the decoding process is presented in Figure 4.14. We consider that for each frame the same MTT will process the same slice.

To see how we can apply our method on the H.264 decoder, we extracted the execution parameters of the tasks in a MTT through profiling of the FFmpeg [1] encoder using three real video files (obtained from [4]) with increasing level of spatial

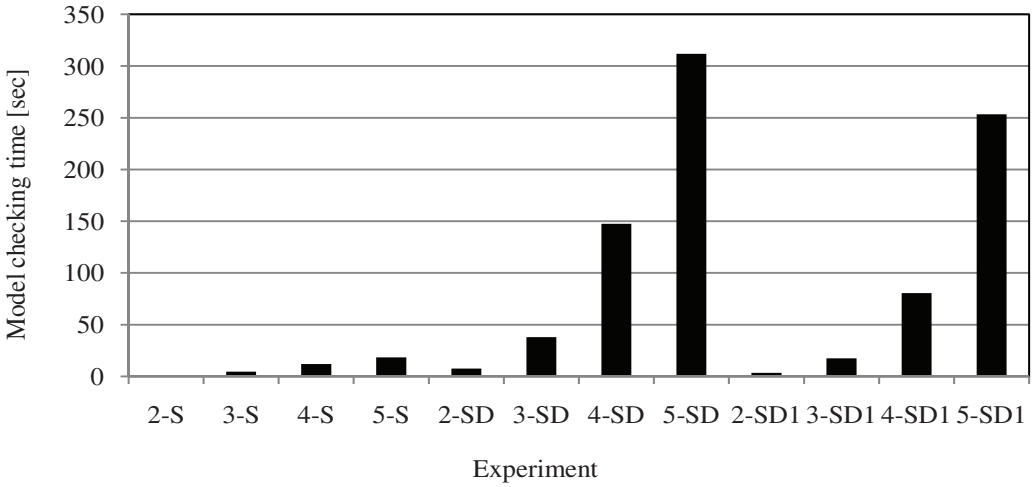


Figure 4.15: Model checking time for H.264 experiments

details and amount of movement:

- *Akiyo* - characterized by *low* spatial detail and *low* amount of movement,
- *Foreman* - characterized by *medium* spatial detail and *low* amount of movement,
- *Mobile* - characterized by *high* spatial detail and *medium* amount of movement.

After profiling we have obtained the $[bcet, wcet]$ intervals shown in Figure 4.14. The execution time for the Entropy Decoding stage depends on the input data and hence its large variation.

We have checked the schedulability of the component model using the UPPAAL model checker by issuing the verification of the *A[] not Error* property (i.e. the Error location is never reached). We ran experiments considering that for each frame of the video 2 (experiment 2-S), 3 (3-S), 4 (4-S) or 5 (5-S) slices are processed in parallel, where for each slice we have a MTT. The number of actual tasks in each experiment is five times the number of slices used for the experiment. First, for each task in a MTT, we have considered the $[bcet, wcet]$ intervals shown in Figure 4.14 and for the period and deadline of the MTT we have chosen the value 100. Since we noticed that the difference between the $wcet$ and the $bcet$ of each task and between execution times of different tasks influences the scalability of our method (the maximum number of supported MTTs), we then ran experiments in which all these time values were doubled (experiments 2-SD, 3-SD, 4-SD, 5-SD). In the last series of experiments (2-SD1, 3-SD1, 4-SD1, 5-SD1) we doubled only the execution time of the first task in the MTTs. We have chosen only this task since it has the greatest execution time, which varies along the largest interval. For all experiments we have used a set of three execution servers, one on each processor, with periods equal to 50 and a total processor utilization of 2.0 meaning that every 50 time units the servers provide 100 execution units to the decoder. The experiments were executed on a machine with Intel Core 2 Quad 2.40 GHz processor and 4 GB RAM running Ubuntu. The model checking time of all experiments is presented in Figure 4.15.

We can see in Figure 4.15 that although the number of tasks gets up to 25 the model checking time is rather small (maximum 310 seconds). Also it can be seen that the complexity is a factor of the number of tasks, but is influenced also by the difference between the best and worst case execution time. Also, from the last

four experiments (2-SD1 - 5-SD1) it can be observed that by increasing the difference between parameters of different tasks, the model checking time also grows even more than when we double all task parameters. We believe this is due to the fact that the change in these experiments has increased the non-determinism of the model.

4.4. Concluding Remarks

This chapter presents the proposed schedulability analysis method for multi-core contract-based scheduling, which assumes that the real-time system runs a set of component-based applications, each with its own timing constraints. We have shown in the first section that the utilization-bound methods used in classical real-time scheduling give only pessimistic and sufficient tests for deciding whether a system using service contracts is schedulable or not.

The proposed method uses the timed automata formalism and transforms the schedulability problem into a reachability one solved through model checking. The timed automata model of the system employs discrete time semantics. The contributions of this chapter are focused on the following areas:

c1. We formalize the problem of deciding the schedulability of multi-core contract-based systems [60] utilizing timed automata.

c2. We introduce model checking as a technique for analyzing the global preemptive multi-core schedulability of an hierarchy of system components consisting of independent tasks with fixed execution time, each with its own timing constraints. Unlike the analytical schedulability method described in [60], the method proposed here is exact and can use any scheduling policy, not just fixed priority policies.

c3. We extended the schedulability method proposed for independent tasks (see c2.) to a more realistic system, where the components can consist of tasks with precedence constraints between them and can have variable execution time. For this model we give a method for schedulability checking using timed automata that approximates the behavior of a stopwatch automata. The approximation is conservative, in the sense that any component deemed schedulable by our method would also be schedulable in the stopwatch automata model.

5. MODEL CHECKING FOR MULTI-CORE TIME PARTITIONS DESIGN

5.1. Problem Formulation

Contract-based scheduling is tightly connected with resource temporal partitioning according to which a resource (e.g. the processor) is shared by several components, each composed of a set of schedulable entities (i.e. tasks). In this context, two problems must be solved:
(1) making real-time guarantees to components running in a temporal partition, and
(2) designing the temporal partition that fulfills the timing constraints of a component.

In the previous chapter we have used model checking to derive real-time guarantees for the components running within a processing resource partition (i.e. time partition or execution time server). Conversely, in this chapter we propose a model checking method for designing a multi-core time partition for a given system component so that it fulfils the component requirements with the minimum resource utilization.

Generation of time partitions is an important problem. Such partitions are used in critical systems like avionics. For example, ARINC-563 [23] – a specification used in digital avionics domain – defines a software interface between applications and the operating system of the avionics computer where each application is assigned a distinct time partition. The operating system supports a two-level hierarchical scheduling framework: a system-level scheduler which schedules the time partitions and application-level schedulers within those partitions. Recently, an extension of the time partitions for multi-core systems has been defined as part of the PikeOS operating system [104]. However, currently time partitions are designed manually. This is not only time consuming but, more important, can lead to over approximations of the application's resource requirements and under utilization of processing resources.

In this chapter we use the model described in Section 4.2. to develop a method for generating multi-core time partitions for a two level hierarchical scheduling system based on the parameters of the components that will execute in those time partitions. The block scheme of this hierarchy is depicted in Figure 5.1: the tasks of each system component are assigned to a different multi-core time partition and are scheduled on it using a component-level scheduler. Further, all time partitions are scheduled on the multi-core platform using a system-level scheduler.

Although, the proposed method uses the model in the previous chapter (see Section 4.2.), we impose a few restricting assumptions on it:

a1. The characteristics of the tasks (arrival times, periods and execution times) are given as requirements to the scheduling algorithm and do not change at scheduling time.

a2. All tasks are periodic and independent of each other. They share only the time partition supplied to the parent component.

a3. System components are independent of each other.

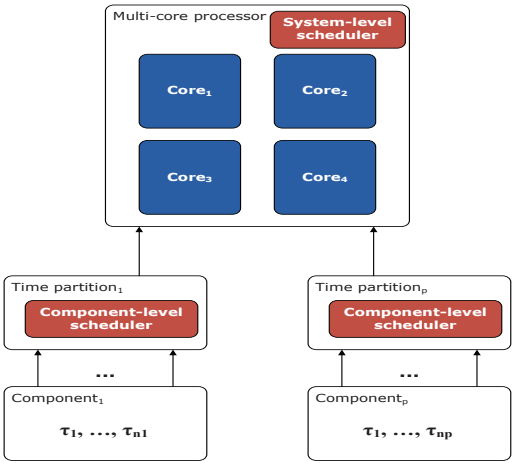


Figure 5.1: A two-level scheduling hierarchy with multi-core time partitions

a4. The scheduling algorithm has succeeded only if all task deadlines are met. If a task misses its deadline it is considered a failure.

a5. The only shared resource in the system is the processor.

a6. The speed of the multi-core processor does not change.

a7. A task cannot execute on two or more cores simultaneously, and a core cannot execute two or more tasks simultaneously.

a8. Preemption is permitted at any time but we do not consider any overheads.

a9. Task migration is allowed at any time and no overhead is associated with migration.

a10. The priorities of the tasks are fixed.

a11. The execution time of a task is a constant and does not variate within an upper and lower bound.

a12. The resource partitions allocated to a component are not preemptable.

5.2. Temporal Partition Design

In the recent years the real-time research community gave considerable attention to hierarchical scheduling. Mok et al. [136] introduced the concept of resource model for characterizing the resource supply provided by a parent component to one of its child components. This resource model enables child components to have different schedulers and allows independent schedulability analysis for each component. Several resource models have been considered in the context of uniprocessor systems (e.g. bounded-delay resource model [136] or the periodic resource model [153]). As shown in [77] simply replicating these resource models to multiprocessor systems is inflexible and leads to wasted processor utilization.

Designing a resource model for a given application so that it fulfils the application requirements with the least resource utilization is a problem that until now has been tackled only for uniprocesor systems. Almeida and Pedreiras [11] introduced a technique for computing the parameters of an execution time server from the task parameters using the worst case response time of each task. Automatic scheduling of time partitions for an ARINC-653 platform has been the focus of the work in [76]. Here, they model ARINC-653 as a two-level hierarchical system, and develop compositional analysis techniques for it. A merit of the work in [76] is the fact that preemption and migration costs are considered. However they compute the time partitions as simple component interface, characterized by a computational request and a period. Based on this, temporal partitions is transformed into tasks with offset zero, scheduled based on a fixed-priority policy.

Easwaran et al. [77] introduced a MPR model where the contribution of each processor to a component resource supply is not the same. Their work addresses a multi-level containment hierarchy where components have hard real-time timing requirements and consist of sporadic tasks. They provide a sufficient condition for component schedulability under the global EDF scheduling policy and the MPR model based on resource supply bound function and component workload. We also use a periodic resource model but unlike the work in [77] we test schedulability of a component using reachability checking of the timed automata model of the system which is not altered by the pessimism of a workload-based condition. Moreover they introduce a technique for transforming a MPR model into a set of periodic tasks which are scheduled by the parent component, but the examples presented in the paper show that the difference between the processor utilization of these tasks and the resource requirements of the child component grows with the period of the resource. In [116] the problem of scheduling multi-level component hierarchies on multiprocessor resource partitions is also considered but is mostly restricted to soft real-time sporadic tasks and assumes that only a small number of tasks have hard timing constraints.

5.3. System Model

A multi-core time partition is implemented as a set of fixed Major Time Frames (MTFs), one for each core. The length of the MTF on all cores must be the same and the frames must be synchronized (i.e. they all start simultaneously). Each MTF consists of several scheduling windows. A scheduling window is defined by an offset relative to the beginning of the MTF and a length. The length of the major time frame identifies the frequency with which its scheduling window execute, i.e., it represents the period of the time partition. Therefore all scheduling windows are periodic. It is also possible for the period of a scheduling window to be chosen from one of the divisors of the length of the MTF. The component associated with a time partition gains access to the processor whenever one of the scheduling windows becomes active and is preempted when the window terminates [104]. As it is possible for two or more active scheduling windows in different MTFs to overlap, the partition allows concurrent execution of two or more tasks. From this specification we derive next a formal definition of the multi-core time partition.

Definition 9. *A multi-core time partition mTP is described by a tuple $\langle \mathcal{SW}, P \rangle$ where:*

- *P is the length of the Major Time Frame (MTF) or the period of the time partition and $\mathcal{SW}$ is the set of scheduling windows sw_k during which the cores are available to a system component,*

- *each scheduling window $sw_k \in \mathcal{SW}$ is a tuple (c_k, o_k, l_k, t_k) where c_k indicates the core which is available, o_k defines the offset of sw_k to the beginning of the MTF, l_k specifies the length of the window, while t_k gives its period with the restriction that the period is an exact divisor of P.*

Example 5. *Figure 5.2 shows two time partitions $mTP_1 = \langle \mathcal{SW}_1, 100 \rangle$ and $mTP_2 = \langle \mathcal{SW}_2, 100 \rangle$ scheduled on a dual-core processor. mTP_1 contains three scheduling windows, $\mathcal{SW}_1 = \{SW_1, SW_2, SW_4\}$, where $SW_1 = (1, 0, 10, 100)$, $SW_2 = (1, 50, 20, 100)$ and $SW_4 = (2, 0, 25, 100)$. mTP_2 also contains three scheduling windows, $\mathcal{SW}_2 = \{SW_3, SW_5, SW_6\}$, where $SW_3 = (1, 25, 25, 100)$, $SW_5 = (2, 40, 20, 100)$ and $SW_6 = (2, 80, 20, 100)$. It can be seen that all scheduling windows are periodic with period equal to 100 (i.e. the length of MTF). Note that both partitions contain overlapping scheduling windows allocated to different cores which means that different tasks of the corresponding components can run in parallel.*

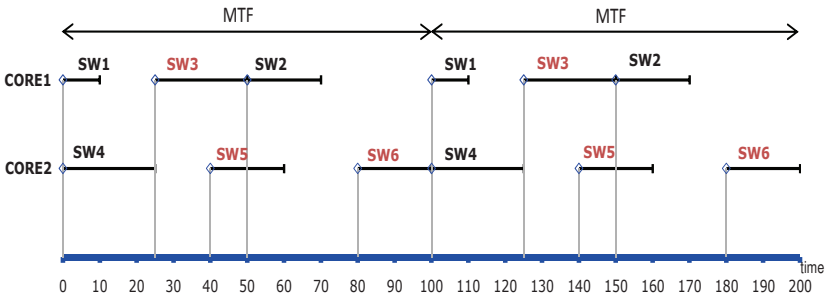


Figure 5.2: An example of two time partitions scheduled on a dual-core processor

The method proposed in this thesis for designing mTP uses model checking on the timed automata model of a scheduling level introduced in Section 4.2.. We remind that the model consists of three timed automata: one for the component (see Figure 4.3), one for the component-level scheduler (see Figure 4.5) and one for the resource level (see Figure 4.4). Therefore it is necessary to associate the time partition with a timed automaton which models the activation of the scheduling windows.

Definition 10. *In the timed automata formalism a multi-core time partition model $mTPM$ is a pair $\langle mTP, \mathcal{A}_{mTPM} \rangle$ where:*

- *mTP is the multi-core time partition defined previously, and*
- *$\mathcal{A}_{mTPM}$ is a timed automaton over the action set $\Sigma_{\mathcal{SW}}$ modeling the activation scheme of the scheduling windows in the time partition where:*
  - *$\Sigma_{\mathcal{SW}} = \{active, inactive\}$,*
  - *action $active$ signals to the component scheduler that a processor is now available to the component, while $inactive$ signals that a processor can no longer be used to schedule the tasks of the component.*

To satisfy the time demands of a component, the time partition must supply sufficient computational resources. For example, time partition mTP_1 in Example 5 provides 55 units of resource every 100 units of time and the ratio $\frac{55}{100}$ represents the resource bandwidth or average processor supply provided over time.

Furthermore, we assume that the components hosted by these multi-core time partitions consist of a finite set of n tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where each task

$\tau_i \in \mathcal{T}$ is specified as a tuple $\tau_i = (C_i, T_i, O_i, D_i)$, with C_i being the worst case execution time of the task, T_i the inter-arrival time between different instances of the same task, O_i the first release of the task and D_i is the deadline of the task where $C_i \leq D_i \leq T_i$ (i.e. tasks have constrained deadlines). The tasks of the component are scheduled on the associated time partition using a fixed priority global multi-processor scheduler with preemption enabled. Without loss of generality, we assume tasks in $\mathcal{T}$ are ordered in decreasing order of their priority.

5.4. Multi-core Time Partition Generation

In this section we propose a model-checking based technique for generating the multi-core time partition that provides the least resource supply satisfying the demands of a component consisting of a set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ of hard real-time periodic tasks scheduled under a global fixed-priority scheduling policy. More specifically, we determine the multi-core time partition $mTP = \langle \mathcal{SW}, P \rangle$ of the $mTPM$ model defined in Section 5.3.

Our generation method is based on the diagnostic traces given by the UPPAAL model checker when verifying a temporal logical property on the timed automata network composed of the three automata composing a scheduling level in the hierarchy of schedulers. A trace is a sequence of states and transitions representing a possible execution path of the modeled system. UPPAAL can generate such traces either by using the simulation feature or through its model checker. The model checker generates a diagnostic trace witnessing a submitted property or a counterexample trace if the checked property is not satisfied.

The proposed method uses both kind of traces. This leads us to a problem as we have to simulate the model for a specified amount a time. In a previous chapter we introduced the notion of *feasibility interval*: a finite interval such that if all tasks in a taskset meet their deadline in this interval then they will always meet their deadline (no deadline miss will ever occur) as, after that, the schedule is periodic. Therefore, we could use the feasibility interval also as a simulation interval. This feasibility interval is computed as a function of the parameters in the taskset and of the scheduling policy. For single processor, it has been proved that such a feasibility interval exists for all task models and scheduling policies. However, in global multiprocessor real-time scheduling, this interval has been computed only for independent periodic tasksets scheduled according to a fixed-priority policy [66, 67]. However, until now, the problem has not been tackled for multiprocessor hierarchical systems. In the next section, starting from the results in [66] we prove that a feasibility interval can also be computed for hierarchical scheduling with fixed-priority component-level schedulers.

5.4.1. Periodicity of Hierarchical Scheduling Algorithms

A taskset $\mathcal{T} = (\tau_1, .., \tau_n)$ is *feasible* with respect to a multiprocessor platform π if there exists at least one scheduling algorithm which can schedule all possible job sequences generated by the tasks on the system and no job misses its deadline.

Definition 11. *[66] If A is an algorithm which schedules $\mathcal{T}$ upon a multiprocessor platform to meet the deadlines, then the system $\mathcal{T}$ is said to be A-feasible.*

It has been proved in [66] that any feasible schedule of a set of asynchronous constrained deadline periodic tasks using a fixed-priority preemptive scheduling policy

on a uniform multiprocessor platform is periodic from a specific point in time. According to [66] the feasibility interval is $[0, S_n + LCM(p_1, p_2, ..., p_n))$ where S_i is defined inductively as follows:

$$S_i = \begin{cases} O_i, & \text{if } i = 1, \\ max\{O_i, O_i + \lceil \frac{S_{i-1} - O_i}{T_i} \rceil T_i\}, & \forall i \in \{2, ..., n\}. \end{cases} \tag{5.1}$$

This result is presented in [66] as the following theorem.

Theorem 4. *[66] For any preemptive fixed-priority algorithm A, if an asynchronous constrained deadline system $\mathcal{T}$ is A-feasible, then the A-feasible schedule of $\mathcal{T}$ on m uniform processors is periodic with a period P from instant S_n where S_i is defined inductively as in Equation 5.1.*

We extend now the above result assuming the tasks are scheduled on a resource partition using a hierarchical global multiprocessor fixed-priority scheduler with preemption. In what follows we use $H(\mathcal{T})$ to denote the Least Common Multiple (LCM) of the task periods in $\mathcal{T}$. Without loss of generality we assume the tasks in $\mathcal{T}$ are ordered in decreasing order of their priority. Let $\mathcal{T}_{ext}$ denote the task set obtained by extending $\mathcal{T}$ with m tasks with offset 0 and period P and m being the number of processors: $\mathcal{T}_{ext} = \{\tau'_1, ..., \tau'_m, \tau_1, \tau_2, ..., \tau_n\}$. The tasks in $\mathcal{T}_{ext}$ are ordered in decreasing order of their priority.

Theorem 5. *For any preemptive fixed-priority scheduling algorithm A, if the asynchronous periodic deadline constrained taskset $\mathcal{T}_{ext}$ is A-feasible on m uniform processors and P is an exact divisor of $H(\mathcal{T})$, then the subset $\mathcal{T} \subset \mathcal{T}_{ext}$ is A-feasible on a multiprocessor time partition with period P and the A-feasible schedule is periodic with a period equal to $H(\mathcal{T})$ from instant S_n, defined by Equation 5.1. (Assuming that the execution time of each task is constant.)*

Proof. For a set of m processors we use $\sigma_j(t) = i$ to indicate that task τ_i is scheduled on processor j at time t and $\sigma_j(t) = 0$ to show that processor j is idle at time t. The availability $a(t)$ of the processors is then defined as $a(t) = \{j \mid \sigma_j(t) = 0\} \subseteq \{1, ..., m\}$.

If $\mathcal{T}_{ext}$ is A-feasible on the m processors and P is a divisor of $H(\mathcal{T})$ then $H(\mathcal{T}_{ext}) = H(\mathcal{T})$ and according to Theorem 4, the schedule is periodic with period $H(\mathcal{T})$ from instant S_n.

We denote by $\sigma'(m)$ the schedule of the m highest priority tasks $\tau'_1, ..., \tau'_m$ and by a_m the corresponding availability of the processors. Since $S'_m = 0$, it results that $\sigma'(m)$ is periodic with period P from 0. Since tasks in $\mathcal{T}_{ext} - \mathcal{T}$ have higher priorities than any task in $\mathcal{T}$, then the scheduling of tasks in $\mathcal{T}$ will not interfere with tasks in $\mathcal{T}_{ext} - \mathcal{T}$ that are already scheduled. Moreover, for all $t \geq 0$ we have $a_m(t) = a_m(t + P)$ meaning that the availability of the processors repeats and forms a periodic resource partition with period P. Only at these moments we can schedule tasks in $\mathcal{T}$. Therefore, the task set $\mathcal{T}$ is schedulable with algorithm A on a m multiprocessor time partition with period P. □

5.4.2. Time Partition Generation

The first parameter of the multi-core time partition mTP that shall be defined is the length of the MTF or the period. Based on Theorem 5, in the first phase of our method, we choose the period of the mTP as the first randomly generated exact divisor of $H(\mathcal{T})$ greater or equal to a given threshold value. This threshold value is left to the choice

of the system designer. For example, a possible criteria which can be used when specifying this threshold value could be the preemption overheads.

Besides the threshold value for the period, our method takes as input a set of constraints, in the form of a set of scheduling windows in time partitions which may have been previously determined for other components in the system. If this set is empty then the next phase determines the minimal number of cores required by the component. In order to generate the mTP with utilization as small as possible, we must use the smallest number of cores m. If $U(\mathcal{T})$ denotes the total utilization of the set of tasks in the component, then $\lceil U(\mathcal{T}) \rceil$ is clearly a lower bound on m. As we assume that the maximum number m_{avail} of available cores is known, this number can be considered as an upper bound for m. For each possible value of m in the range $[\lceil U(\mathcal{T}) \rceil, m_{avail}]$ we follow the steps described below:

(1) we build a multi-core time partition where the set of scheduling windows is $SW = \{\langle c_k, 0, P, P \rangle \mid c_k = 1, .., m\}$,

(2) the component automaton and the time partition automaton $\mathcal{A}_{mTPM}$ are augmented with a Stop state which will be reached when the clock of the system hits the end of the feasibility interval,

(3) we test if the tasks are schedulable with the current value of m by asking UPPAAL to check the property $\phi_{overrun} \equiv$ (A [] not Error); if the verification fails we increment m and if $m \leq m_{avail}$ we go back to step 1, otherwise the tasks are not schedulable on the available cores with the chosen algorithm and the time partition generation fails.

If a set of scheduling windows is received as input, then we will consider the period P of the time partition to be computed as the LCM of the periods of the scheduling windows received as input constraints. Next, we derive a set of scheduling windows, encompassing the free time intervals on the multi-core platform. These scheduling windows will be considered as an initial time partition for the following phase of the method, instead of the time partition defined in step 1 above.

After the minimum number of necessary cores is determined, the next phase of our method consists in finding the scheduling windows of the mTP when those cores should be available for scheduling the tasks of the component. As these scheduling windows have the same characteristics as periodic tasks, scheduling on the parent level will not need to worry about any constraints on the tasks of the child component. We generate a set $TP_{initial}$ of scheduling windows as follows:

(1) we get an initial schedule for the tasks of a component using the diagnostic trace generated by the UPPAAL model checker for the reachability property $\phi_{stop} \equiv$ (E <> Stop),

(2) afterwards, we parse the trace and search for the period P_{min} with the minimum workload,

(3) using the partial schedule of tasks during the period P_{min}, we build the set $TP_{initial}$ containing only scheduling windows that overlap exactly the intervals in which there are tasks running on the m processors,

(4) since we want to keep the number of scheduling windows on each processor as low as possible, we further do a cleanup procedure in which subsequent scheduling windows separated by an inactive interval smaller than a given threshold value are merged into a single time interval.

After getting the initial configuration for the time partition $TP_{initial}$, we ask UPPAAL to check the property $\phi_{overrun}$ on the resulting model of the system in order to find out if any of the tasks misses a deadline. If the property is not satisfied a counterexample trace will be generated by UPPAAL and we use this trace to adjust the current configuration and get a new configuration TP. In order to obtain a new configuration of the scheduling windows from the current one we identify the task that missed its deadline (the problem task) and adjust the scheduling windows active between the task's last release and the moment when the deadline miss occurred by extending them with the time required by the problem task to not miss its deadline anymore. For example, if the problem task needed to execute for 3 more time units but no scheduling window in $\mathcal{TP}_{initial}$ provided these 3 units during the task's last release time and its deadline, we try to extend one or more windows in $\mathcal{TP}_{initial}$ such that we get these additional 3 units. A scheduling window can be extended by starting earlier or finishing later only if by doing this operation it does not overlap with another scheduling window on the same core (in the same or another time partition). Also the missing time units are added in sequence (i.e. it is not useful to add 2 execution units to 2 windows on different cores such that the 2 units overlap).

After computing the new $\mathcal{TP}$, we insert it in the model and repeat the previous step until the result of the verification is positive. Since, in the second phase of our method, when the minimum number of necessary cores was determined, we have checked if the tasks are schedulable with the full capacity of the processor available to them, it is guaranteed that the loop in this final phase will finish. Moreover, although a single possible schedule is used for the adjustment of $\mathcal{TP}$ (i.e. the one in the diagnostic trace) since the adjustment phase finishes when there are no more schedules that miss a deadline, the final $\mathcal{TP}$ guarantees that all tasks of the application will meet their deadlines.

5.5. Experimental Evaluation

In this section we present the results of our investigations into the influence of different parameters on the time partition generation. In order to observe the behavior of the proposed method in different circumstances, we performed a series of experiments on randomly generated tasksets with different parameters.

Consequently, investigations on the efficiency of the proposed model required a mean for generating task sets. As mentioned above, the proposed model uses a discrete time formalism. In the model, along with a real-valued clock, another integer-valued clock is utilized for keeping track of the time until the end of the feasibility interval. In UPPAAL the maximum value of an integer variable is 32767. This limitation imposes 32767 as an upper bound on the end of the feasibility interval for the tasksets used in the model. For this reason it is necessary to have an algorithm that randomly generates the periods of tasks as general as possible but with the restriction that the LCM of the periods remains under a specified threshold value. The task period generator we have used was proposed by Goossens and Macq in [90] and generates periods as divisors of a specified value. The task generation algorithm must also allow setting a target utilization $U(\mathcal{T})$ of the task set. In order to generate the task utilization values we have used a version of the UUnifast [49] algorithm, adapted for multiprocessors, in which we consider only generated utilization values that are in a specified interval. The procedure for generating all task parameters for a component is presented in Figure 5.3.

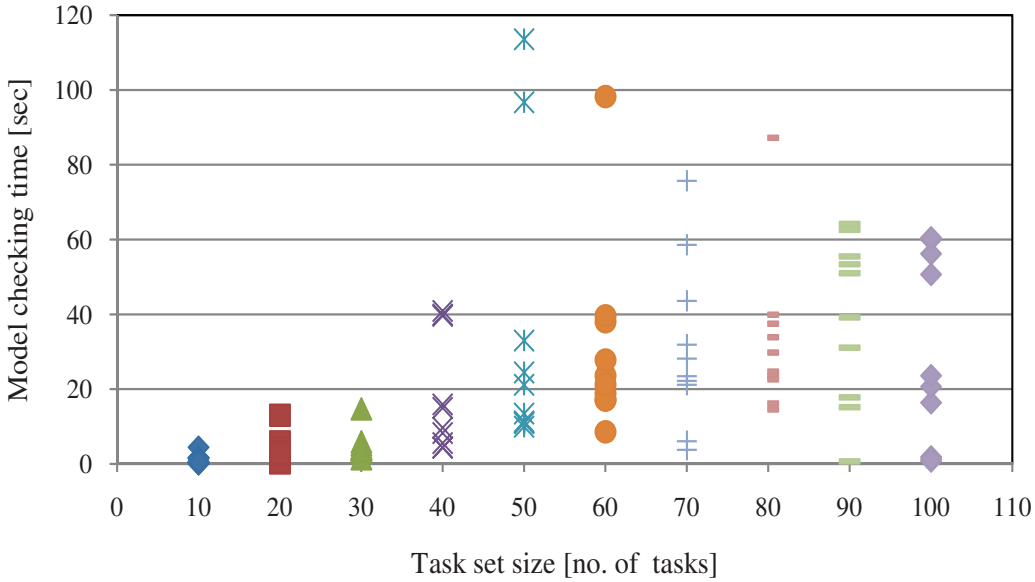
Modified UUnifast(n, $U(\mathcal{T})$) procedure
$sumU = U(\mathcal{T});\ totalU = 0;\ i = 1;$
while ($i \leq n-1$)
$nextSumU = sumU * random()^{\frac{1}{n-i}}$;
$u(i) = sumU - nextSumU$;
if ($minU \leq u(i) \leq maxU$ *and* $\frac{U(\mathcal{T}) - totalU - u(i)}{n-i} \geq minU$)
$totalU = totalU + u(i)$;
$sumU = nextSumU$;
$i = i + 1$;
$u(n) = min(MAX\_U,\ sumU,\ U(\mathcal{T}) - totalU)$;
return u;
GenerateTasks(n, $U(\mathcal{T})$) procedure
$u = UUnifast(n, U(\mathcal{T}))$;
for($i = 1$ *to* n)
$t_i = generatePeriod()$; // *as in [90]*
$w_i = max(1,\ \lfloor\ u(i) * t_i\ \rfloor)$;
$o_i = round(rand(minO,\ maxO) * t_i)$;
$d_i = round(rand(minD,\ maxD) * (t_i - w_i)) + w_i$;

Figure 5.3: Task generation procedure

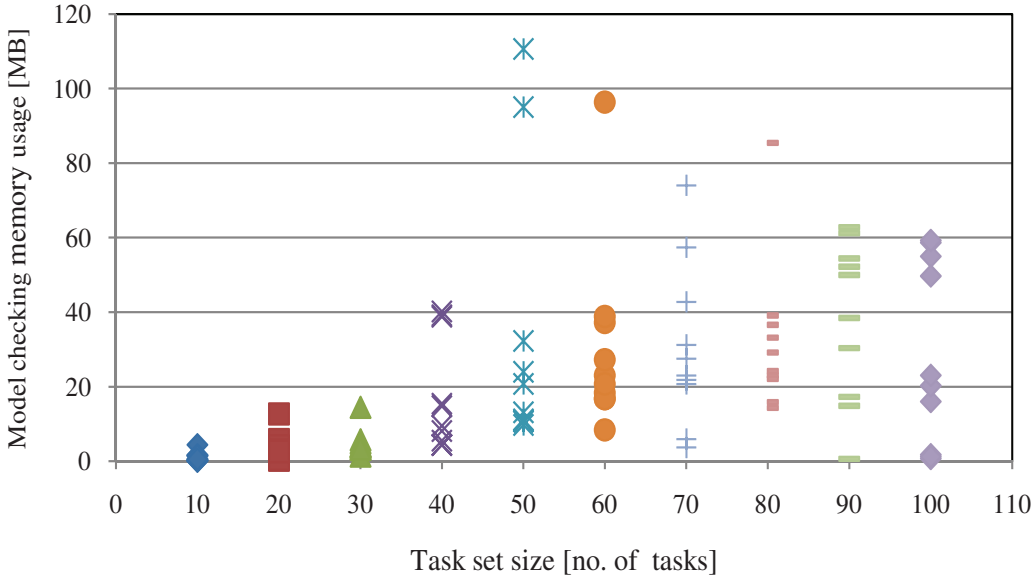
The experiments were run on a machine with Intel Core 2 Quad 2.40 GHz processor and 4 GB RAM running Ubuntu and the utility program memtime [3] was used for measuring the model checking time and memory usage.

As one of the major concerns related to model checking is state space explosion, in the first experiment we analyzed the scalability of our model. The results presented for this experiment were obtained for the model in which the component and resource generator automata contain the Stop states. Since the proposed time partition generation method implies checking repeatedly the property A[] not Error for the system model we measured the model checking time and memory usage for this operation for tasksets of different size. In order to do this we have used randomly generated sets of tasks with periods in the range $[10, 300]$ and utilizations (i.e. $executiontime/period$) generated in the range $[0.05, 1]$. The offset of each task was set to a value equal to the period multiplied with a randomly generated number in the interval $[0, 0.3]$. The deadline of each task was set by multiplying a randomly generated number in the interval $[0.7, 1]$ with the period of the task. Task priorities are assigned according to the RM scheduling policy. Also, the task sets were accommodated by a resource partition with 9 active time intervals and a total utilization of 4.5. Figure 5.4 shows how the model checking time and memory usage increase with the number of tasks in the set. It can be noticed that for the same size of the task set the performance of the model checking can vary between rather larger limits (e.g. for sets of 50 tasks the model checking time grows from 10 seconds to approximatively 115 seconds). This is due to the size of the hyper-period of the task sets (i.e. the LCM of the task periods), larger the hyper-period larger the model checking time and memory consumption.

Since our method assumes that the period of service contract is chosen as the smallest divisor of the LCM of the task periods greater or equal to a threshold value and we let the system designer specify this threshold value, the goal of our second experiment is to see what are the effects of different periods on the final service contract utilization. Using the same parameters as for the first experiment, we have generated three sets of 20, 30 and 40 tasks, respectively. For all sets the LCM of the task periods is the same. The total utilization for each of these task



(a) Model checking time



(b) Model checking memory usage

Figure 5.4: Influence of task set size on model checking performance

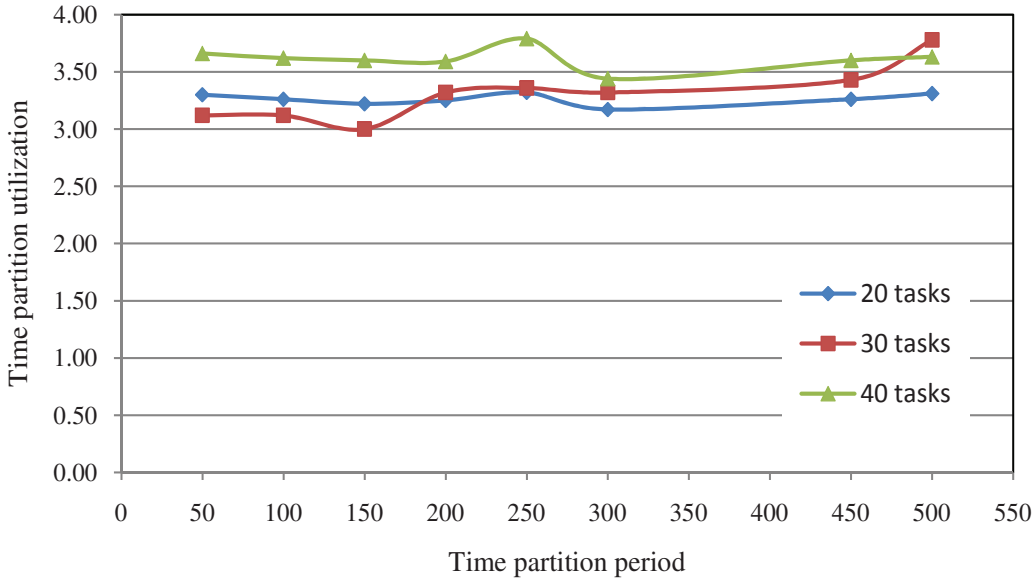


Figure 5.5: Influence of task set size and time partition period on multi-core time partition utilization

sets is approximately 3.0. The results in Figure 5.5 show that although there are values of the period for which the utilization (bandwidth) of the time partition for the component presents higher values, there are also large intervals for which the time partition utilization is almost constant. We believe the higher values are due to the fact that we start our service contract generation procedure by choosing the period with minimum workload in the UPPAAL trace. On the other hand, one may have noticed that an increased number of tasks in the set does not necessary determine a higher resource utilization. The set with 40 tasks always gets a higher resource utilization only because its total utilization is slightly over 3.0 (i.e. 3.16), while for the sets with 20 and 30 tasks, which have equal total utilization (2.88), there are cases in which the service contract for 30 tasks has lower utilizations than the one for 20 tasks.

In the next experiment we tested what happens if more than the minimum number of required cores is used. For this purpose we have generated two task sets with 20 and 40 tasks, respectively and total utilization approximately 2.0. The individual parameters of the tasks were generated using the same limit values as for the other experiments. It can be seen in Figure 5.6 that the minimum bandwidth loss is guaranteed only when using the minimum necessary number of cores, although for certain values of the time partition period the resource utilization is the same even when more cores are used. A higher number of used cores may determine a tendency to increase the concurrency level in the initial phase of our method when the period with the minimum utilization is determined and since the maximum concurrency will lead to a higher workload, the chosen period will not cover this high concurrency and as a consequence we will get a higher number of missed deadlines in the next phase of the proposed method.

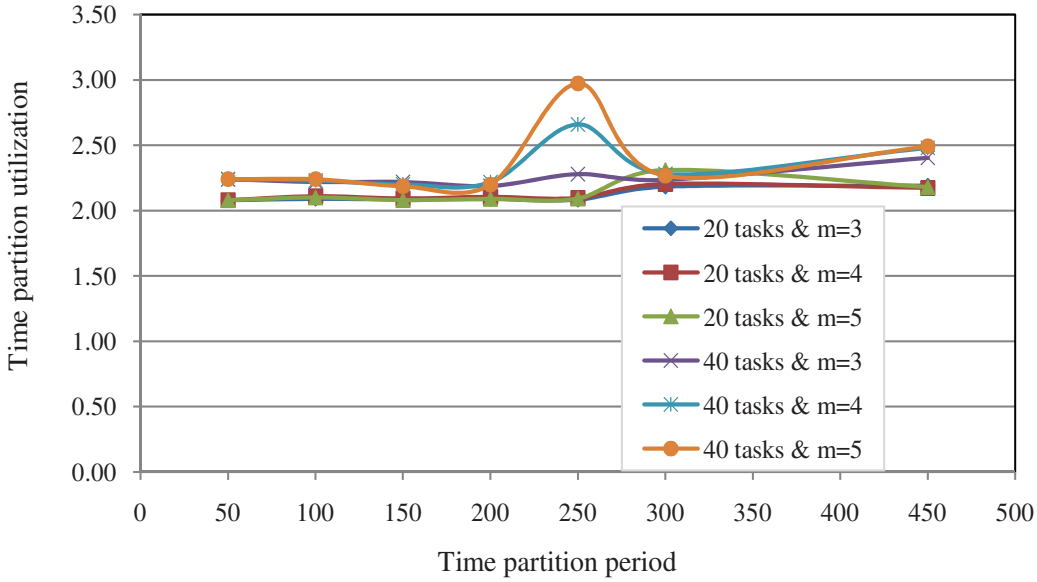


Figure 5.6: Influence of core number on multi-core time partition utilization

5.6. Concluding Remarks

In this chapter, we have presented a technique for generation of multi-core time partitions. The problem of determining the time partition hosting a system component was formulated as a reachability problem and a technique based on verification of two temporal logic formulas was proposed. The experiments performed on the model have shown that the method is rather scalable as it can be used for systems consisting up to 100 tasks. Moreover, the experiments have also shown that although the targeted systems have hard real-time constraints, the solution proposed here is capable of generating time partitions with low utilization loss.

The contributions of this chapter are focused on the following areas:

c1. We show that any feasible schedule of a set of periodic independent tasks executed in a multiprocessor time partitioned system repeats itself after some period. This result has not been proved before.

c2. We introduce model checking as a technique for designing the multi-core temporal partition that fulfills the timing constraints of a component. Such temporal techniques are currently designed manually and therefore are subject to human error.

6. RESOURCE SHARING IN MULTI-CORE REAL-TIME SYSTEMS

6.1. Motivation

Over the years, the real-time research community put considerable efforts for developing multiprocessor scheduling algorithms assuming real-time applications consist only of independent tasks. However, typical applications include tasks that share resources like data structures or I/O devices which require appropriate techniques for ensuring mutual exclusion, such that only a single task uses the resource at all times.

In the previous chapters we focused on the problem of scheduling a set of real-time applications consisting of independent or interacting tasks on a multi-core platform. However, task interactions issues were limited only at the various precedence constraints that can appear between tasks. In what follows the focus shifts on issues resulted from coordinating task access to shared data structures (e.g. queues, stacks, lists).

Mutual exclusion mechanisms work by enforcing a task to block while waiting for the release of a resource. The time spent by a task waiting for a resource is critical for hard real-time systems where a missed deadline can cause irreparable damage. Therefore this waiting time must be bounded. Schedulers based on priority-driven scheduling algorithms, either static or dynamic, are susceptible to priority inversions due to blocking. When a high priority task blocks waiting for a resource held by a lower priority task, if the low priority task is repeatedly preempted by tasks with intermediate priority, it will not be able to complete and release the resource, causing the high priority task to remain blocked for a very long time.

For single processor real-time systems, there is an extensive set of scheduling theories handling the priority inversion problem, among these the family of Priority Inheritance Protocols (PIPs) [151], the Priority Ceiling Protocol (PCP) [151] or the Stack Resource Policy (SRP) [24].

In multiprocessor systems, the priority ceiling protocols are affected by the parallel execution of tasks and consequently, they need considerable revisions. For partitioned multiprocessor scheduling, there are a number of extensions of the uniprocessor resource sharing protocols (e.g. [61, 87, 88, 124, 143], but for global multiprocessor scheduling the research is just in initial phase (e.g. [50, 74]). We believe this gap is due to the so called “Dhall effect” [73], a scheduling paradox which makes that tasksets with low processor utilization are unschedulable on a multiprocessor platform. Only in late 1990s and early 2000 it was proved that this effect occurs only for particular sets of tasks which may never appear in real applications [86, 141]. After this moment, global multiprocessor real-time scheduling became a major research area.

Furthermore, global multiprocessor scheduling under PIP suffers of a major limitation. On uniprocessor systems, a low priority job can block a higher priority job, directly or indirectly, only once and only if it has been released prior to the latter. However, on multiprocessor systems under PIP, a low priority task can interfere multiple times with the same higher priority job and increase significantly its blocking time [74]. Existing protocols have tackled this issue in various ways. First, there was

the FMLP, a protocol for global and partitioned scheduling, designed for tasksets with priorities assigned based on the EDF policy. According to this priority ordering, at most two jobs of a low priority task will interfere with any job of a higher priority task. A more recent protocol, the P-PCP [74], controls the lower priority interference by tight control of the moment when a task is allowed to use a shared resource. However, in P-PCP, reducing the interference comes at the cost of reducing also parallelism and thus, the efficiency of the multi-core platform.

In this chapter, we improve the state-of-the-art in resource sharing protocols for global multiprocessor real-time scheduling by designing a new protocol called Limited Blocking Priority Ceiling Protocol (LB-PCP). This protocol will be extended in the next chapter for virtualized real-time systems. LB-PCP can be applied for synchronizing non-nested resource accesses between sporadic real-time tasks scheduled according to any global fixed-priority preemptive scheduling policy. The protocol allows one to control the number of blockages suffered by each task from lower priority tasks executing critical sections, without forbidding totally the execution of such tasks. This way, LB-PCP can limit the interference on a higher priority task from lower priority tasks no matter what priority assignment policy is used and does not restrict the ability to exploit the parallelism provided by the multi-core platform.

Moreover, since a protocol should also have a schedulability test to check pre-runtime deadline guarantees, we also give a schedulability test based on response-time analysis of our protocol. While in previous chapters, we employed model checking to derive schedulability tests for virtualized real-time systems, in this chapter and the following one we develop analytical schedulability tests and leave model checking as part of future work. The schedulability analysis in the present chapter extends the state-of-the-art response-time analysis of Guan et al. [94] to include resource sharing based on LB-PCP.

6.2. System Model

We focus on the problem of scheduling a set of real-time tasks, $\mathcal{T} = \{\tau_1, .., \tau_n\}$, which are released either periodically, at fixed time intervals, or sporadically with a minimum inter-arrival time between two successive releases. We use τ_i^j to denote the j^{th} job of task τ_i. The tasks are scheduled on a multi-core or a multiprocessor platform comprised of m identical processors using a global fixed-priority preemptive scheduling policy, i.e. a task is not assigned a-priori to a specific processor. Further, we make the following assumptions about the task model:

a1. Each task τ_i $(1 \leq i \leq n)$ is characterized by a tuple (C_i, T_i, D_i), where T_i denotes the minimum inter-arrival time (or *period*) between two successive jobs of the task, C_i its *worst-case execution time* and D_i its *relative deadline*.

a2. All tasks are periodic or sporadic.

a3. We consider only *constrained-deadline* tasks with $D_i \leq T_i$, $\forall i$.

a4. Each task τ_i has a unique *base priority* equal to i.

a5. Tasks are sorted in decreasing order of their priority, i.e. for every pair (τ_i, τ_l) if $i < l$ then the priority of τ_i is higher than the priority of τ_l.

a6. During the execution of a job τ_i^j its priority may be increased at level h. We call this elevated priority level *effective priority*.

a7. The priorities of the tasks are assigned statically.

a8. A job cannot execute on two or more cores/processors simultaneously, and a core/processor cannot execute two or more jobs simultaneously.

a9. Preemption is permitted at any time.

a10. Task migration is allowed at any time.

a11. It is considered that the costs of preemption and migration are included in the execution time.

a12. Besides the processor, the tasks also share a set of resources which can be accessed only non-preemptively, in mutual exclusive manner.

Shared resources. The jobs of any task can issue requests for exclusive access to a set of shared resources $R_1, R_2, ..., R_r$. A request issued by a job τ_i^j for a resource R_k is *granted* as soon as the job holds the resource. After the job has executed for the amount of time that it requires R_k, the resource is said to be *released*. If a request by job τ_i^j for resource R_k cannot be granted immediately then τ_i^j is said to be *blocked* on R_k. When the resource access to R_k will be granted, τ_i^j will be *unblocked*.

A request for a resource R_l can be *nested* within the request for another resource R_k if and only if the request for R_l is issued after the request for R_k is issued and completes before the request for R_k completes. Here, we consider only *non-nested* resource accesses but nested accesses could be handled by using resource groups and group locks as in FMLP [50]. Furthermore, we assume that a task can be preempted while it holds the lock of a resource and that the task will hold the lock until it explicitly releases it.

We assume that for each task τ_i the worst case resource usage among all requests for a resource R_k by a job of τ_i is $C_{i,k}$. Further, we use $CT_{i,k}$ to denote the worst-case total resource usage time for R_k by any single job of τ_i. We use $\mathcal{R}(\tau_i)$ to denote the set of resources used by τ_i.

Definition 12. *The worst-case response time RT_i of task τ_i is the longest time from the arrival of any job τ_i^j of τ_i until the job completes executing. In terms of worst-case response time, a task τ_i is schedulable if $RT_i \leq D_i$.*

Interference and workload. For uniprocessor systems, the common approach to analyze the schedulability of constrained-deadline sporadic tasks is to search for a few worst-case job arrival sequences and test if the considered scheduling algorithm can successfully schedule all these worst-case sequences. However, for global multiprocessor scheduling of sporadic tasks there is no known worst-case job arrival sequence. This means that we cannot determine exactly the interference suffered by a task during a time interval as a result of the execution of the other tasks during that interval and instead, we can only derive an upper-bound [26].

Formally, the *interference* $I_i(x)$ on task τ_i over a time interval of length x is the sum of all intervals in which τ_i is ready for execution but cannot be scheduled due to the execution of higher priority tasks. We can also define the interference $I_i(a, x)$ of task τ_a on task τ_i as the sum of all intervals in which τ_i cannot execute because τ_a is executing.

The work done by a task τ_i in an interval of length x is denoted by the *workload* $W_i(x)$. In terms of task workload, the interference $I_i(a, x)$ denotes the part of the workload $W_a(x)$ that can prevent task τ_i from running. The workload $W_i(x)$ is divided in three parts (depicted in Figure 6.1) as follows:

- *carry-in*: the contribution of at most one job with release time prior to the beginning of the analyzed time interval and deadline in the analyzed time interval.
- *body*: the contribution of all jobs that are released and have deadline in the analyzed time interval.
- *carry-out*: the contribution of at most one job that is released prior to the end of the analyzed time interval but has deadline after the analyzed time interval.

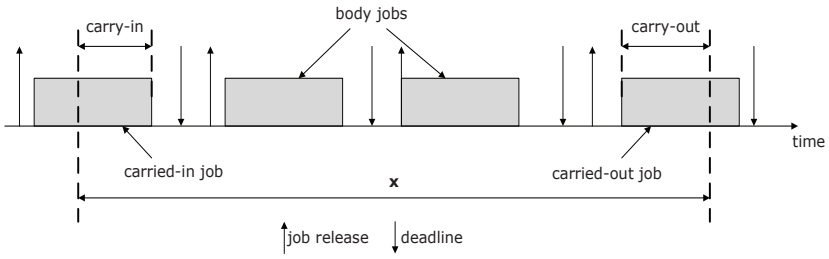


Figure 6.1: Workload components of task τ_i in an interval of length x

6.3. Schedulability Tests for Global Multiprocessor Systems

In this section we outline a sufficient schedulability test for global multiprocessor scheduling based on response time analysis (RTA) of the system under test, developed by Guan et al. [94]. In the following sections we extend this Response Time Analysis (RTA) schedulability test to handle mutually exclusive access to shared resources using LB-PCP. The test in [94] is based on a technique for global multiprocessor schedulability analysis proposed by Baruah [26], which can be summarized as follows:

(1) Consider the interval from the arrival r_i of some task τ_i until its deadline d_i, referred as the *problem window*, at the end of which the task misses its deadline.

(2) After ignoring tasks with lower priorities than τ_i, extend the problem window to an earlier time instant t_0 which is the latest time instant before r_i at which at least one processor is idle.

(3) Identify a condition necessary for the deadline miss to occur, for example during the interval $[r_i, d_i)$ all m available processors execute higher priority tasks for more than $D_i - C_i$.

(4) Derive an *upper bound* on the maximum interference on task τ_i from higher priority tasks in interval $[t_0, d_i)$.

(5) Form a schedulability test by negating the unschedulability test given by the inequality between the upper bound on interference and the amount of execution necessary for τ_i to miss its deadline.

By using Baruah's technique, the number of tasks which contribute with carry-in work is reduced to $(m-1)$ since at time t_0 at least one processor is idle. In [94], Guan et al. use this observation to improve the response time analysis proposed by

Bertogna and Cirinei [44] which derive an upper bound on the response time of a task in a globally scheduled system using upper bounds on the interference but assume that all higher priority tasks have carry-in. The upper bound on the response time of a task can be derived by the following fixed-point iteration (see Theorem 7 in [44]):

$$RT_i = C_i + \left\lfloor \frac{1}{m} \sum_{\forall \tau_h \in hp(i)} I_i(h, RT_i) \right\rfloor, \tag{6.1}$$

where $hp(i)$ denotes the set of tasks with priorities higher than τ_i and $I_i(h, RT_i)$ is the interference on τ_i from task τ_h in a time interval of length RT_i. The iteration starts with $RT_i = C_i$ and continues until either RT_i converges or $RT_i > D_i$ which means that the system is unschedulable.

Guan et al. derived a RTA-based schedulability condition by considering the upper bounds for workload and interference of the higher priority tasks in a problem window of length x. In what follows, Guan's RTA is detailed. Since only $m-1$ tasks do carry-in, they define two kinds of workload and interference bounds: one for when the task does not have carry-in (NC) and one for when it does have (CI). The upper bounds on the workload of a task τ_h with higher priority than τ_i during the problem window are defined by $W_h^{NC}(c,x)$ and $W_h^{CI}(c,x)$, computed for the scenario depicted in Figure 6.2 where $c = C_h$. In what follows we will use the notation $\|E\|_{up}$ as shorthand for $max(E, up)$ and $\|E\|^{lo}$ as shorthand for $min(E, lo)$.

$$W_h^{NC}(c,x) = \left\lfloor \frac{x}{T_h} \right\rfloor c + \|x \mod T_h\|^c \tag{6.2}$$

$$W_h^{CI}(c,x) = \left\lfloor \frac{\|x-c\|_0}{T_h} \right\rfloor c + c + \alpha(h,c,x), \tag{6.3}$$

where $\alpha(h,c,x)$ represents the carry-in:

$$\alpha(h,c,x) = \left\| \|x-c\|_0 \mod T_h - (T_h - RT_h) \right\|_0^{c-1}. \tag{6.4}$$

Note that the carry-in is limited to $c-1$. This is a result of the way time is represented in [94] and [44]. Time is represented as non-negative integer values where each values t represents the whole interval $[t, t+1)$.

If task τ_i is schedulable then an upper bound on the interference of the higher priority task τ_h can be derived as:

$$I_i^{NC}(h,c,x) = \|W_h^{NC}(c,x)\|^{x-C_i+1} \tag{6.5}$$

$$I_i^{CI}(h,c,x) = \|W_h^{CI}(c,x)\|^{x-C_i+1} \tag{6.6}$$

Note that the interference in Equation (6.1) can be computed just as $I_i^{CI}(h, C_h, RT_i)$ (i.e. $I_i(h, RT_i) = I_i^{CI}(h, C_h, RT_i)$).

Using Equations (6.5) and (6.6) the *total interference* $\Omega_i(x)$ on task τ_i is computed as the maximal value of the sum of all higher-priority tasks' interference among all possible cases:

$$\Omega_i(x) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}} \left(\sum_{\tau_h \in \tau^{NC}} I_i^{NC}(h, C_h, x) + \sum_{\tau_h \in \tau^{CI}} I_i^{CI}(h, C_h, x) \right), \tag{6.7}$$

where $\mathcal{Z}$ is the set of all partitions of the set of tasks with priority higher than τ_i into tasks with carry-in (τ^{CI}) and tasks without carry-in (τ^{NC}) such that $\tau^{NC} \cap \tau^{CI} = \emptyset$ and

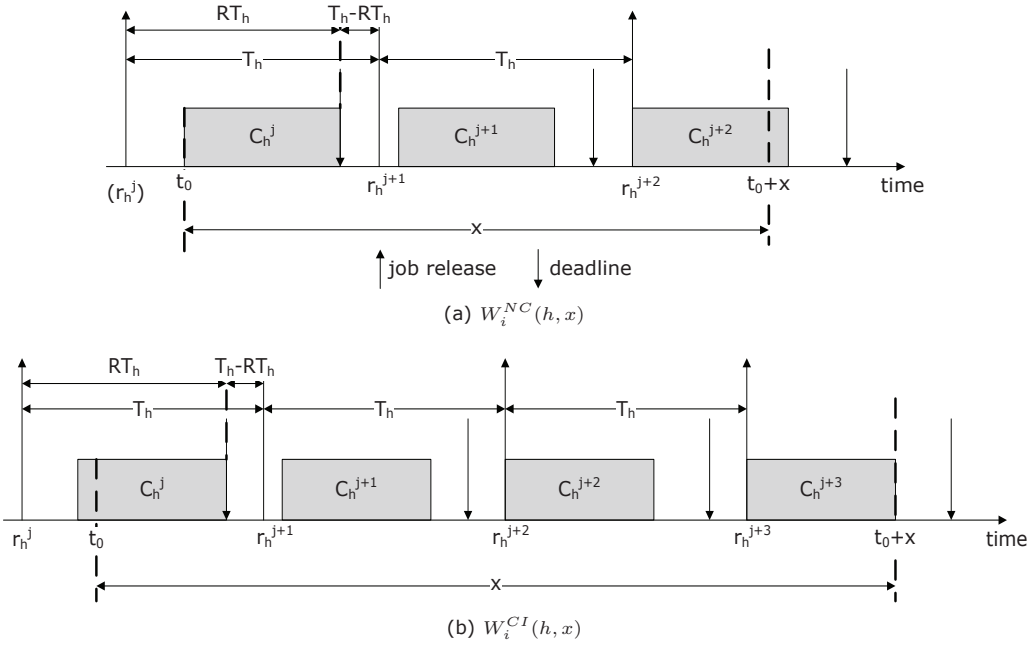


Figure 6.2: Worst-case workload of τ_h in problem window of length x

$|\tau^{CI}| \leq m-1$.

Using the total interference defined by Equation (6.7), Guan et al. derive a new expression for the fixed-point iteration in Equation (6.1) and derive an upper bound on the response time RT_i by doing iterative fixed-point search on the following expression (see Theorem 1 in [94]):

$$RT_i = C_i + \left\lfloor \frac{\Omega_i(RT_i)}{m} \right\rfloor. \tag{6.8}$$

Iteration starts with $RT_i = C_i$ and continues until the value of RT_i does not change anymore, in which case RT_i gives the worst-case response time of τ_i, or when $RT_i > D_i$ in which case the task is unschedulable.

6.4. The Limited Blocking Priority Ceiling Protocol

In this section we present in detail the Limited Blocking Priority Ceiling Protocol (LB-PCP) proposed by us for resource sharing in global fixed-priority preemptively scheduled systems.

Global multiprocessor scheduling under PIP suffers a major limitation. Whenever a job τ_l^j holds a shared resource R_k, its priority may be raised from l to h as a result of a higher priority job $\tau_h^{j'}$ *directly blocking* on R_k (requesting R_k). While τ_l^j runs at priority h it may prevent another high priority job $\tau_{hl}^{j''}$ $(l > hl > h)$ from running. We call such situations *indirect blockage*. On uniprocessor systems, a low priority job can block a higher priority job, directly or indirectly, only once and only if it has been

released prior to the latter. As pointed out in [74], on multiprocessor systems under PIP, a lower priority task can interfere multiple times with the same higher priority job and increase significantly its blocking time.

With LB-PCP we provide the possibility of reducing the indirect blockage of a task. For each priority level i (task τ_i), we define a parameter IBT_i $(1 \leq i \leq n)$ indicating the number of times jobs with priority higher than i can be indirectly blocked. This way we control the lower-priority interference on jobs on these priority levels. A higher value for IBT_i increases parallelism but also implies that more jobs with base-priority lower or equal to i will be allowed to execute at higher effective priority. On the other hand, by lowering IBT_i we may decrease the level of parallelism but, the lower-priority interference will also be decreased.

Notations. We introduce first some notations that are used in the description of the protocol. Each of the values bellow are updated at each scheduling event (job release, job finish, resource request, resource release):

- $\lceil R_k \rceil$: Denotes the highest priority of any task that uses resource R_k $(1 \leq k \leq r)$. $\lceil R_k \rceil$ is therefore the priority ceiling of resource R_k.
- EP_i: The effective priority of the current job of task τ_i. This is equal to i when τ_i does not hold any resource and can be increased up to $\lceil R_k \rceil$ when τ_i holds resource R_k $(1 \leq k \leq r)$.
- IBC_i: The current number of times that jobs with priority higher than i can be indirectly blocked. Note that a job is indirectly blocked if it is ready for execution but cannot execute (not enough processors available) and there is another job with base priority lower than it, but running with effective priority higher or equal to i while it holds the lock of a resource. Initially, this number is equal with IBT_i and is decreased every time a job with base priority higher than i gets indirectly blocked.
- LPR_i: The number of jobs with base priority lower than i but potential effective priority higher than i. Actually, this is the number of tasks with base priority lower than i that currently hold a resource with priority ceiling greater than i. Thus, at any moment the priority of these tasks could increase.
- LBT_i: The base priority of the job that indirectly blocks the current job of task τ_i.

In order to control the amount of indirect blockage affecting a task LB-PCP enforces at all times the following rule:

> **A job τ_i^j $(1 \leq i \leq n)$ is not granted a shared resource R_k unless the following condition remains true for all priority levels lv, $i > lv > \lceil R_k \rceil$: the number of jobs with base priority lower than lv using a resource with priority ceiling higher than lv is at most IBC_{lv+1}.**

Otherwise job τ_i^j is suspended, even if R_k is not locked. In order for the condition to hold is is also required that the IBT_i configuration parameters satisfy the property $IBT_1 \leq IBT_2 \leq ... \leq IBT_n$. Furthermore, since the m highest priority tasks can never be indirectly blocked, we have $IBT_1 = ... = IBT_{m+1} = 0$. Note that the difference $IBT_{i+1} - IBT_i$ gives the maximum number of times a job of task τ_i can be indirectly blocked by a lower priority job.

The global fixed-priority scheduling algorithm supporting resource sharing based on LB-PCP is described in Algorithm 1 along with Algorithm 2. Before going into

the details of the algorithm, several facts, not visible in the description of algorithms 1 and 2, must be mentioned:

(1) The IBC_i $(1 \leq i \leq n)$ counter is initialized with the value of IBT_i.

(2) Whenever a job with base priority i finishes, for all priority levels l, $i < l \leq n$, IBC_l is incremented with $[(IBT_{i+1} - IBT_i) - (IBC_{i+1} - IBC_i)]$ (the number of indirect blockages suffered by the finished job of τ_i). This way we set again $IBC_{i+1} - IBC_i = IBT_{i+1} - IBT_i$.

(3) For all priority levels i the LPR_i counter is decremented by 1 whenever a job with base priority lower than i releases a resource with priority ceiling higher than i.

(4) For all priority levels i the LBT_i is cleared when either the current job of task τ_i finishes or the job that was blocking τ_i releases the resource that determined the increase of its effective priority.

We assume that point (1) above is executed at system initialization, while points (2)–(4) are performed whenever a job is released or finishes its execution and whenever a job requests or releases a resource. Afterwards, the scheduling algorithm 1 is executed. Algorithm 1 evaluates the current status of each task in decreasing order of their effective priority. For each task that has a job ready for execution or executing, the algorithm checks if it can schedule it in the next time slice.

Since LB-PCP allows only a given number of indirect blockages for each task, every time a job requests a resource the algorithm has to check that by granting the access will not violate this constraint. Thus, if the job τ_i^j is not requesting any resource and a processor is available then the job is scheduled immediately (Lines 3 & 4 in Algorithm 1). If there is no available processor but there is a job with lower base priority than τ_i^j currently running with effective priority higher than i, then this job indirectly blocks τ_i^j and we decrease by 1 the IBC counter for all priority levels lower than i (Lines 25–30 in Algorithm 1). The counter is decremented only the first time the indirect blockage is detected, that is, if we detect the blockage of job τ_i^j at time t and between time t and $t + t'$ the job does not execute at all and the job that is blocking it does not finish its critical section, we will decrement the counter only at time t. To make sure that we count only once the blocking of τ_i^j by a certain lower priority job $\tau_l^{j'}$ using a specific resource, we set LBT_i to the base priority of that blocking job (Line 29 in Algorithm 1) and reset it only when τ_i^j finishes or the blocking job finishes the current resource access. This is all right since the response time analysis described in Section 6.5., considers that a task indirectly blocks another task for the whole duration of the resource access. Moreover, if another job $\tau_{l1}^{j'}$ with $l_1 < l$ would start running while τ_i^j is blocked by $\tau_l^{j'}$ then that job will be counted as blocking τ_i^j only if the duration for which its priority is raised above i is longer than the duration of the resource access of $\tau_l^{j'}$.

If τ_i^j requests a resource R_k and the resource is not locked by another job, then we run Algorithm 2 to make a decision about whether the request should be granted or not. While τ_i^j holds R_k, its effective priority could be increased to $\lceil R_k \rceil$. This means that τ_i^j could indirectly block any job with priority lv, $\lceil R_k \rceil < lv < i$. Moreover, the job with base priority level lv could also be indirectly blocked by other jobs with priority lower than lv which already locked a resource with ceiling greater than lv. The number of such jobs is accounted by LPR_{lv}. Therefore the resource requested by τ_i^j is granted only if, for each priority level lv between i and $\lceil R_k \rceil$ (not including these limits), the number of allowed indirect blockages is greater than the number of jobs

Algorithm 1 Resource sharing under LB-PCP

for each job τ_i^j in decreasing effective priority order **do**
  if there are unassigned processors **then**
    if τ_i^j is not requesting a resource **then**
      assign processor to τ_i^j
    else
      Let R_k be the resource requested by τ_i^j
      if R_k is locked **then**
        Let τ_l be the task currently holding resource R_k
        if $l > i$ **then**
          $EP_l = i$
        end if
      else
        if τ_i^j is one of the m highest priority jobs **then**
          assign processor to τ_i^j
        else if R_k can be granted to τ_i^j according to rules in Algorithm 2 **then**
          assign processor to τ_i^j
          for $ind = max(m, \lceil R_k \rceil) + 1$ **to** $i - 1$ **do**
            $LPR_{ind} = LPR_{ind} + 1$
          end for
        else
          suspend τ_i^j
        end if
      end if
    end if
  else if τ_i^j is one of the m highest base priority ready jobs **and** there is a lower base priority job $\tau_l^{j'}$ running with priority $EP_l < i$ **then**
    for $ind = i + 1$ **to** n **do**
      $IBC_{ind} = IBC_{ind} - 1$
    end for
    Set LBT_i to the base priority of the job blocking τ_i^j
  end if
end for

with lower base priority that currently hold a resource with priority ceiling greater than the analyzed level lv. The access will be allowed only if $(IBC_{lv+1} - IBC_{lv}) - LPR_{lv} > 0$, $\forall\, lv,\ \lceil R_k \rceil < lv < i$, which means that each task τ_{lv} can be indirectly blocked at least one more time. If the resource cannot be granted because this could lead to the outrunning of the maximum allowed indirect blockages of at least one higher priority job, the job requesting R_k is suspended.

Remember that the number of blockages allowed on a priority level is decreased by 1 only when the job that indirectly blocks the job on level lv changes, which means that the same job cannot block twice another job while continuously holding a given resource. In order for a given job $\tau_l^{j'}$ to be considered as blocking more than once a job τ_{lv}^j $(l > lv)$, $\tau_l^{j'}$ should make another request for a resource and that will require reevaluation of condition in Algorithm 1.

Theorem 6. *At all time instants and for all priority levels* i, $1 \leq i < n$, $LPR_i \leq IBC_{i+1}$.

Proof. Whenever a job of a task τ_l with $l > i$ requests access to a resource R_k with

Algorithm 2 Access rules to R_k by job τ_i^j

```
1: for ind = ⌈R_k⌉ + 1 to i − 1 do
2:   if (IBC_{ind+1} − IBC_{ind}) − LPR_{ind} ≤ 0 then
3:     return false
4:   end if
5: end for
6: return true
```

$\lceil R_k \rceil < i$, the resource is granted only if, for all priority levels i, $l > i > \lceil R_k \rceil$, it is true that $IBC_{i+1} - IBC_i > LPR_i$ (see Algorithm 2). It follows that, R_k is granted only if the number of indirect blockages allowed on each level i is higher than the number of tasks with potential effective priority higher than i (task with lower base priority that currently hold a resource with priority ceiling higher than i).

Each of the LPR_i jobs can cause a single block of the current job of task τ_i. Therefore, once $IBC_{i+1} - IBC_i = LPR_i$ for at least one level i ($l > i > \lceil R_k \rceil$), no other jobs with base priority $l > i$ can run at effective priority higher than i, so LPR_i cannot increase. At this time, the number of potential indirect blockages on all levels higher or equal to i can be pessimistically computed as $\sum_{h=1}^{i} LPR_h \leq \sum_{h=1}^{i}(IBC_{h+1} - IBC_h) = IBC_{i+1} - IBC_1 = IBC_{i+1}$. Since $LPR_i \leq \sum_{h=1}^{i} LPR_h$ it follows that $LPR_i \leq IBC_{i+1}$. □

Example 6. *Consider a set $\tau = (\tau_1, .., \tau_6)$ comprised of six sporadic tasks and two shared resources R_1 and R_2. The tasks execute on a multiprocessor platform consisting of 3 processors ($m = 3$). Tasks τ_2 and τ_5 request resource R_1 and tasks τ_3 and τ_6 request resource R_2. Note that tasks request the resources immediately after they start executing and use the resources for their entire execution time. Tasks τ_1 and τ_4 do not use any shared resources. Figure 6.3 shows the schedule of this taskset under PIP, while the schedule under LB-PCP is shown in Figure 6.4.*

Under PIP, when task τ_6 requests R_2 at time t_0, the request is granted. Later, at time t_1, a new job of task τ_3 arrives and requests R_2 and gets directly blocked. As a result, the priority τ_6 is increased to level 3. At time t_2 a new job of τ_1 arrives and also a job of τ_2. The job of τ_2 directly blocks because it requests R_1 which is held by τ_5. Consequently the priority of τ_5 is raised to effective level 2. At the same time, the current job of τ_4 will be preempted and, because we have two jobs of lower base priority running, we say that τ_4 is indirectly blocked. In fact, τ_4 will be able to run again only after the job of τ_2 finishes, but this is not enough and the job of τ_4 will miss its deadline.

In contrast, under LB-PCP the request of τ_6 for R_2 is not granted at time t_0 and instead, the job of τ_6 is suspended until the job of task τ_4 finishes. Therefore, the interference on the job of task τ_4 will be lower and it will not miss its deadline anymore.

6.5. Schedulability of Limited Blocking Priority Ceiling Protocol

In this section we compute the worst-case response time RT_i of a task τ_i scheduled using a global fixed-priority preemptive scheduler with resource sharing under LB-PCP. This RTA is done for each task in decreasing order of their priority. There are five parameters that influence the value of RT_i: (1) the task's worst-case

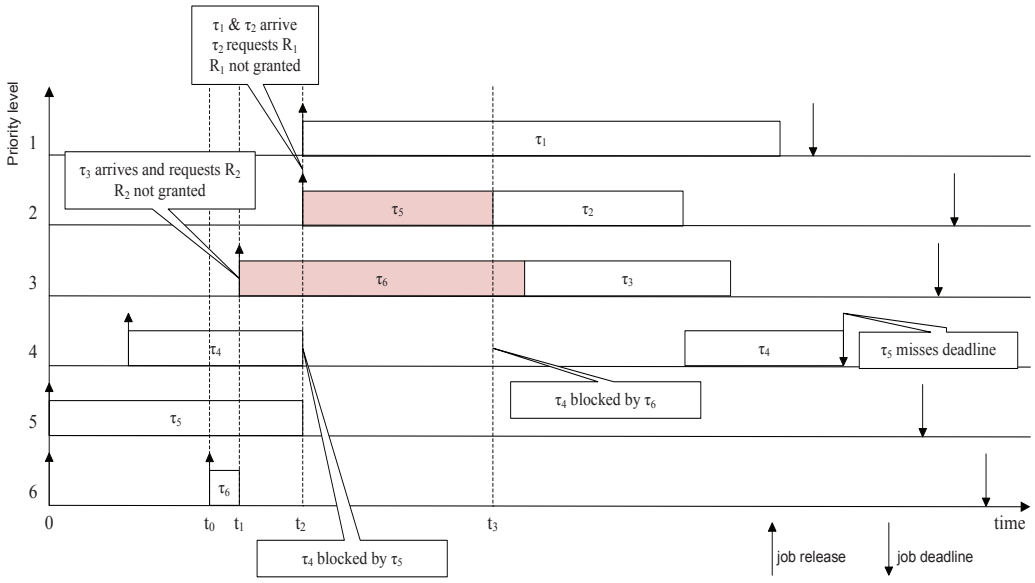


Figure 6.3: Example under PIP $(m = 3)$

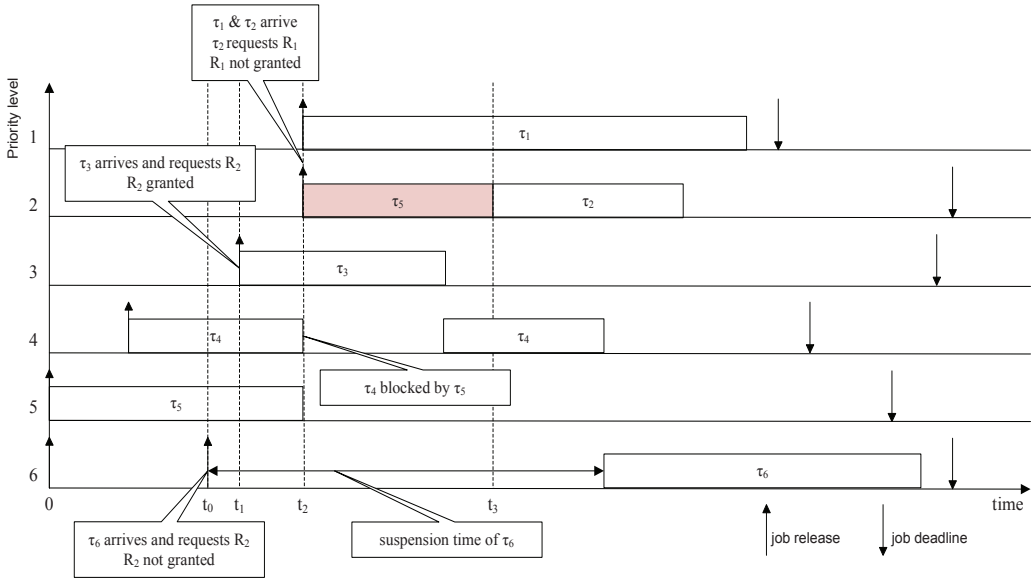


Figure 6.4: Example under LB-PCP $(m = 3)$

execution time C_i, (2) the amount of time task τ_i is blocked waiting for the resources it requests to be granted, (3) the amount of execution that lower priority tasks execute at a priority higher than that of τ_i, (4) the amount of time τ_i is suspended when condition on Line 15 in Algorithm 1 is violated, and (5) the amount of execution that tasks with priority higher than τ_i can perform.

Therefore, under LB-PCP there are two possible sources of blocking for τ_i:

(1) *Direct blocking* occurs when τ_i is one of the m highest priority ready tasks and requests resource R_k but the lock for R_k is locked by some other task. We denote the maximum total amount of time τ_i can be directly blocked as DB_i.

(2) *Indirect blocking* occurs when τ_i cannot execute because a lower base priority task τ_l executes with an effective priority higher than i while it accesses a resource. We denote the maximum total amount of time task τ_i can be indirectly blocked as IB_i.

In the worst case, a job of task τ_i requests a resource R_k just after the resource has been granted to a lower priority task and all higher priority tasks also request access to the resource. Therefore, the worst case direct blocking time can be expressed as follows:

$$DB_i(RT_i) = \sum_{R_k \in \mathcal{R}(\tau_i)} \left(N_{i,k} \cdot (\max_{\tau_l \in lp(i)} \{C_{l,k}\} - 1) + \sum_{\tau_h \in hp_k(i)} I_i^{CI}(h, CT_{h,k}, RT_i) \right) \tag{6.9}$$

where $N_{i,k}$ denotes the maximum number of times an instance of τ_i requires access to resource R_k. $I_i^{CI}(h, CT_{h,k}, RT_i)$ is the interference due to higher priority tasks requesting the same resource R_k during RT_i and is computed using Equation (6.6) but considering only the set of higher priority tasks that use R_k and assuming that the interference produced by a task τ_h is due to $CT_{h,k}$ units of work:

$$I_i^{CI}(h, CT_{h,k}, RT_i) = \| W_h^{CI}(CT_{h,k}, RT_i) \|^{RT_i - C_i + 1} \tag{6.10}$$

In the equations above we used $lp(i)$ to denote the set of tasks with priorities lower than τ_i and $hp(i)$ to denote the set of higher priority tasks. Also $hp_k(i)$ denotes the set of all tasks with priority higher than i that use resource R_k.

The indirect blocking time IB_i for task τ_i states that a lower priority task τ_l may interfere with τ_i multiple times and may delay the execution of τ_i just as much as any higher priority task. This is why it is desirable to limit the set of lower priority task instances that may execute while τ_i is ready for execution or is executing. Under LB-PCP, the number if indirect blockages for the job of priority i is limited to $IBT_{i+1} - IBT_i$. Therefore IB_i may be upper bounded as follows:

$$IB_i(RT_i) = \left\lfloor \frac{(IBT_{i+1} - IBT_i) \max_{\substack{\tau_l \in lp(i) \wedge \\ R_x \in \mathcal{R}(\tau_l) \wedge \lceil R_x \rceil < i}} (C_{l,x})}{\min(m, IBT_{i+1} - IBT_i)} \right\rfloor \tag{6.11}$$

Note that, in the worst case, a job of τ_i may be indirectly blocked by $IBT_{i+1} - IBT_i$ complete executions of critical sections of jobs with lower base priority.

Every time a job τ_i^j requests a resource R_k, it may be suspended if the condition on Line 15 in Algorithm 1 is violated. In the worst case the job has to wait for shared resource executions of every job with priority between i and $\lceil R_k \rceil + 1$ (not including the margins of the interval) and every resource, different from R_k, with priority ceiling higher than their base priority. Note that we do not consider interference

from lower priority jobs holding resources with priority ceiling higher than i since this interference is already considered in the IB_i component of the response. Furthermore, since the tasks currently executing with effective priority higher than their base priority may lead to the scenario where indirect blocking is not allowed anymore for some higher priority jobs ($IBC_{h+1} - IBC_h = 0$), job τ_i^j will also have to wait until the last of these jobs finishes its execution. Therefore, suspension time can be bounded by:

$$susp_i(RT_i) = \left\lfloor \frac{\Omega_{i,hp}^{sus}(RT_i)}{m} \right\rfloor + \sum_{R_k \in \mathcal{R}(\tau_i)} N_{i,k} \max_{\substack{\tau_h \in hp(i) \wedge \\ i > h > \lceil R_x \rceil}} (C_h) \tag{6.12}$$

with

$$\Omega_{i,hp}^{sus}(RT_i) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}^{\lceil R_k \rceil}} \left(\sum_{\tau_h \in \tau^{NC}} I_i^{NC}(h, \beta_h, RT_i) + \sum_{\tau_h \in \tau^{CI}} I_i^{CI}(h, \beta_h, RT_i) \right) \tag{6.13}$$

where $\mathcal{Z}^{\lceil R_k \rceil}$ denotes the set of all task partitions with base priority between i and min_k with $min_k = \min_{R_k \in \mathcal{R}(\tau_i)} \lceil R_k \rceil + 1$. Further,

$$\beta_h = \sum_{R_x \in \mathcal{R}(\tau_h) \wedge R_x \notin \mathcal{R}(\tau_i) \wedge \lceil R_x \rceil < h} CT_{h,x}. \tag{6.14}$$

Although the set partitioning problem is a NP-complete problem, in this situation, $\Omega_{i,hp}^{sus}(RT_i)$ and all total interference values that will be defined bellow can be computed in linear time by computing the $m-1$ maximal values of the difference $I_i^{CI}(h,c,x) - I_i^{NC}(h,c,x)$ and adding the sum of these difference values to the sum of the $I_i^{NC}(h,c,x)$ values, as pointed out by Baruah [26].

Another factor that must be considered in the response time analysis of task τ_i is the interference from higher priority tasks executing outside critical sections considered in the suspension time evaluation. This includes (1) tasks with priority h, $i > h > min_k$, executing outside critical sections or using resources with priority ceiling equal to h, and (2) tasks with priority $h \leq min_k$ executing outside critical sections of resources also used by τ_i:

$$\begin{aligned} \Omega_i(RT_i) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}} \Bigg(& \sum_{\substack{\tau_h \in \tau^{NC} \wedge \\ i > h > min_k}} I_i^{NC}(h, \theta_h, RT_i) + \sum_{\substack{\tau_h \in \tau^{NC} \wedge \\ h \leq min_k}} I_i^{NC}(h, \gamma_h, RT_i) \\ & + \sum_{\substack{\tau_h \in \tau^{CI} \wedge \\ i > h > min_k}} I_i^{CI}(h, \theta_h, RT_i) + \sum_{\substack{\tau_h \in \tau^{CI} \wedge \\ h \leq min_k}} I_i^{CI}(h, \gamma_h, RT_i) \Bigg) \end{aligned} \tag{6.15}$$

where

$$\theta_h = C_h - \Big(\sum_{\substack{R_x \in \mathcal{R}(\tau_h) \\ \wedge R_x \in \mathcal{R}(\tau_i)}} CT_{h,x} + \beta_h \Big) \tag{6.16}$$

and

$$\gamma_h = C_h - \sum_{\substack{R_x \in \mathcal{R}(\tau_h) \\ \wedge R_x \in \mathcal{R}(\tau_i)}} CT_{h,x} \tag{6.17}$$

From the discussions above, we can determine the worst-case response time of a task scheduled under LB-PCP by incorporating the blocking and suspension factors into the recurrence relation defined by Equation (6.8):

$$RT_i = C_i + susp_i(RT_i) + DB_i(RT_i) + IB_i(RT_i) + \left\lfloor \frac{\Omega_i(RT_i)}{m} \right\rfloor \tag{6.18}$$

The recurrence starts with $RT_i = C_i$ and ends when RT_i converges in which case RT_i gives the worst-case response time of task τ_i or when $RT_i > D_i$ which means that the task is unschedulable.

Example 7. *Consider a system consisting of the taskset $\tau = \{\tau_1, ..., \tau_8\}$ with task parameters given in Table 6.1, shared resources R_1, R_2, R_3 and R_4 and $m = 2$ processors. The tasks are ordered in decreasing order of their priority and their total utilization is 113.59%. Under LB-PCP each instance of a task can be indirectly blocked at most once. Columns 7 to 9 of Table 6.1 show the worst case response times of each task when resource sharing is handled according to PIP, P-PCP and LB-PCP, respectively. Note that for all protocols the taskset is schedulable. For tasks with low priority, PIP and P-PCP give smaller response times than LB-PCP. This is due to the larger suspension time suffered by these tasks under LB-PCP. However, for some of the medium priority tasks, the response time under LB-PCP is better than P-PCP due to the smaller suspension from lower priority tasks. At first sight, it seems that PIP would be better than both P-PCP and LB-PCP. This is because the effect of decreasing the lower priority interference is more visible on tasksets of larger cardinality than what is used in this example.*

Table 6.1: Numeric example for task response times (task set schedulable)

Task	C	T	D	U	Resource	RT		
						PIP	P-PCP	LB-PCP
τ_1	8	17	17	47.06%	R_1	10	10	10
τ_2	2	46	33	4.35%	R_1	10	10	10
τ_3	30	79	72	37.97%	R_3	36	47	41
τ_4	1	78	55	1.28%	R_2	24	38	29
τ_5	4	65	58	6.15%	R_3	40	51	51
τ_6	10	71	66	14.08%	R_4	38	39	44
τ_7	1	64	63	1.56%	R_2	29	30	55
τ_8	1	89	85	1.12%	R_4	34	34	57

Example 8. *Consider a system consisting of the taskset $\tau = \{\tau_1, ..., \tau_8\}$ with task parameters given in Table 6.2, shared resources R_1, R_2, R_3 and R_4 and $m = 4$ processors. The tasks are ordered in decreasing order of their priority and their total utilization is 125.74%. Under LB-PCP each instance of a task can be indirectly blocked at most once. Columns 7 to 9 of Table 6.2 show the worst case response times of each task when resource sharing is handled according to PIP, P-PCP and LB-PCP, respectively. Note that the taskset is schedulable only when LB-PCP is used. Task τ_5 is the first to miss its deadline under both PIP and P-PCP. Under PIP and P-PCP τ_5 can be indirectly*

blocked by both tasks τ_7 and τ_8 and as a result it becomes unschedulable. By contrast, under LB-PCP τ_5 is indirectly blocked only once and hence, the lower response time and the positive schedulability result.

Table 6.2: Numeric example for task response times (taskset not schedulable)

Task	C	T	D	U	Resource	RT		
						PIP	P-PCP	LB-PCP
τ_1	1	76	54	1.32%	R_4	34	34	34
τ_2	85	180	175	47.22%	R_3	86	86	86
τ_3	3	82	75	3.66%	R_3	44	44	44
τ_4	268	495	434	54.14%	R_2	291	291	291
τ_5	1	105	84	0.95%	R_1	108	164	78
τ_6	9	163	132	5.52%	R_1	-	-	79
τ_7	65	882	735	7.37%	R_4	-	-	514
τ_8	51	917	882	5.56%	R_2	-	-	576

6.6. Performance Evaluation

To characterize the performances of the schedulability test presented in Section 6.5., there are various metrics which can be used. A presentation of these metrics can be found in Section 2.3. For evaluating the performance of LB-PCP we follow a simulative approach that considers the number of schedulable tasksets detected by the schedulability test detailed in Section 6.5. among a randomly generated distribution of tasksets. Such a ratio is called *acceptance ratio*. Computing it mathematically is possible only for simple cases and consequently, the performance of a schedulability test is evaluated through extensive simulations, where a huge number of synthetic tasksets are generated using random parameters [49].

For evaluating the performance of the LB-PCP protocol, we implemented a tool capable of determining if a set of tasks sharing a specified set of resources is schedulable under LB-PCP or under P-PCP. The tool is platform independent Java code and can be used as a stand-alone application or as a library, providing an interface which can be used by other applications. Basically, the tool receives as input the path of a file containing the set of tasks, the path of a file containing the definitions of the shared resources and describing which tasks use which resources, the number of processors on which the tasks shall be scheduled and the name of the protocol to be used of arbitrating access to shared resources. Using this information, the tool tests if the taskset is schedulable using the equations presented in Section 6.5., for LB-PCP, and the ones presented in [74], for P-PCP respectively.

In what follows we present a set of experiments to analyze and compare the performance of global fixed-priority scheduling with LB-PCP and with P-PCP, the other major resource sharing protocol for which a schedulability test is defined.

6.6.1. Task and resource generation

To properly compare the two schedulability tests, the first issue is how to generate a distribution of tasksets that is representative of the general behavior of a real-time system. The way in which task parameters are selected influences greatly the measured performance [49]. Bini and Buttazzo propose the UUnifast algorithm to generate

tasksets with a uniform distribution of task utilization and a desired total utilization. The algorithm has been employed for evaluations of many uniprocessor schedulability tests, e.g. [142, 84, 72, 163, 131]. The algorithm generates *unbiased* tasksets (the distribution of tasksets is equivalent to selecting tasksets at random from the set of all possible tasksets and discarding the ones that do no meet some parameter constraints). However, the method proposed by Bini and Buttazzo is suitable only when a single processor platform is used because on a multiprocessor system it will generate tasks with utilization larger than one. A multiprocessor version of the method has been proposed by Davis and Burns [69]. The basic idea of their method is that if a task with utilization above one is generated, then the whole taskset generated up to that point is dropped and task generation is restarted. Instead of dropping the whole taskset, we use a slightly modified method and drop only the faulty task.

For each experiment we generate 1000 tasksets. Every task is generated as follows:

- Task utilization is generated randomly using the UUnifast algorithm [49] adapted for generating tasksets with total utilizations greater than 1 and task utilizations in some specified interval. In our case, the task utilization U_i is between 0.001 and 1.0 and the taskset utilization increases from 0.025 to 0.950 times m, in steps of 0.025.
- Task period T_i ($1 \leq i \leq n$) is generated according to a log uniform random distribution, in the time interval 1ms and 1000ms.
- The execution time C_i is computed as $C_i = U_i * T_i$.
- Task deadline D_i is generated using a uniform random distribution, in the range $[C_i + 0.7 * (T_i - C_i), T_i]$.

In the experiments, the priorities of the tasks are determined using several priority assignments policies:

p1) DM [117]: tasks with earlier deadlines have priorities higher than tasks with later deadlines,

p2) RM [123]: tasks with smaller periods have priorities higher than tasks with large periods,

p3) TkC [21]: assigns priorities based on the value of the difference $T_i - kC_i$, where k is computed based on the number of processors m:

$$k = \frac{m - 1 + \sqrt{5m^2 - 6m + 1}}{2m} \tag{6.19}$$

p4) DkC [69]: assigns priorities based on the value of the difference $D_i - kC_i$, where k is computed using Equation (6.19),

p5) T-C: assigns priorities such that tasks with lower $T_i - C_i$ have higher priority.

For each experiment we generate sets of resources, such that each resource is shared by a given number of tasks, selected randomly according to a uniform distribution. For all experiments, except the last one, we assume that a task accesses a resource for a time equal to its execution requirement.

The configuration elements of the two resource sharing protocols are set as follows:

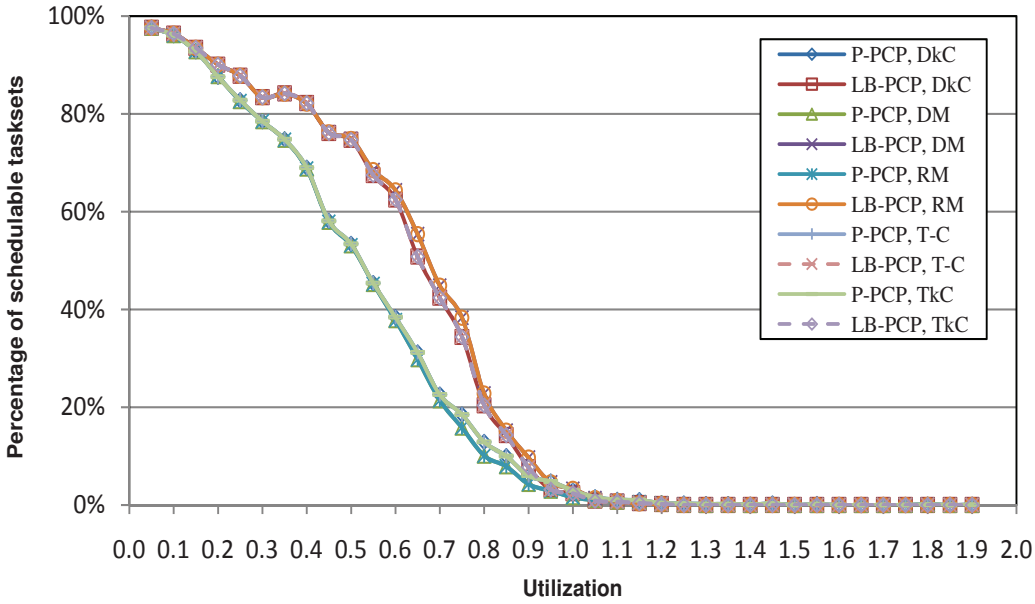


Figure 6.5: Percentage of schedulable tasksets for different priority assignment policies (m=2)

- For LB-PCP, we assume that a job can be indirectly blocked at most once: $IBT_{i+1} - IBT_i \leq 1, \forall\, i$.
- For P-PCP, we configure the tasksets such that the α_i factor (which controls the degree of parallelism) is n for the m highest priority tasks and $\alpha_i = m$ for the other tasks (as recommended in [74]).

6.6.2. Experiment 1: Priority Assignment

In the first experiment, we investigated the effect of different priority assignment policies on the performance of global preemptive fixed-priority scheduling with resource sharing under P-PCP and under LB-PCP. We tested the priority assignment policies mentioned above and for each test we considered a number of tasks equal to $5\times$ *the number of processors*. Figures 6.5, 6.6 and 6.7 show the number of tasksets deemed schedulable for the selected priority assignment policies for 2, 8 and 16 processors. Each task shares a single resource with another task and uses it for its whole execution.

From the figures we can see that selected priority assignment policies have similar performance with respect to the schedulability of the tasksets. Only RM performs slightly better in the case of LB-PCP but the difference is not significant. This observation is valid for the case when 2 processors are used, as well as for the 8 and 16 processors cases. We can conclude from this, that the selection of a given priority ordering it is not very significant for the analyzed scenario. Moreover, from all three figures it can be noticed that LB-PCP performs significantly better than PCP with respect to number of schedulable tasksets, but we will analyze this aspect in later experiments.

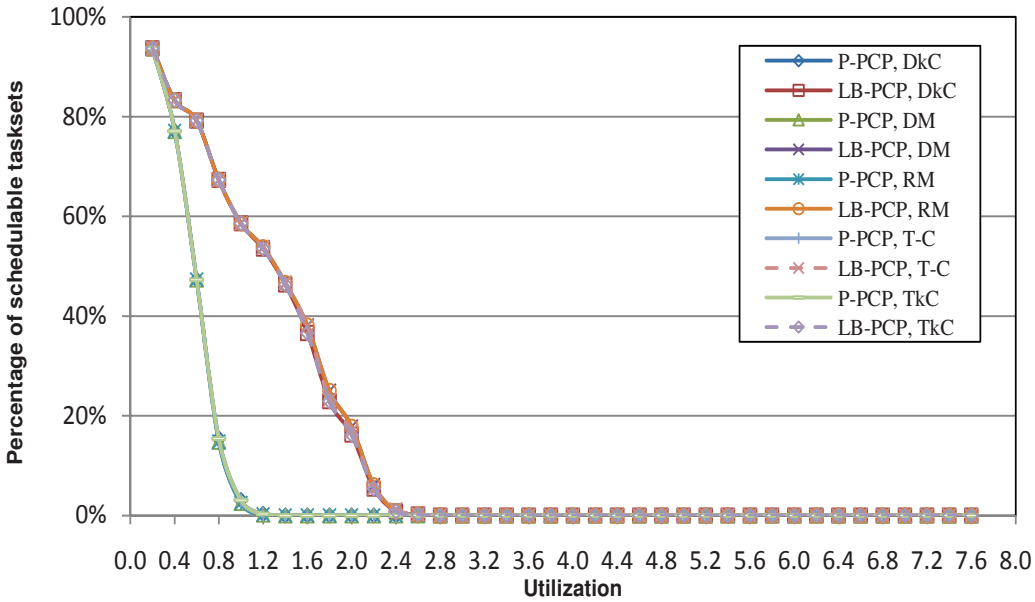


Figure 6.6: Percentage of schedulable tasksets for different priority assignment policies (m=8)

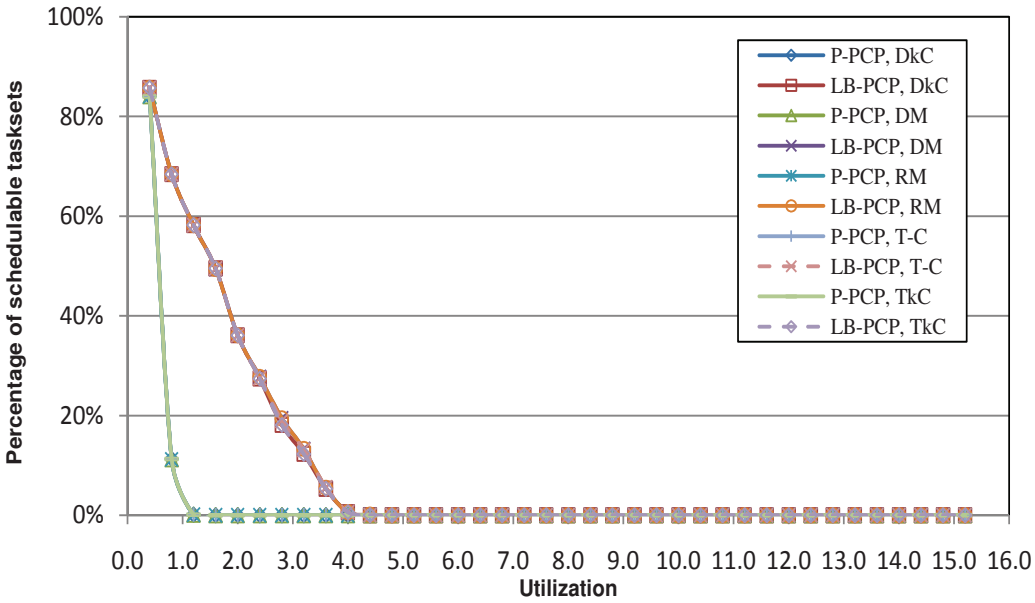


Figure 6.7: Percentage of schedulable tasksets for different priority assignment policies (m=16)

6.6.3. Experiment 2: Processor Number

With this experiment we investigate the influence of the number of processors available in the system on the number of tasksets deemed schedulable when resource sharing is following the rules of LB-PCP compared to when using P-PCP, based on the corresponding response time analyses. We assume that all shared objects are used by exactly two tasks. Figure 6.8 shows the results for 2, 4, 8, 12 and 16 processor platforms. Since in the previous experiment we saw that the priority assignment policy does not influence greatly the schedulability results, for this experiment we have chosen to use the DkC and the DM policies (a reasonable assignment according to the definition in [74]). The figure, however, presents the results only for DkC as the results for DM are similar. Also, the number of tasks was increased proportionally with the number of processors such that we always had a ratio of five tasks per processor.

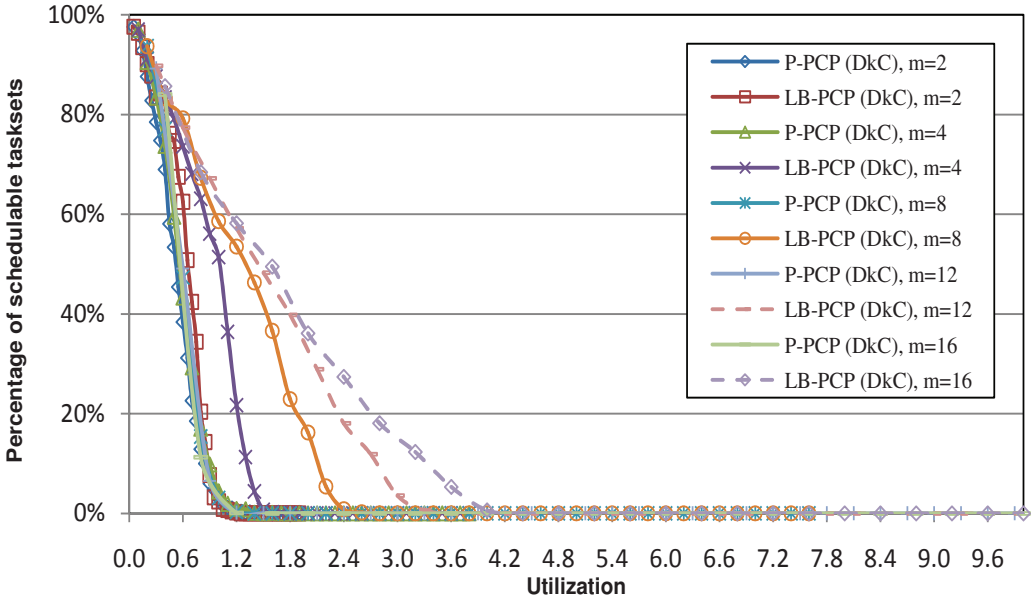


Figure 6.8: Percentage of schedulable tasksets for different number of processors

From the graph, we can see that the number of processors has an important impact on the performance of LB-PCP and also that the higher the number of processors is, the more important is the impact. This does not seem to be valid for P-PCP, which behaves almost the same, no matter the number of processors.

For 2 processors the two algorithms have quite similar performance, but as the number of processors grows, LB-PCP distances itself from P-PCP. Furthermore, in all tests almost 100% of the tasksets are unschedulable according to P-PCP at a utilization level of 1.2 and a little more than 69% of the tasksets with utilization 0.4 are schedulable when using 2 processors. There is a slight improvement of the situation for the 0.4 utilization tasksets when the number of processors increases: 73.7% for 4 processors, 77.10% for 8 processors and finally, 84% for 16 processors.

Comparatively, for LB-PCP with only 2 processors, 82.2% of the tasksets with utilization 0.4 are schedulable, although for 1.0 utilization, we have identified just 0.2% of the tasksets schedulable according to LB-PCP. However, the performance

gap between P-PCP and LB-PCP gets bigger for a greater number of processors. For $m = 4$, approximately 73.7% of the tasksets with utilization 0.4 are schedulable with P-PCP. When using LB-PCP, 73.6% of the tasksets with utilization 0.6 are schedulable. Hence, for $m = 4$ LB-PCP determines 50% better utilization of the processor than P-PCP. Moreover, P-PCP leads to 0% schedulability for a utilization of 1.4, while LB-PCP reaches 0% schedulability at 1.7 utilization.

For $m = 12$ the maximum schedulability achieved with P-PCP is approximatively 49% for 0.6 utilization. By contrast, with LB-PCP we achieve almost 78% schedulable tasksets with 0.6 utilization and reach 48% schedulability only for tasksets with 1.5 utilization. Therefore, in this scenario, LB-PCP determines 150% better utilization than P-PCP.

For 16 processor systems, in all tests 100% of the tasksets with 1.2 utilization where unschedulable under P-PCP. Under LB-PCP, the 100% unschedulability step was reached only at almost 4.0 utilization.

It is clear from the graph and from the discussions above that the difference in performance between P-PCP and LB-PCP increases also as the taskset utilization increases.

6.6.4. Experiment 3: Number of Tasks

In this experiment, we examine the effect of varying the number of tasks on the performance of LB-PCP and P-PCP. Figure 6.9 shows the percent of tasksets that were deemed schedulable on a 4 processors system, for tasksets with 16, 32, 64 and 96 tasks using the DkC priority assignment. Each task shares a single resource with another task. Figure 6.10 shows similar data for tasksets with cardinality 100, 120, 140, 180 and 200. For each configuration of taskset cardinality we increased the taskset utilization from 0.1 up to 3.8 in steps of 0.1.

There first thing to be noted in Figure 6.9 is that, while P-PCP deems less tasksets schedulable as the cardinality of the taskset increases, LB-PCP behaves exactly the opposite. However, if we look at Figure 6.10 this effect seems to diminish as under P-PCP, the number of schedulable tasksets is almost the same no matter the cardinality of the taskset. The behavior can be explained based on the following facts. With a small number of tasks, the interference from higher priority tasks is smaller in both cases. Moreover, a suspended task will have to wait for fewer tasks to finish their critical sections. As the number of tasks increases, the indirect blocking time for tasks scheduled using P-PCP will increase significantly. This is not valid also for LB-PCP, where the indirect blocking time changes only depending on effective task parameters. For a specific taskset utilization, an increase of the number of tasks in the set results in lower task utilization, which also implies lower execution requirements. Under LB-PCP the suspension time depends greatly on the worst case execution requirement of the tasks and this is why the schedulability results improve with the increasing of the taskset cardinality.

Numerically, under LB-PCP 58.1% of the tasksets with 96 tasks and utilization 0.9 are schedulable, while under P-PCP 55.5% of tasksets with utilization of only 0.3 are schedulable. Hence, for 96 tasks per taskset and 4 processors, LB-PCP determines 200% better utilization of the processor than P-PCP. For 16 tasks per taskset, this improvement diminishes. Only 51.3% of tasksets with 0.8 utilization are schedulable under LB-PCP and 53.3% of tasksets with utilization 0.5 are deemed schedulable with P-PCP. This means an improvement of only approximatively 60% in this situation.

Another thing to note is that, as the number of tasks per taskset increases, the performance gap between successive cardinality values decreases. This is more

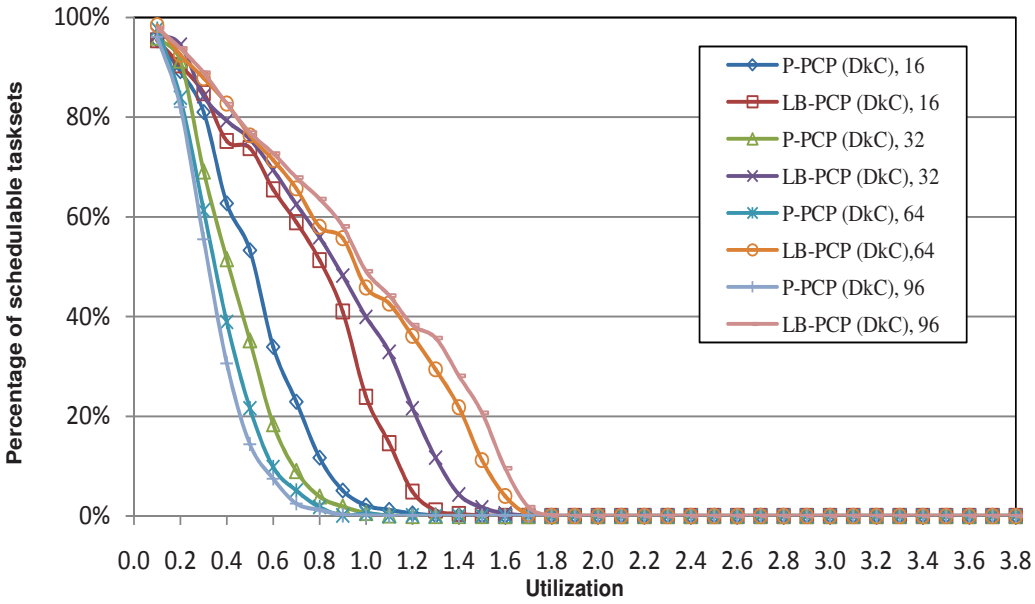


Figure 6.9: Percentage of schedulable tasksets for different number of tasks (4 processors) - taskset cardinality from 16 to 96

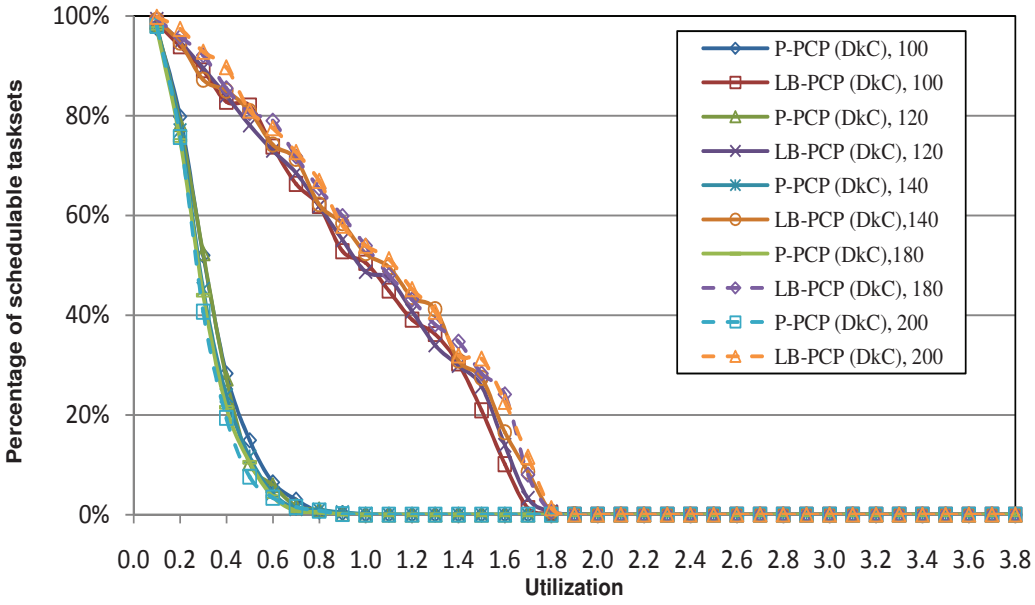


Figure 6.10: Percentage of schedulable tasksets for different number of tasks (4 processors) - taskset cardinality from 100 to 200

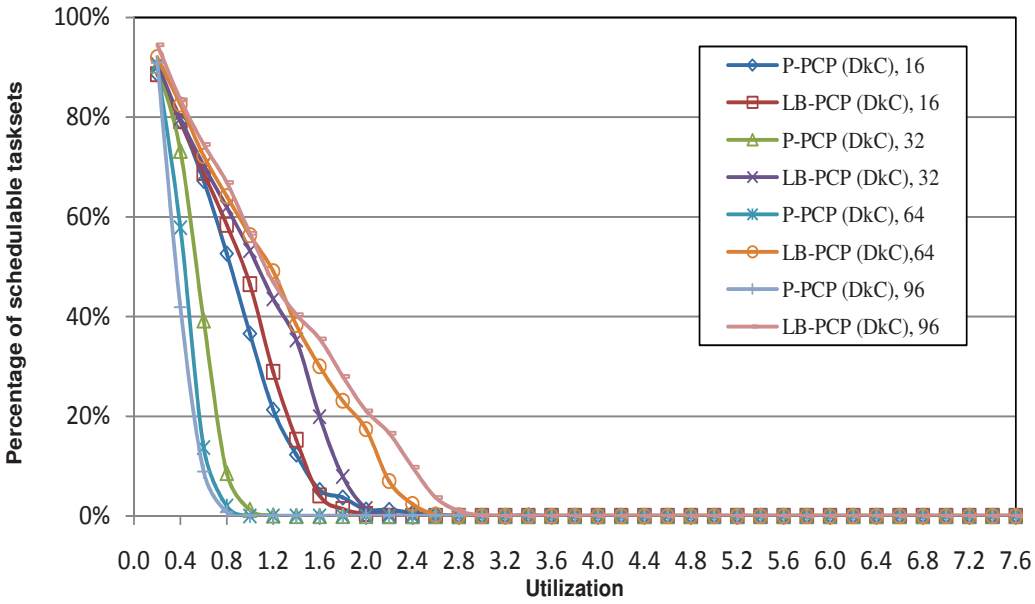


Figure 6.11: Percentage of schedulable tasksets for different number of tasks (8 processors) - taskset cardinality from 16 to 96

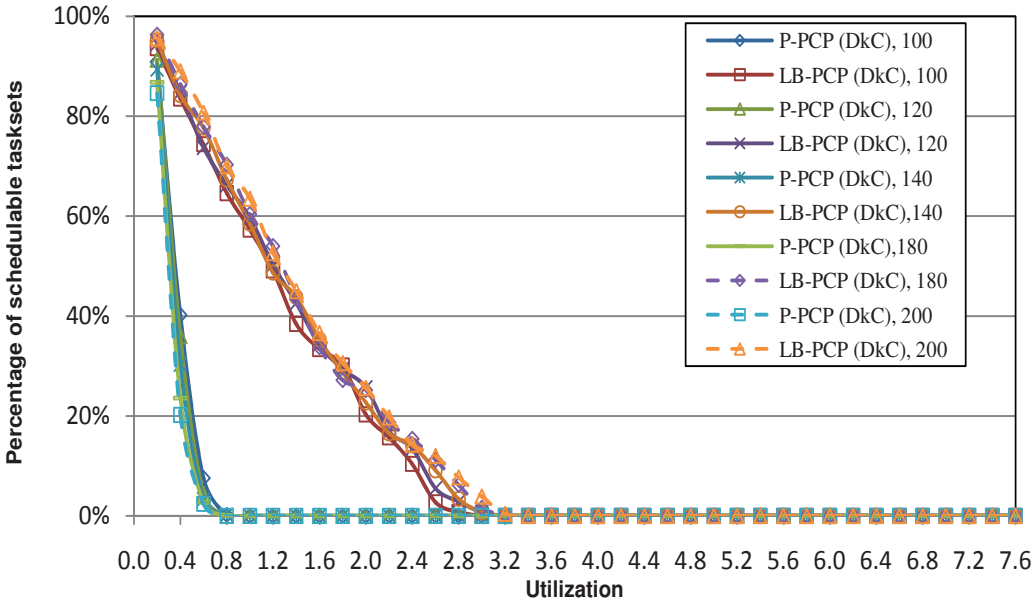


Figure 6.12: Percentage of schedulable tasksets for different number of tasks (8 processors) - taskset cardinality from 100 to 200

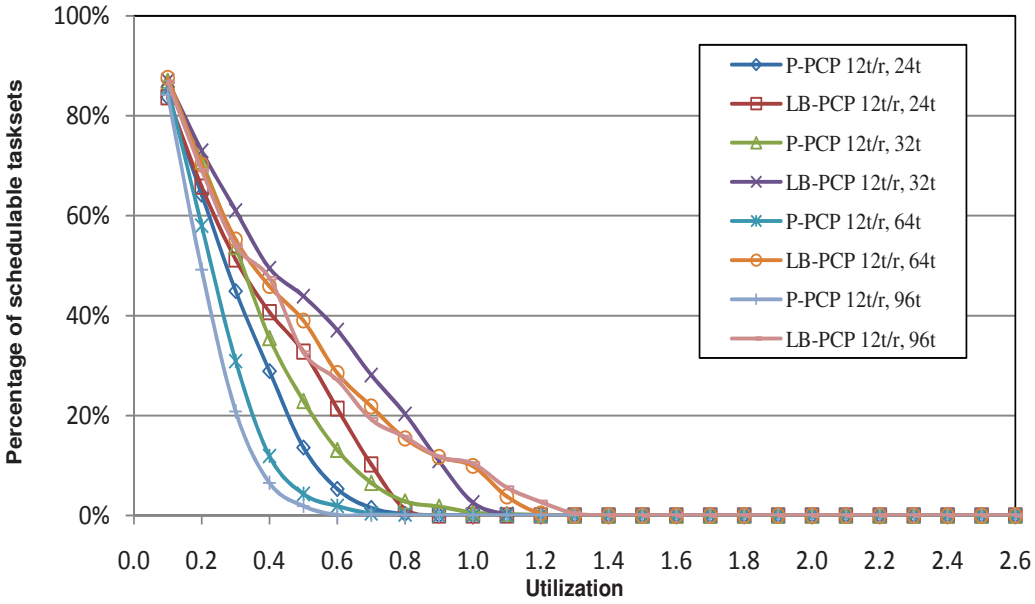


Figure 6.13: Percentage of schedulable tasksets for various taskset cardinalities with each resource being shared by 12 tasks - taskset cardinality from 24 to 96 (4 processors)

visible for LB-PCP and is the effect of the higher priority interference which increases and becomes a dominant component of the task response time.

For a 8 processor system (see Figures 6.11 and 6.12) we can also see a considerable performance gap between the protocols. Furthermore, Figure 6.11 shows that for 16 tasks per taskset, the performance of P-PCP is comparable to that of LB-PCP. This is because in P-PCP, the larger number of processors will determine a lower suspension time for half of the tasks, while the other half will not be suspended at all.

6.6.5. Experiment 4: Number of Tasks Requesting the Same Resource

In this experiment, we investigate the effect of increasing the number of tasks requesting the same resource on the performance of LB-PCP and its competitor P-PCP. For the graphs presented in this section, the priorities of the tasks were assigned using DkC. We first show what happens when we keep fixed the number of tasks using the same resource and increase the taskset cardinality. Further, for a fixed taskset cardinality, we investigate what happens when the number of tasks sharing the same resource increases. We carry all these investigations for tasksets comprised of 24 up to 96 tasks, but also for larger tasksets with 100 up to 200 tasks. For all the above test cases, we assume that 4, 8 and 16 processors platforms are used.

In the 4 processors case (Figures 6.13 and 6.14), the results for both low taskset cardinality and high taskset cardinality follow a pattern. In both cases we fixed the number of tasks sharing the same resource to be half the minimum considered cardinality, namely 12 and 50 tasks per resource. It can be noted that, in both cases and for both protocols, the percent of tasksets deemed schedulable increases as the taskset cardinality increases, but only until some limit is reached: 32 tasks

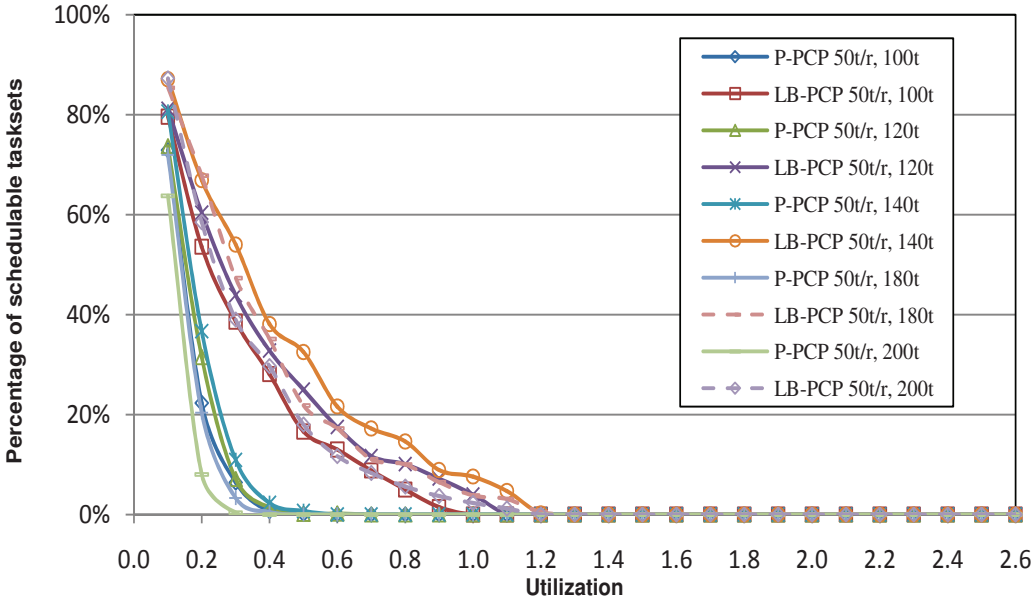


Figure 6.14: Percentage of schedulable tasksets for various taskset cardinalities with each resource being shared by 50 tasks - taskset cardinality from 100 to 200 (4 processors)

and 140 tasks, respectively. After this limit is reached the number of schedulable tasksets starts to decrease as the cardinality increases. When half of the tasks in a taskset use the same resource, the direct blocking time becomes an important term in the response time computation, while the influence of indirect blocking decreases. With LB-PCP each task will be suspended only while a part of the higher priority tasks execute critical section, while with P-PCP a task is suspended while all higher priority tasks execute their critical sections. Based on this observations and on the fact that when the number of tasks in the taskset increases, the utilization of each task decreases, we get a higher percent of schedulable tasksets for 32 or 140 tasks per taskset. However, as we increase the number of tasks further, although the task utilization decreases, the interference from both lower and higher priority tasks using shared resources increases and fewer tasks manage to meet their deadlines.

The same pattern can be noticed in Figures 6.15 and 6.16 when a 8-processor platform is used. Again, as in the previous experiments, it can be noted that for 8 processors, LB-PCP gives better performance than when 4 processors are used. This performance improvement is not as significant for P-PCP, for which, for 200 tasks and very low taskset utilization (0.2) the percentage of successfully scheduled tasksets increases to 84.6% while for 4 processors it was approximatively 76%.

In the second phase of this experiment we keep the taskset cardinality fixed and vary the number of tasks using the same resource. Figure 6.17 shows the results of the experiment for a 4-processor platform with 24 tasks per taskset. Figure 6.18 depicts the situation for tasksets of 200 tasks. The number of tasks using the same resource is increased from a ratio of $\frac{1}{12}$ up to $\frac{1}{2}$ of the taskset cardinality. It can be observed that for both LB-PCP and P-PCP, the percentage of schedulable tasksets decreases as the number of tasks sharing the same resource increases. A large ratio leads to more tasksets deemed unschedulable. For example, in the test with half of

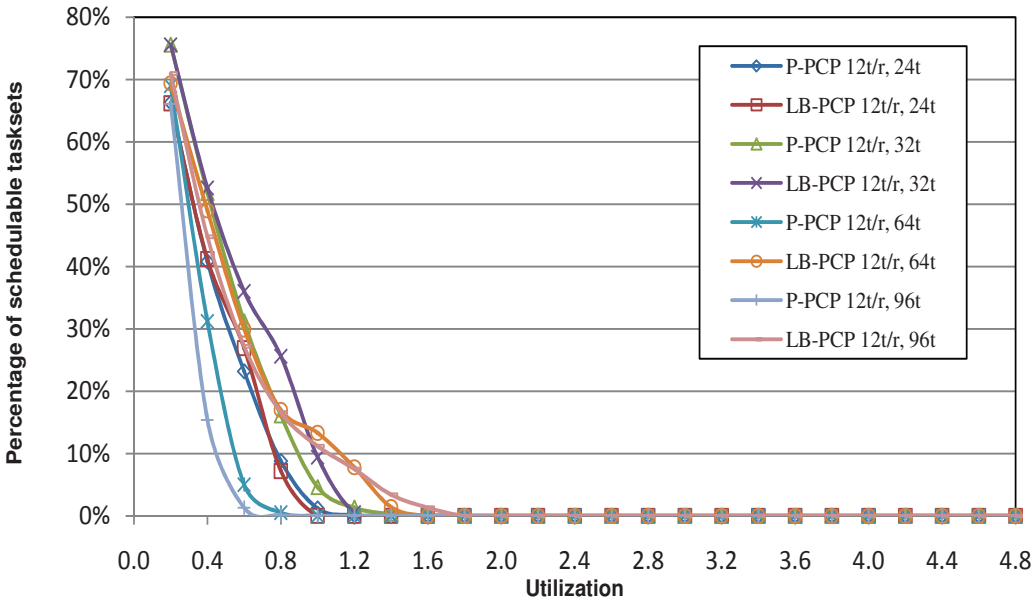


Figure 6.15: Percentage of schedulable tasksets for various taskset cardinalities with each resource being shared by 12 tasks - taskset cardinality from 24 to 96 (8 processors)

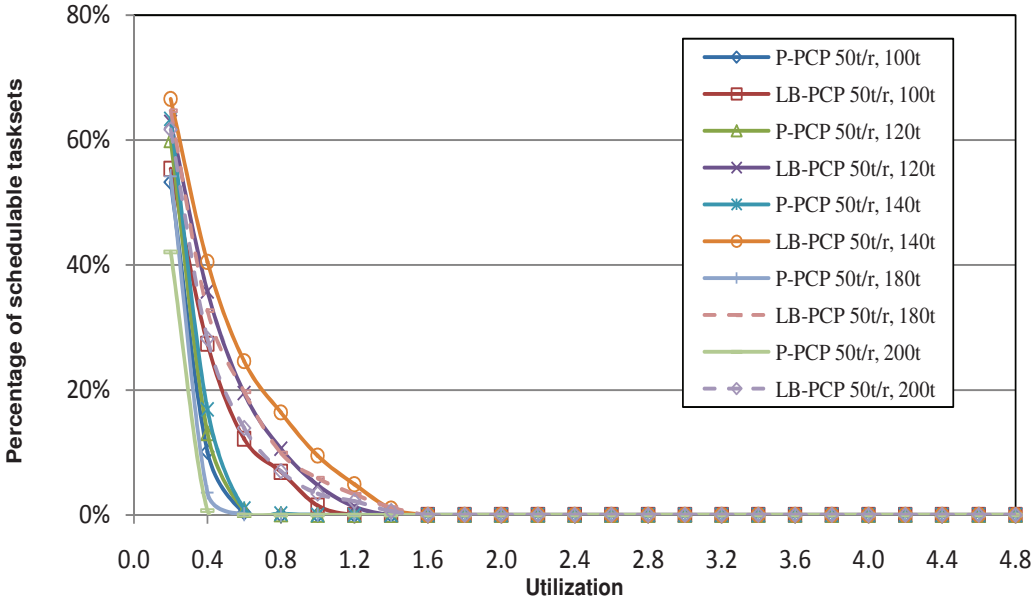


Figure 6.16: Percentage of schedulable tasksets for various taskset cardinalities with each resource being shared by 50 tasks - taskset cardinality from 100 to 200 (8 processors)

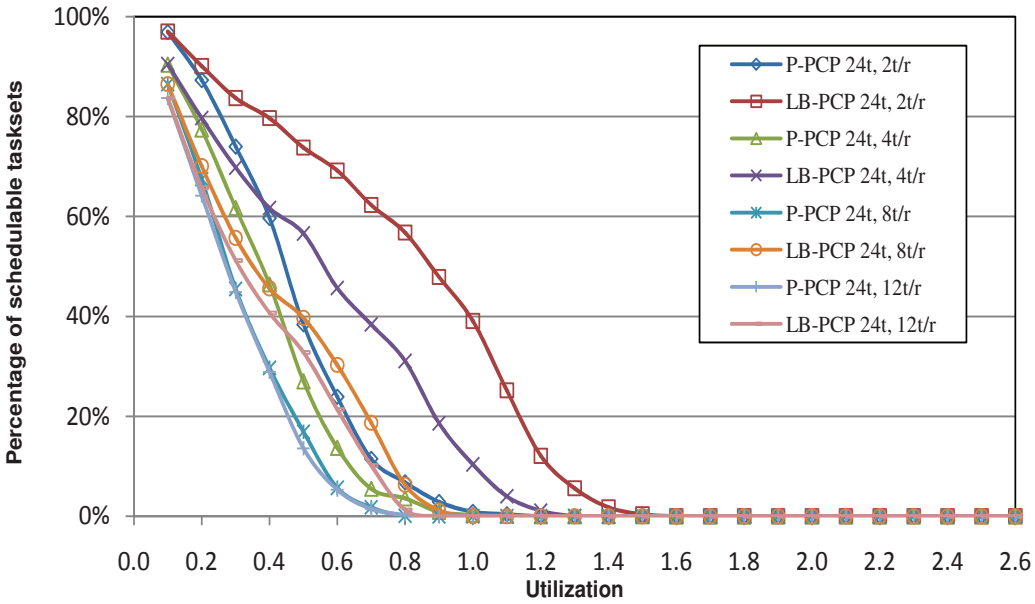


Figure 6.17: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 24 (4 processors)

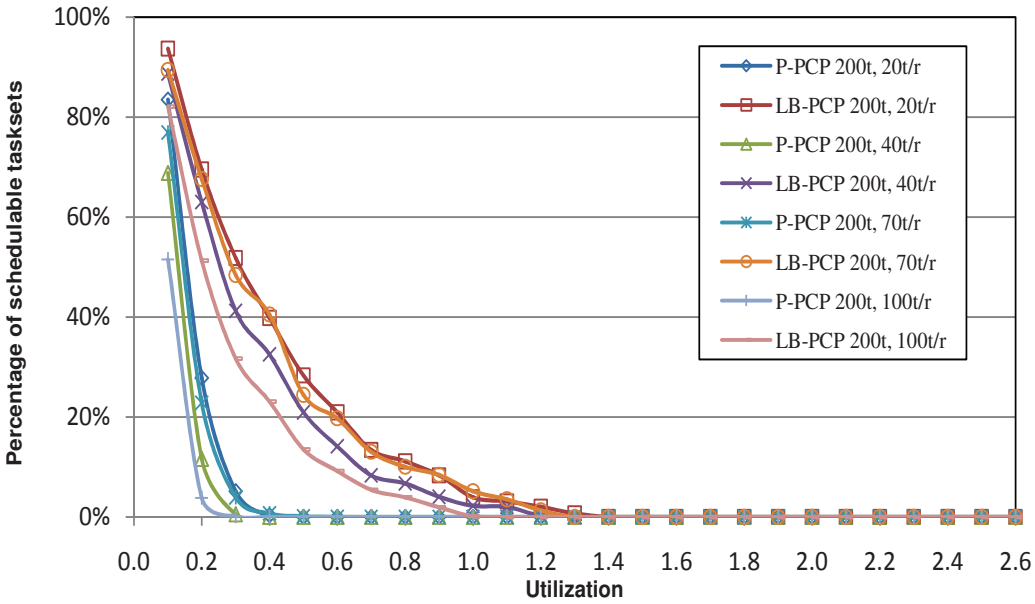


Figure 6.18: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 200 (4 processors)

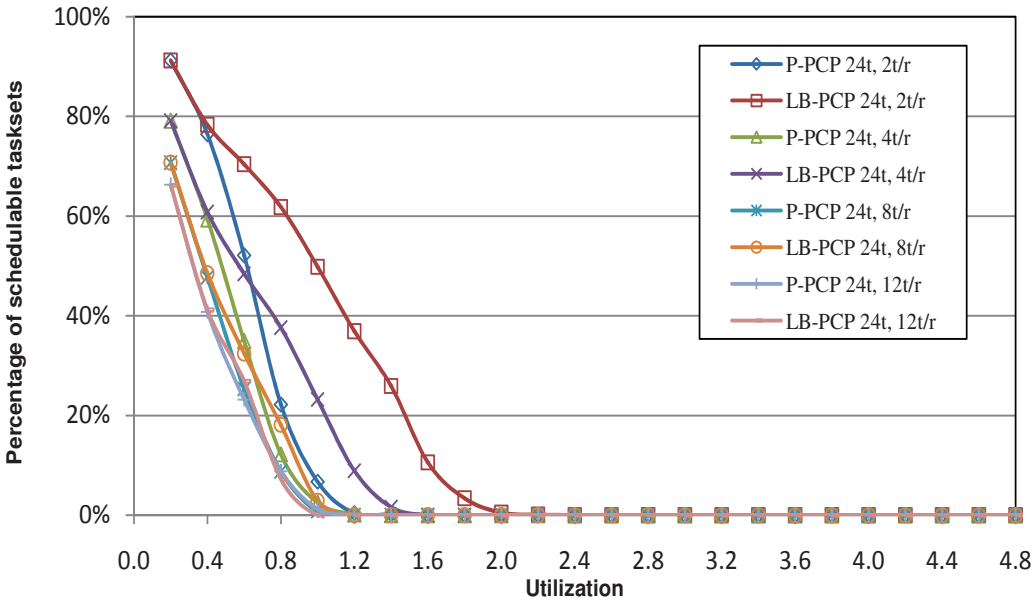


Figure 6.19: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 24 (8 processors)

the tasks sharing the same resource, for LB-PCP approximatively 11% of the tasksets with 24 tasks are schedulable at utilization level 0.7. For the same test case, with P-PCP, 13% of the tasksets with utilization 0.5 are schedulable. Therefore, in this test case, for LB-PCP we get a 40% performance gain. If the ratio decreases such that $\frac{1}{12}$ of the tasks use the same resource, the performance gap between the two algorithms increases. With LB-PCP, 57% of the tasksets with 24 tasks and utilization 0.8 are schedulable, while with P-PCP the same percent is achieved only for tasksets with utilization 0.4, which means a performance gain of 100% with LB-PCP. For large tasksets, the performance difference between the two protocols is maintained. Moreover, for a large number of tasks sharing a resource, the performance of P-PCP drops rapidly as we increase the utilization of the tasksets with 0% schedulability already at 0.3 utilization, while with LB-PCP we reach 0% schedulability only at 1.0 utilization (230% better utilization).

However, for small tasksets, as we increase the number of processors to 8 and 16 we note that the performance difference decreases and even inverts (for 16 processors). We can see in Figure 6.19, when half of the tasks use the same resource in an 8-processor system, LB-PCP gives the same schedulability results as P-PCP. For the test with only 2 tasks sharing the same resource, with P-PCP 52% of the tasksets with utilization 0.6 are schedulable and for LB-PCP 50% of tasksets with utilization 1.0 are schedulable which means almost 70% performance gain with LB-PCP. Thus, for 8 processor the performance improvement is maintained as long as the number of tasks using the same resource is small and drops when we increase this number. For 16 processors (see Figure 6.21), P-PCP proves to be better than LB-PCP no matter the ratio of tasks sharing the same resource. By increasing the number of processors to a value closer to the taskset cardinality, the chances for a task to be indirectly blocked decrease. In P-PCP the α configuration parameter of P-PCP will also increase

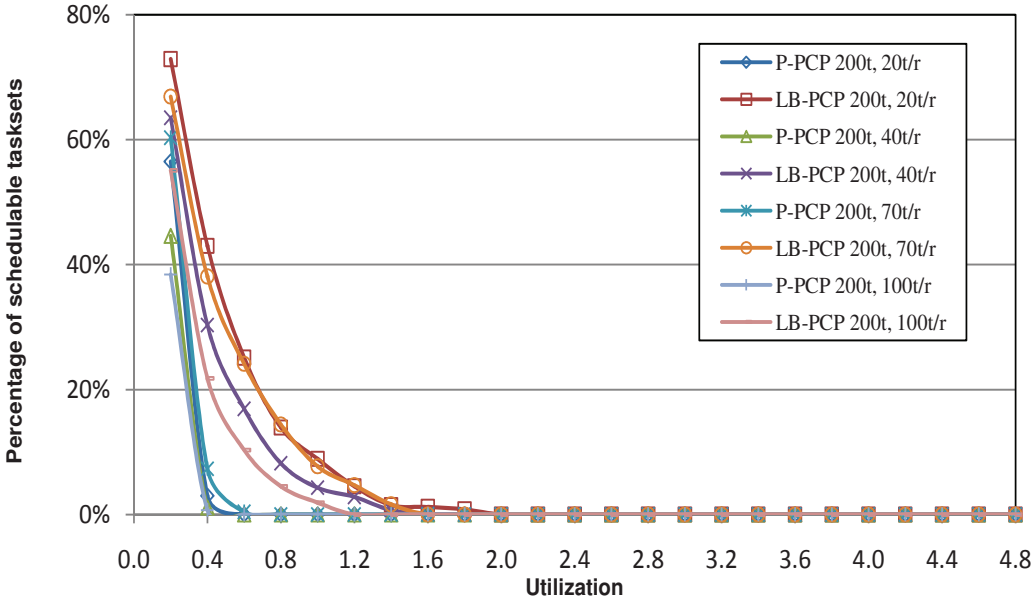


Figure 6.20: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 200 (8 processors)

(remember that for tasks with priority lower or equal to the number of processors, α equals to n and with m for the rest of the cases) and lower priority tasks will be seldom suspended, where the duration of the suspension will decrease proportionally with the increase in processor number. On the other side, in LB-PCP the suspension time has a component not influenced by the number of processors, namely the last term in the right hand side of Equation (6.12).

The situation is not the same for large tasksets. For 200 tasks per taskset running on a 8 (Figure 6.20) or 16 (Figure 6.22) processor platform, LB-PCP is always better than P-PCP.

6.6.6. Experiment 5: Multiple Resource Accesses per Task

In the previous experiments we assume each task uses a single resource. In this experiment, we investigate the effects of doubling the number of resources accessed by a task. Therefore, each task in a taskset uses two resources for a total amount equal to its execution requirement. Each resource is shared by two or more tasks. Like in experiment 4, we first show what happens if we keep fix the number of tasks sharing the same resource and increase the taskset cardinality and then, we investigate the effects of increasing the number of tasks sharing a resource for small and large taskset cardinalities. The results presented here use DkC for assigning task priorities.

In Figure 6.23, we can note that for P-PCP and 4 processors, the pattern identified in the previous experiment is still present: the schedulability increases until taskset cardinality reaches 32 tasks and decreases for larger cardinality values. For LB-PCP, this effect occurs only for small taskset utilizations. For utilization values greater than 1.0, the schedulability for tasksets with 64 and 96 tasks is greater than for the ones with 32 tasks. This is the effect of the fact that larger taskset utilization

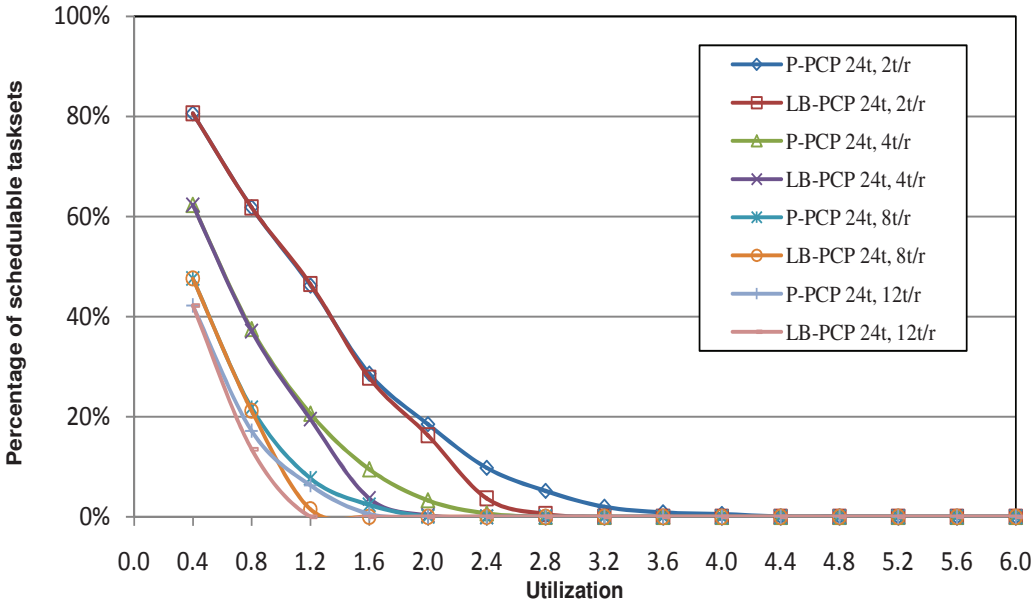


Figure 6.21: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 24 (16 processors)

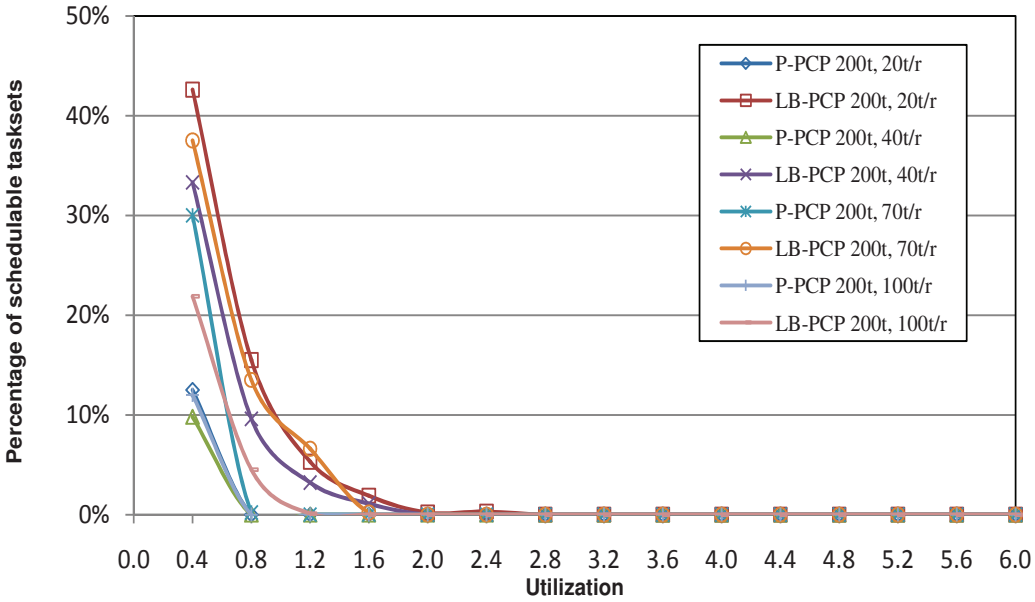


Figure 6.22: Percentage of schedulable tasksets for various number of tasks requesting the same resource - taskset cardinality 200 (16 processors)

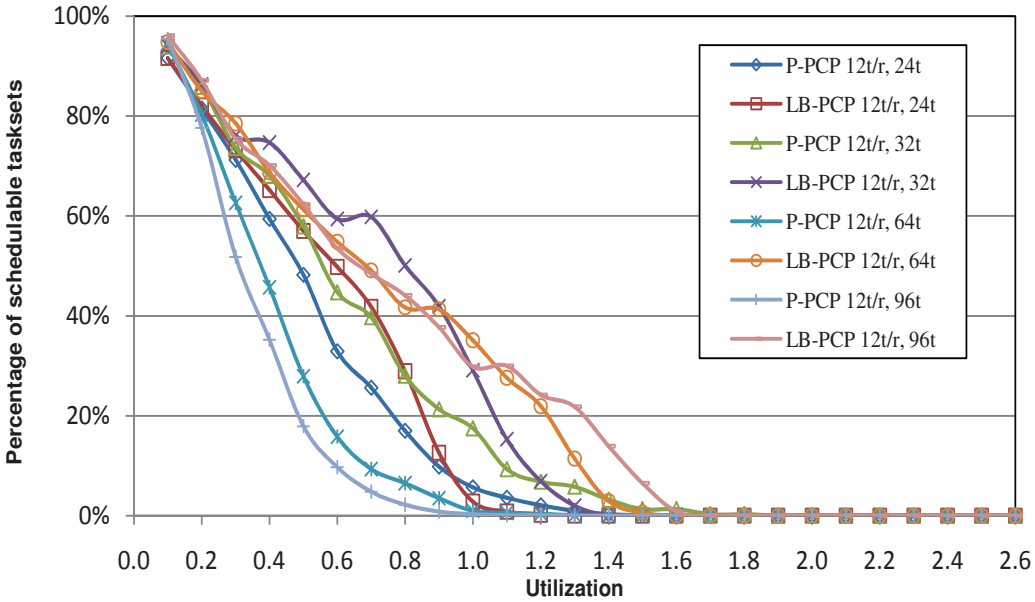


Figure 6.23: Percentage of schedulable tasksets for various taskset cardinalities with each task using two resources and each resource being shared by 12 tasks - taskset cardinality from 24 to 96 (4 processors)

for identical taskset cardinality means smaller individual task utilization.

Smaller task utilization may be obtained by reducing the execution time and therefore the length of the critical sections. This results also in reduced suspension time. Since each task uses two resources, it means that it can get suspended twice and hence, the higher the suspension time, the higher the chances for the task to miss its deadline. However, for very large tasksets (see Figure 6.24), the utilization threshold is much greater than 1.0 and since taskset schedulability decreases faster, we cannot make the same observation.

The same pattern is noticed in Figures 6.25 and 6.26 for an 8 processor system. Furthermore, we can see that for 8 processors, as taskset utilization increases, P-PCP outperforms LB-PCP for taskset with low cardinality (24 and 32). This is caused by the higher suspension time in case of LB-PCP. Also, it can be noted that for 8 processors, both LB-PCP and P-PCP give better performance than when 4 processors are used.

In the second phase of this experiment, we keep the taskset cardinality fixed and vary the number of tasks using the same resource. Figure 6.27 shows the results of the experiment for a 4-processor platform with 24 tasks per taskset. Figure 6.28 depicts the situation for tasksets of 200 tasks. The number of tasks using the same resource is increased from a ratio of $\frac{1}{12}$ up to $\frac{1}{2}$ of the taskset cardinality. It can be observed that for both LB-PCP and P-PCP, the percentage of schedulable tasksets decreases as the number of tasks sharing the same resource increases. A large ratio leads to more tasksets deemed unschedulable.

In general, LB-PCP performs better than P-PCP, but for small tasksets with high utilization and large ratios of *task cardinality/tasks using the same resource*, P-PCP is slightly better than LB-PCP. For large tasksets however, LB-PCP always outperforms LB-PCP.

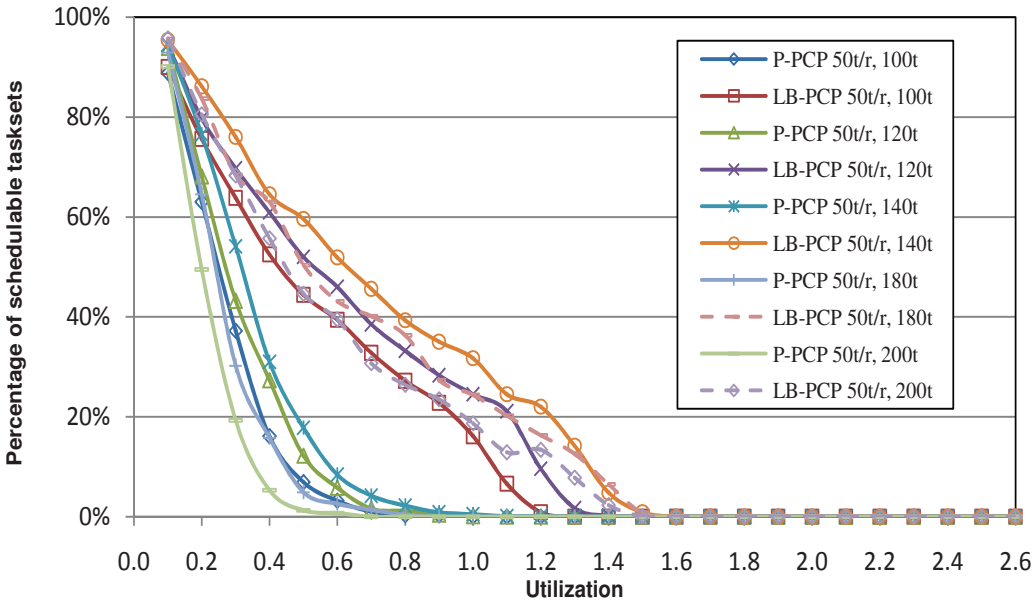


Figure 6.24: Percentage of schedulable tasksets for various taskset cardinalities with each task using two resources and each resource being shared by 50 tasks - taskset cardinality from 100 to 200 (4 processors)

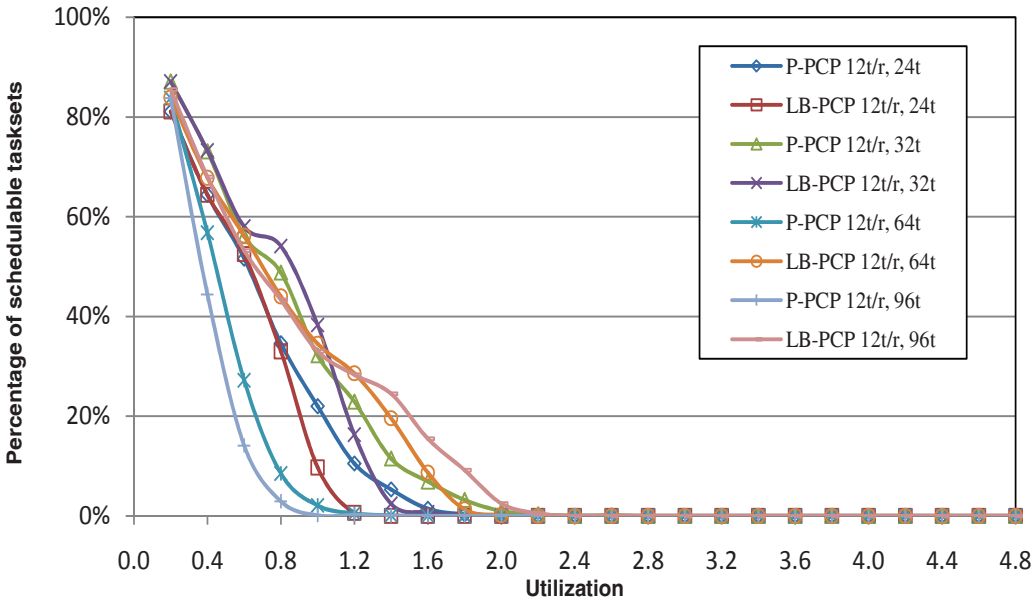


Figure 6.25: Percentage of schedulable tasksets for various taskset cardinalities with each task using two resources and each resource being shared by 12 tasks - taskset cardinality from 24 to 96 (8 processors)

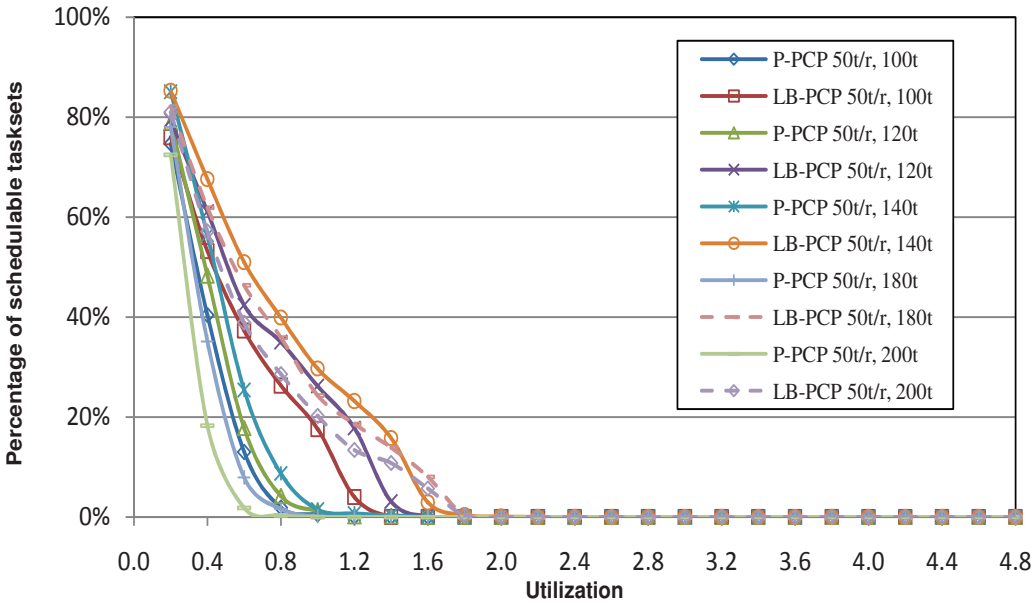


Figure 6.26: Percentage of schedulable tasksets for various taskset cardinalities with each task using two resources and each resource being shared by 50 tasks - taskset cardinality from 100 to 200 (8 processors)

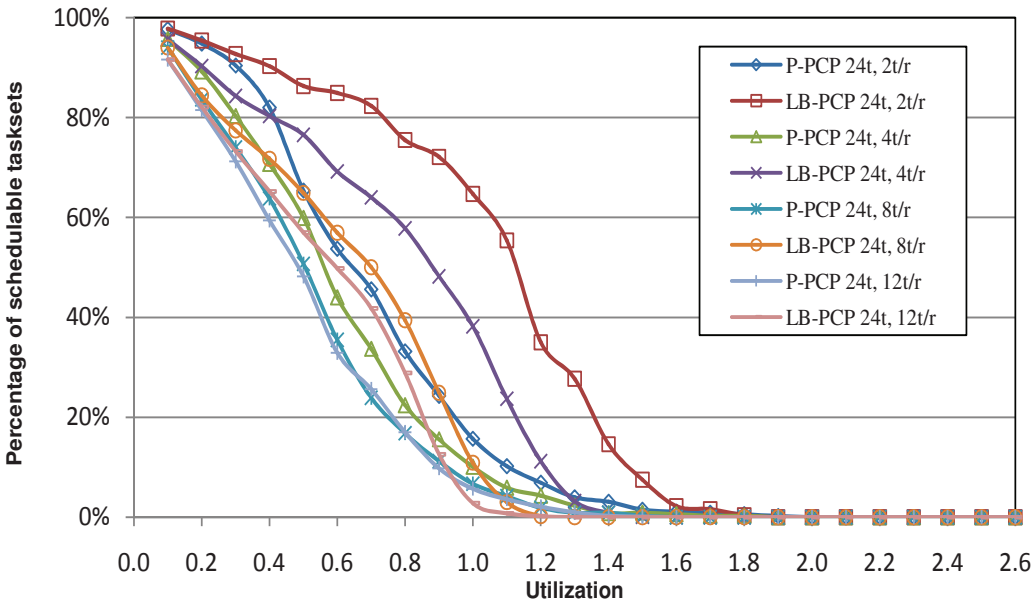


Figure 6.27: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 24 (4 processors)

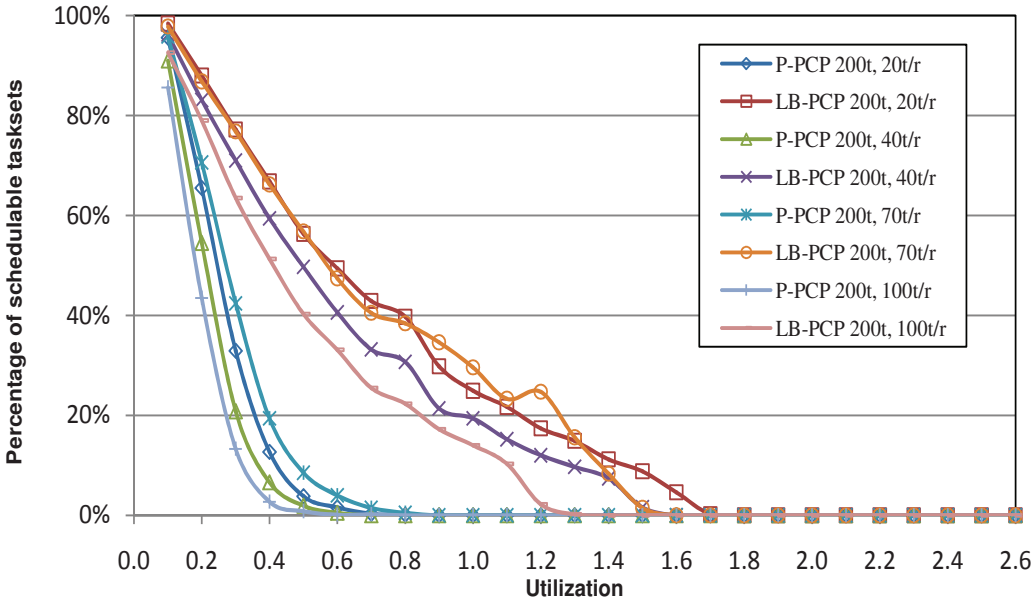


Figure 6.28: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 200 (4 processors)

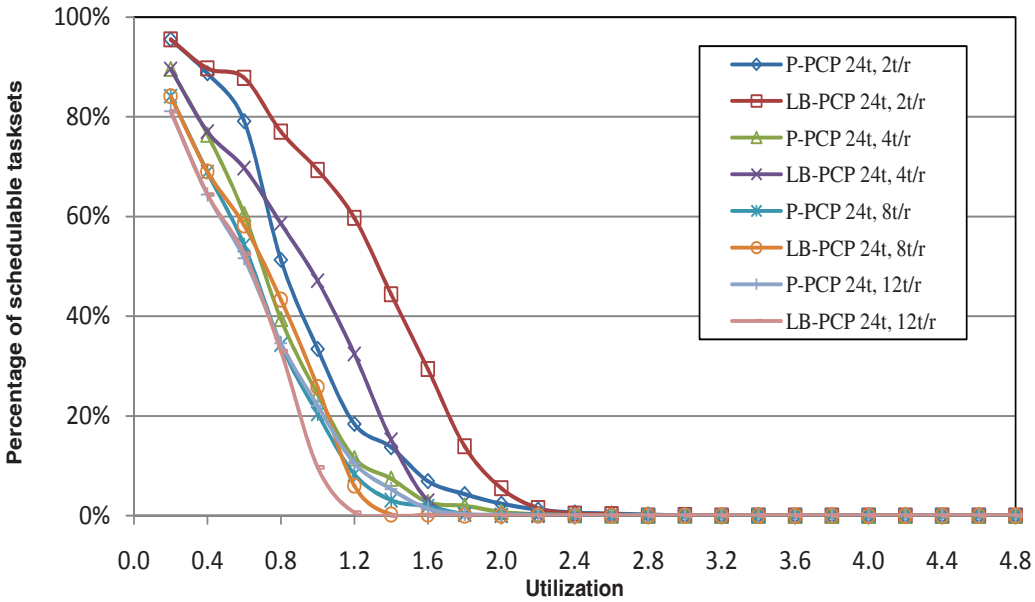


Figure 6.29: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 24 (8 processors)

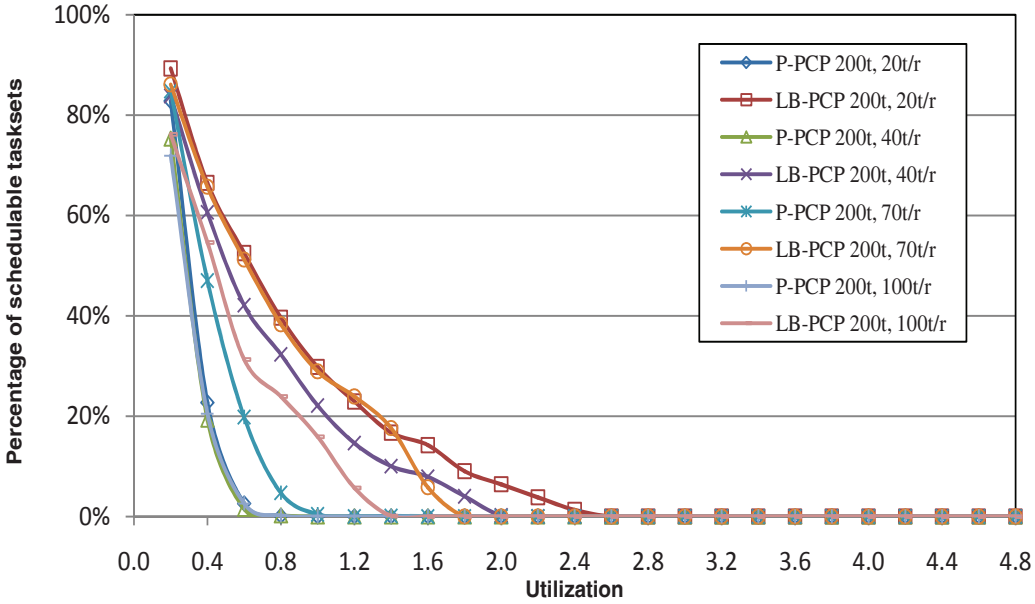


Figure 6.30: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 200 (8 processors)

For example, in the test with half of the tasks sharing the same resource, for LB-PCP, 14% of the tasksets with 200 tasks are schedulable at utilization level 1.0. For the same test case, with P-PCP, 13% of the tasksets with utilization 0.3 are schedulable. Therefore, in this test case, for LB-PCP we get a 233% performance gain. If the ratio decreases such that approximately $\frac{1}{12}$ of the tasks use the same resource, the performance gap between the two algorithms is maintained. With LB-PCP, we get 0% schedulability for tasksets with 200 tasks at about 1.7 utilization, while with P-PCP the same percent is achieved already for tasksets with utilization 0.9, which means a performance gain of 88% with LB-PCP.

Again, we note that, for small tasksets, as we increase the number of processors to 8 and 16 the performance difference decreases and even inverts (for 16 processors). We can see in Figure 6.29 that, in a 8 processor system, only if the number of tasks sharing the same resource is small, LB-PCP behaves significantly better than P-PCP. For the test with 24 tasks per taskset and only 2 tasks sharing the same resource, with P-PCP 33% of the tasksets with utilization 1.0 are schedulable and for LB-PCP 30% of tasksets with utilization 1.6 are schedulable which means almost 60% performance gain with LB-PCP. Thus, for 8 processor the performance improvement is maintained as long as the number of tasks using the same resource is small and drops when we increase this number. For large tasksets, LB-PCP continues to dominate P-PCP in all test cases (see Figure 6.30).

For 16 processors (see Figure 6.31), P-PCP proves to be better than LB-PCP no matter the ratio of tasks sharing the same resource. Again, this is because, for low cardinality tasksets, a high number of processors means less chances of indirect blocking and, with P-PCP lower priority tasks will be suspended for less time. On the other side, in LB-PCP the suspension time has a component not influenced by the number of processors, namely the last term in the right hand side of Equation (6.12).

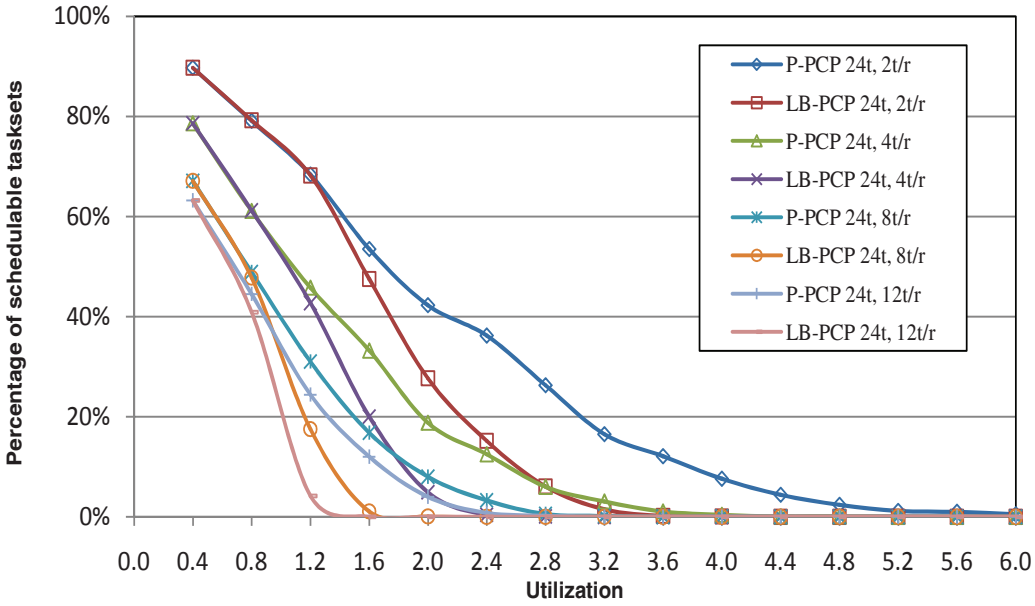


Figure 6.31: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 24 (16 processors)

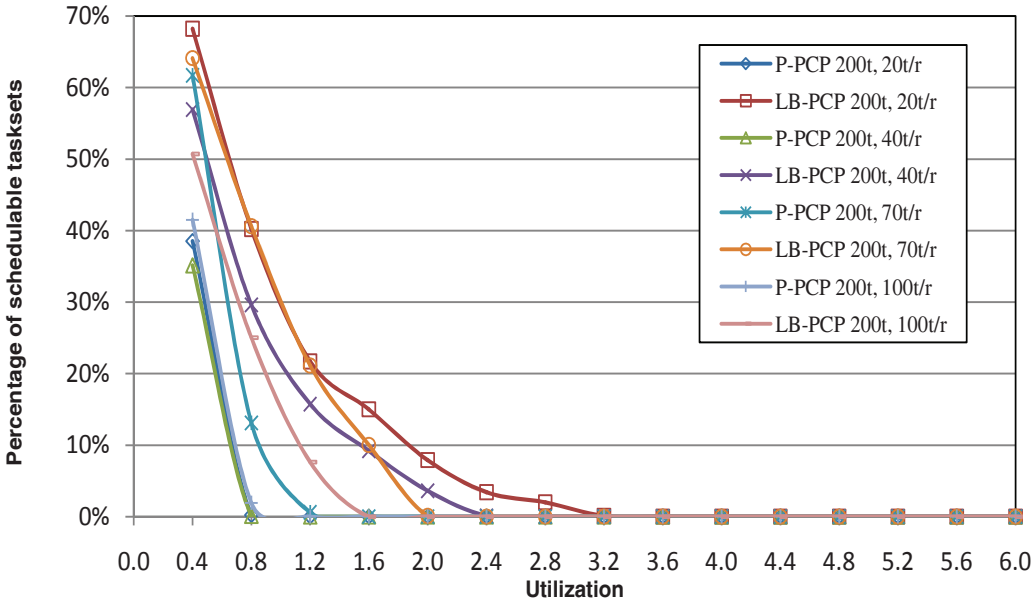


Figure 6.32: Percentage of schedulable tasksets for various number of tasks requesting the same resource and each task using two resources - taskset cardinality 200 (16 processors)

For large tasksets and 16 processors, again LB-PCP performs better than P-PCP

If we look back at experiment 4, we can see that the situations in which LB-PCP performs worse than P-PCP are quite similar. From this we can conclude that LB-PCP behaves well even when each task uses several resources and can be used with large and medium sized critical sections, but we expect that, for very short resource access, P-PCP will outperform LB-PCP.

6.6.7. Experiments Summary

In previous sections we presented the results of a set of experiments performed in order to give some insight on the performance of LB-PCP and, at the same time, we compared it with one of the most recent protocols proposed for a similar system setup. Based on all these experiments, we can draw several conclusions:

- The performance of LB-PCP, as well as that of P-PCP, is not influenced significantly by the priority assignment policy.
- If a small number of tasks share the same resource for a large amount of time, LB-PCP outperforms (i.e. deems more tasksets schedulable) P-PCP always and the performance gap between the two increases as the number of processors increases.
- For small task systems with high resource contention values, LB-PCP performs better than P-PCP only if the number of processors is small and its performance decreases as the number of processors increases.
- For large task systems with high resource contention values, LB-PCP performs always better than P-PCP.
- For large tasksets, LB-PCP will deem more tasksets schedulable than P-PCP, no matter the number of available processors, resources shared by each task or number of tasks sharing each resource.

6.7. Concluding Remarks

This chapter introduced the Limited Blocking Priority Ceiling Protocol, a resource sharing protocol suitable for multiprocessor real-time systems scheduled with a global preemptive fixed-priority algorithm. We have shown in Chapter 2. that, until recently, resource sharing in multiprocessor systems focused more on partitioned scheduling algorithms, and only recently, the interest for designing resource sharing protocols for globally scheduled systems has aroused.

The motivation of our work was to improve upon the current state-of-the-art in resource sharing protocols for global multiprocessor real-time scheduling. We started from the major limitation in applying PIP directly for multiprocessors: the large interference of low priority jobs on the higher priority ones. The intuition behind our work was the idea that, by finding a way to control this high interference we could increase the benefit of parallel processing provided by the multiprocessors.

The contributions of this chapter are focused on the following areas:

c1. We introduce the Limited Blocking Priority Ceiling Protocol (LB-PCP), a protocol for mutual exclusive resource sharing under globally scheduled real-time multiprocessor systems. The protocol can control the interference on high priority

tasks from lower priority tasks no matter what priority assignment policy is used and does not restrict the ability to exploit the parallelism provided by the multiprocessor platform.

c2. We give a schedulability test for LB-PCP. The test allows one to check off-line if a given set of tasks will satisfy all its timing constraints. Based on this schedulability test, we compare the performance of LB-PCP with the performance of P-PCP, the only known global multiprocessor resource sharing protocol.

7. A RESOURCE SHARING PROTOCOL FOR VIRTUALIZED MULTI-CORE SYSTEMS

7.1. Motivation

Currently, real-time research puts considerable effort into designing and implementing HSFs. Hierarchical scheduling is the mean to enforce real-time guarantees in virtualized systems. Such hierarchical frameworks allow building systems from independently designed, implemented and validated applications, which must execute concurrently on a shared platform. A key requirement for these systems is that any application that meets its timing constraints when running in isolation, must also meet its timing constraints when running on the shared platform. The basic HSF assumes each application has a dedicated scheduler for its tasks. The processing capacity required to schedule these tasks and the timing constraints of the application are specified by the so-called *service contract* [95] or by the *interface* [153] of the application. Based on the service contract, a separate execution server is allocated to each application, a technique which enables dividing the processor capacity among all applications. Further, each of the servers allocated to the set of applications in the system is seen by a system-level scheduler as a task with a unique priority while, at the next level the tasks of the same application are scheduled by the dedicated scheduler according to an application-specific algorithm.

Until recently, the real-time research focused more on HSFs running on a processing platform that consists of a single-core processor. The work on extending these frameworks to multi-core architectures is still in an incipient phase. Such extensions must allow each application to run its tasks in parallel which implies that more that one server may need to be used by each application. Regarding this scenario, a single schedulability test is known in the literature [60], but this test is limited to applications consisting of completely independent tasks. For independent tasks or tasks with precedence dependency, [126] and [127] propose schedulability analysis methods based on formal verification with timed automata. Related to the same issue of multi-core HSFs, finding the application interface or service contract such that application parallel execution is enabled, also dealt only with independent tasks [77, 125].

Moreover, the little work on mutually exclusive resource sharing in HSFs uses only one server per application [138, 137, 139] or task [78] and therefore it is not possible for the tasks of the application to execute in parallel. Our goal in this chapter is to propose a synchronization protocol that can be used in multi-core systems without restricting application-level parallelism and that can be applied when schedulers at all levels of the hierarchy use a global multiprocessor scheduling approach. Furthermore, in order to enable the evaluation of the proposed protocol, an extended schedulability test based on response time analysis is also proposed.

Motivational example. Consider the mine pump problem introduced by Burns and Wellings [54]. A mine has several sensors which control a pump pumping water out of the mine and a sensor to monitor the methane level in the mine. A schematic diagram of the system is given in the left side of Figure 7.1. Two sensors indicate the water level. When the water level is high, the pump is switched on to pump the water out of the mine. When the water level is low, the pump is switched

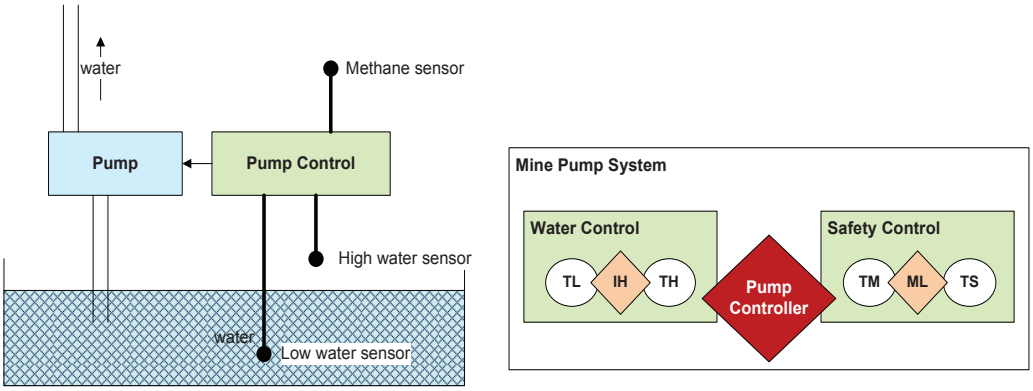


Figure 7.1: The mine pump system

off. Another sensor is used for monitoring the methane level in the mine. If the methane level is above a certain threshold, the pump must not operate.

The scheduling hierarchy of the mine pump system is depicted in the right side of Figure 7.1. The water level sensors raise interrupts handled by two sporadic tasks, *TH* and *TL*, belonging to application *Water Control*. As in [54], the interrupt handling routines are provided by an object *IH*, shared by the two tasks. The interrupt handler is therefore a locally shared resource. The application *Safety Control* is responsible of handling events coming from the methane sensor. A task *TM* of this application periodically pulls the methane sensor. If the methane level is higher than the threshold, the task stops the pump. Another periodic task *TS* supervises the pump for safety purposes, stopping and starting the pump according to the current value of the methane reading. Both tasks will use a locally shared data item ML representing the value of the last reading of the methane sensor. The pump is controlled by the *Pump Controller* object. This object provides operations for starting and stopping the pump and therefore it will be accessed both by tasks of application *Water Control* and by tasks of the *Safety Control* application.

7.2. Hierarchical System Model

In this chapter, we focus on the problem of scheduling multiple real-time applications using a multiprocessor hierarchical scheduling framework. Each application consists of a set of real-time tasks, $\mathcal{T} = \{\tau_1, .., \tau_n\}$, which are released either periodically at fixed time intervals, or sporadically with a minimum inter-arrival time between two successive releases. We use τ_i^j to denote the j^{th} job of task τ_i. The applications are scheduled on a multi-core processor or a multiprocessor platform comprised of m identical processors. In order to enable isolation between different applications, each application is associated with a *set of servers* and its tasks execute within the capacity of the corresponding servers. By associating a set of servers with each application instead of just one server, we enable parallel execution of the application's tasks.

There are two levels of scheduling in the system. At the first level, a scheduling policy selects globally, among all server sets, the m servers which will access the m processors at any time. At the next level, an application-specific scheduling policy selects the tasks that will execute on the associated servers (see Figure 7.2). It is

assumed that both scheduling levels employ global fixed-priority preemptive policies, i.e. a server is not assigned to any specific processor and, similarly, a task is not assigned to any specific server in the application's associated set.

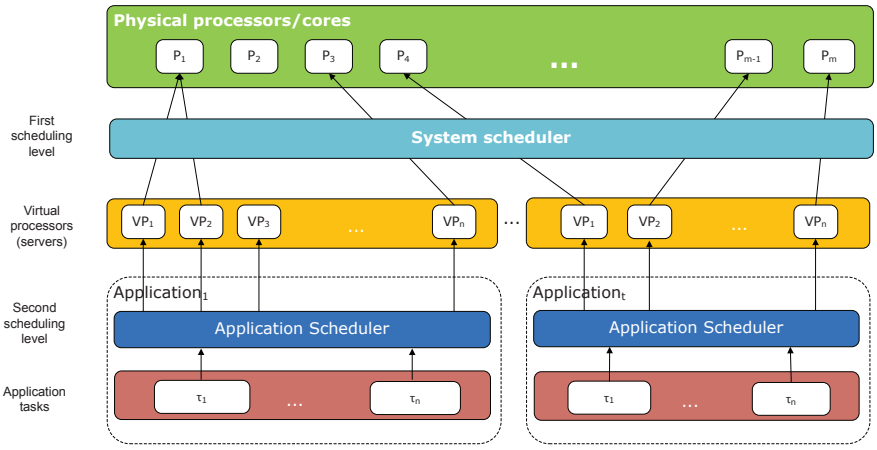


Figure 7.2: A two-level hierarchical scheduling framework

Each server σ_s $(1 \leq s \leq p)$ can be characterized by a unique priority s within all servers in the system, a *capacity* or *budget* C_s and a *replenishment period* T_s. The capacity of the server represents the maximum amount of computation time the server can normally provide in one invocation. The replenishment period is the time before the server's capacity will be again fully available. Without loss of generality, we assume that servers are also sorted in decreasing order of their priorities. Note also that a server cannot be scheduled on more than one processor at any given time instant.

Furthermore, the system model is also based on the following assumptions, representing an extension of the model introduced in Section 4.2.1.:

a1. Each application task τ_i $(1 \leq i \leq n)$ is characterized by a tuple (C_i, T_i, D_i), where T_i denotes the minimum inter-arrival time (or *period*) between two successive jobs of the task, C_i its *worst-case execution time* and D_i its *relative deadline*.

a2. All tasks of each application are periodic or sporadic.

a3. We consider only *constrained-deadline* tasks with $D_i \leq T_i$, $\forall i$.

a4. Each task τ_i has a unique *base priority* equal to i.

a5. Tasks are sorted in decreasing order of their priority, i.e. for every pair (τ_i, τ_l) if $i < l$ then the priority of τ_i is higher than the priority of τ_l.

a6. During the execution of a job τ_i^j its priority may be increased at level h. We call this elevated priority level *effective priority*.

a7. The priorities of the tasks and servers are assigned statically.

a8. A task cannot execute on two or more cores simultaneously, and a core cannot execute two or more tasks simultaneously.

a9. Task and server preemption is permitted at any time.

a10. Task and server migration is allowed at any time.

a11. It is considered that the costs of preemption and migration are included in the task execution time.

a12. Besides the processor, the tasks of each application can also share a set of resources which can be accessed only non-preemptively, in mutual exclusive manner.

a13. Different applications can share resources which can be accessed only non-preemptively, in mutual exclusive manner.

We consider applications scheduled under simple *periodic server* sets although the analysis can be extended for other execution time servers algorithms like *Deferrable Server* [156] or *Sporadic Server* [154]. A periodic server is released with a fixed period and executes any ready tasks until its capacity is exhausted. To fully consume the capacity of the servers in the application's set, we consider that each application contains a set of idle tasks that continuously carry out some work.

Recall from Chapter 6. that the worst-case response time RT_i of task τ_i is defined as the longest time from the arrival of any job τ_i^j of τ_i until the job completes executing. Similarly, we define here the worst-case response time RT_s of a server σ_s.

Definition 13. *The worst-case response time RT_s of a server σ_s is the longest time elapsed from the moment when the server capacity is replenished until its capacity is exhausted. In terms of worst-case response time, a server σ_s is schedulable if $RT_s \leq T_s$.*

Shared resources. The jobs of any task can issue requests for exclusive access to shared resources $R_1, R_2, ..., R_r$. In order to avoid deadlock, we adopt a strategy similar to the one in [50] and use resource groups to support nestable resource accesses. Two resource R_k and R_l are in the same group if and only if there exists at least one job which issues a request for R_l that is nested within a request for R_k. Resources that are non-nestable (i.e. there are no requests for them nested within requests for other resources) will form groups by themselves. Access to a resource group is restricted by a group lock which is a binary semaphore or a mutex. Consequently, a job may block only once when requesting the first resource in the group as nested requests will be granted immediately since the job will already hold the group lock. Furthermore, we assume that a task can be preempted while it holds the lock of a local resource group and that the task will hold the lock until it explicitly releases it.

Depending on whether a resource is shared only by tasks in a single application or by tasks in different applications, we identify two kinds of resources: *local* and *global*. Moreover, we make distinction between *local* and *global resource groups*. Local resource groups are comprised only of local resources while global resource groups can contain either only global resources or both global and local resources.

We denote by $\mathcal{G} = \mathcal{G}_l \cup \mathcal{G}_g$ the set of all resource groups, where $\mathcal{G}_l$ is the set of all local resource groups and $\mathcal{G}_g$ is the set of all global resource groups. We assume that for each task τ_i the worst case resource group usage time among all requests for a resource group $g \in \mathcal{G}$ by job of τ_i is $C_{i,g}$. Also, we consider that $C_{i,g} < C_s \ \forall i, g \ and \ s$. Further, we use $CT_{i,g}$ to denote the worst-case total resource group usage time for g by any single job of τ_i.

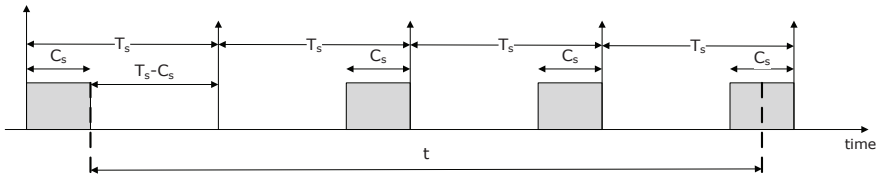


Figure 7.3: The supply bound function of a periodic execution server

7.3. Response Time Analysis in Hierarchical Scheduling

In this section we introduce the schedulability analysis for independent hierarchical real-time systems scheduled on a multiprocessor platform. The schedulability test introduced here will be extended in the following sections to handle also resource sharing and task synchronization. In order to guarantee the schedulability of an application, it is necessary to first make sure that all its servers are schedulable. For this purpose we can apply the RTA-based analysis developed by Guan et al. [94] and presented in the Section 6.3. After checking the schedulability of the servers, the next step is to check the schedulability of the application tasks executing within those servers. We extend the schedulability analysis in [94] for the case when the execution of the tasks is constrained by the application servers.

To be able to analyze the schedulability of an application, it is necessary to calculate the minimum processor supply provided by its assigned set of servers. For a periodic server σ_s its supply bound function $sbf_s(t)$ gives the minimum processor supply for any time interval of length t. In order to compute $sbf_s(t)$ we assume the worst case, when the time interval starts right after the capacity of the server has exhausted and, in all periods that follow, the capacity is replenished as late as possible (see Figure 7.3):

$$sbf_s(t) = \left\| \left\lfloor \frac{t-(T_s-C_s)}{T_s} \right\rfloor C_s \right\|_0 + \left\| \left(t-(T_s-C_s)\right) \mod T_s - (T_s-C_s) \right\|_0^{C_s} \tag{7.1}$$

The supply bound function of a server is a non-decreasing step function and, in order to decrease the time complexity of the computations, a linear approximation of it is often used. Such a linear function is defined in [153]:

$$lsbf_s(t) = \frac{C_s}{T_s}(t - 2(T_s - C_s)) \tag{7.2}$$

Using Equation (7.1), the supply bound function of a set of servers can be computed as the sum of all its server's supply functions: $sbf(t) = \sum_{\forall \sigma_s} sbf_s(t)$. Although schedulability conditions can be derived using $sbf(t)$, in order to reduce the time-complexity of the analysis, the following linear bound of $sbf(t)$ (obtained by summing up the servers' linear bounds) will be used instead :

$$lsbf(t) = U_\sigma(t - \delta_\sigma) \tag{7.3}$$

where $U_\sigma = \sum_{\forall \sigma_s} \frac{C_s}{T_s}$ and

$$\delta_\sigma = \frac{2}{U_\sigma} \sum_{\forall \sigma_s} \frac{C_s}{T_s}(T_s - C_s). \tag{7.4}$$

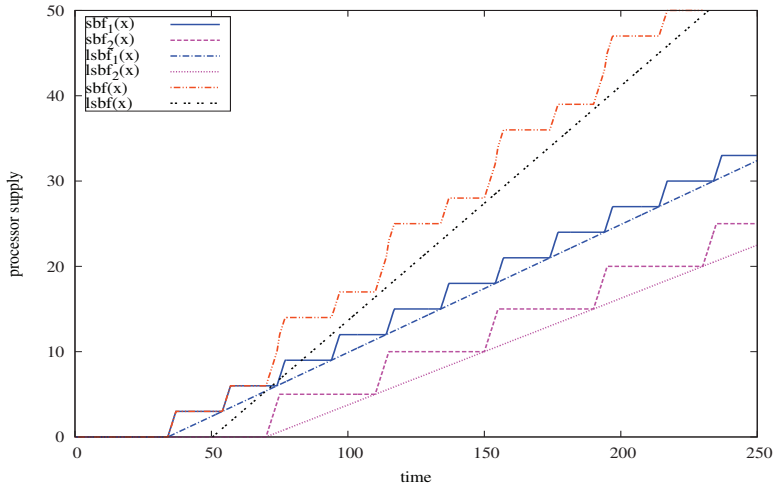


Figure 7.4: The supply bound of a server set and its linear bound

Example 9. *Figure 7.4 depicts the supply bound function and its linear bound for a set of two servers* $\sigma = \{\sigma_1, \sigma_2\}$ *with* $\sigma_1 = (3, 20)$ *and* $\sigma_2 = (5, 40)$*. The figure also shows the supply bound functions of the two servers and the corresponding linear bounds.*

To analyze the schedulability of a task τ_i, we use the same technique as [26, 94] and consider that a release of this task is the first job to miss its deadline. We then compute what is the total interference from higher priority tasks during a problem window of length x, as presented in Section 6.3. The job of τ_i that misses its deadline is called *problem job*. Since the deadline of the job is missed, it means that the total interference $\Omega_i(x)$ together with the computation time requested by τ_i is higher than the processor supply of the set of servers, or formally $\Omega_i(x) > sbf(x) - C_i$. However, as pointed out in [60], because the problem job cannot execute in parallel, it is possible to lower this bound on the total interference. If $mp(C_i)$ represents the maximum computation time provided by the set of servers during any C_i out of D_i than it is sufficient for $\Omega_i(x)$ to be higher than $sbf(x) - mp(C_i)$ in order for τ_i to miss a deadline [60]. An algorithm for computing $mp(C_i)$ of a set of servers is given in [60] (see Algorithm 3). Furthermore, in order to reduce complexity we use $lsbf(x)$ instead of $sbf(x)$ and get the following schedulability condition for τ_i:

$$\Omega_i(x) \leq lsbf(x) - mp(C_i) \tag{7.5}$$

After defining an upper bound on the total interference on the problem job of τ_i during the problem window, in what follows we use this result to find the worst case response time of the task by extending the analysis in [94] to hierarchical systems respecting the model introduced in Section 7.2. Note that, for beginning, we assume that tasks do not share any resource other than the processors.

In the following lemmas and theorems, f_i denotes the finish time of job τ_i^j and t_0 denotes the latest time instant before the release time r_i of job τ_i^j at which at least one processor is idle (see Figure 7.5).

Lemma 1. *For all jobs* τ_i^j *and all* $x < f_i - t_0$*, the following holds:*

$$\Omega_i(x) > lsbf(x) - mp(C_i). \tag{7.6}$$

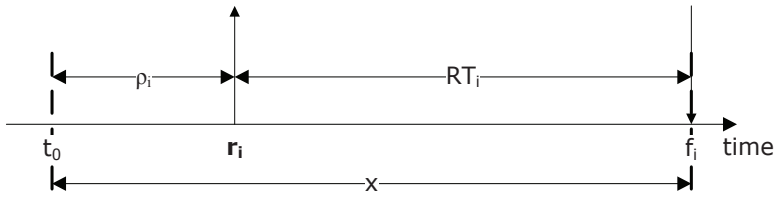


Figure 7.5: The problem window for task τ_i

Proof. Since we assume $x < f_i - t_0$ it means that job τ_i^j has executed for less than C_i time units during interval $[t_0, t_0 + x)$. This means that the higher priority tasks kept busy the available servers for at least $lsbf(x) - mp(C_i) + 1$ time units. Therefore we have $\Omega_i(x) \geq lsbf(x) - mp(C_i) + 1$, which is equivalent to the lemma. □

For each task τ_i the problem window starts at moment t_0, which is ρ_i time units before the release time r_i of τ_i. Similar to [94] we consider $t_0 = r_i - \rho_i$ and express the response time analysis based on the following lemma:

Lemma 2. *Given a $\rho_i \geq 0$, let $\mathcal{X}$ be the minimal solution of the recursive equation*

$$x = \frac{\Omega_i(x) + mp(C_i)}{U_\sigma} + \delta_\sigma. \tag{7.7}$$

Then $\mathcal{X} - \rho_i$ is an upper bound of τ_i's response time for the particular $t_0 = r_i - \rho_i$.

Proof. Similar to [94], lets assume that R and not $\mathcal{X}$ is the worst case response time of τ_i for $t_0 = r_i - \rho_i$ and that $\mathcal{X} - \rho_i < R$. If R is the worst case response time it means that $R = f_i - r_i$. We then have that $\mathcal{X} - \rho_i < f_i - r_i$. It follows that $\mathcal{X} < f_i - r_i + \rho_i = f_i - t_0$.

From Lemma 1 we then have that $\Omega_i(\mathcal{X}) > lsbf(\mathcal{X}) - mp(C_i)$ which, along with Equations (7.3) and (7.4), contradicts the hypothesis that $\mathcal{X}$ is a solution of Equation (7.7). □

It is important to note that the total interference $\Omega_i(x)$ depends only on the length x of the problem window and is independent of ρ_i. Furthermore, from the lemma above we know that $\mathcal{X} - \rho_i$ is the upper bound on the response time of task τ_i for $t_0 = r_i - \rho_i$ which implies that doing RTA according to the lemma for $\rho_i = 0$ ($t_0 = r_i$) will give us an upper bound on the worst-case response time of τ_i.

Theorem 7. *An upper bound on the worst-case response time of task τ_i is given by the minimal solution $\mathcal{X}$ of Equation (7.8) determined by doing fixed-point search starting with $x = C_i$:*

$$x = \frac{\Omega_i(x) + mp(C_i)}{U_\sigma} + \delta_\sigma. \tag{7.8}$$

Proof. From Lemma (2) by considering $\rho_i = 0$. □

The recurrence relation given by Equation (7.8) ends when x converges or when $x > D_i$ in which case the task is unschedulable.

7.4. The Parallel Hierarchical Resource Policy

In this section we present a new synchronization protocol for multi-core/multiprocessor HSFs, called Parallel Hierarchical Resource Policy (P-HRP). P-HRP is the first such protocol for multiprocessor HSFs which enables task parallelism and is designed for global fixed-priority preemptive scheduling. Under P-HRP access to shared resources is governed by a two-level set of rules. At the first level, we have a set of rules controlling access to globally shared resources from tasks assigned to different servers, belonging to different applications. A second level of rules ensures that resources shared only locally by tasks in the same application are accessed in mutually exclusive manner.

Global Resource Groups

If a task holding the lock for a local resource group is suspended, only the suspended task and the tasks in the same application are affected. However, if a task with a global resource group locked is suspended, then tasks in other applications waiting to access the same group will be delayed.

In hierarchically scheduled real-time systems such task suspensions can be caused by depletion of server capacity while the task accesses a global resource. To prevent this problem two mechanisms have been employed previously: *server overrun (with or without payback)* [68] and a *skipping mechanism* [39]. Under the overrun mechanism, upon depletion of the server budget during global resource access, the budget is temporally increased with a statically determined amount for the duration of that access. If the supplementary budget received in one server period is deducted from the next period, we have *server overrun with payback*, otherwise it is *server overrun without payback*. Under the skipping mechanism, the server's remaining budget is checked before entering a critical section and if it is smaller than the length of the critical section, the job requesting the resource is suspended until replenishment of the budget. As proved by Behnam et al., the superiority of one mechanism over the other is highly dependent on application task parameters [37].

In what follows we consider that the server overrun with payback mechanism is employed for solving the capacity depletion problem. This overrun is limited to the maximum resource access time. To ensure that this access time is as short as possible, preemption of the task holding the resource group lock, by other tasks in the same application, must be avoided. Consequently, in the proposed protocol, a task that has locked a global resource group is non-preemptable until it releases the resource group. The following rules govern global resource accesses:

(1) For each application we use MP_a to denote the priority of the highest priority server in its server set. Associated with each global resource group $g \in \mathcal{G}_g$ there is a *global priority ceiling* $GPC(g)$ equal to the maximum MP_a of all applications with tasks that access resources in g.

(2) When a job τ_i^j issues a request for a global resource group g its server must acquire the group lock. If at the time of the request the lock is held by another server, the server and job τ_i^j will be blocked. Blocked servers are added to a prioritized queue and suspended and the blocked job will remain associated to the blocked server.

(3) Whilst a job accesses a resource group the priority of its server is raised to the highest priority of any of the servers blocked on the same group and waiting its release.

(4) Whilst a job access a global resource group it becomes non-preemptable.

(5) If the server capacity is exhausted whilst a job holds the lock of a global resource group, the server continues to execute until the job inside releases the resource group.

(6) If a server overruns its capacity then its capacity in the next period will be reduced by the amount of the overrun.

In a Hierarchical Scheduling Framework, the Limited Blocking Priority Ceiling Protocol can be applied as it is only for local resources. The reason for this is that in a HSF, we must control the number of indirect blockages at application level but also at system level. At application level (second level in the hierarchy depicted in Figure 7.2) the number of indirect blockages due to local resource access can be directly managed using the LB-PCP protocol as presented in Section 6.4. Indirect blockages due to global resource accesses from tasks in the same application can be simply added to those due to local resource accesses. However, when a task belonging to one application uses a global resource group, it can indirectly block a task in another application. Specifically, this happens when the server used by the task holding a global resource group lock runs at increased effective priority and indirectly blocks a server belonging to another application with lower effective priority, but higher base priority. Simply applying LB-PCP at system level (first level in the hierarchy depicted in Figure 7.2) and adding a blocking counter for servers is not useful since a server is just a virtual processor and the tasks assigned to them can change while the server is blocked. A possible approach for applying LB-PCP at system level is proposed in [128].

Local Resource Groups

We assume that access to local resource groups is according to the set of rules described below:

(1) Each local resource group $lg \in \mathcal{G}_l$ has a *local priority ceiling* $LPC(lg)$ associated with it, where $LPC(lg)$ is equal to the highest priority of any task that may access any of the resources in the group.

(2) When a job τ_i^j issues a request for a local resource group lg it must acquire the group lock. If at the time of the request the lock is held by another job, τ_i^j will be blocked. Blocked jobs are added to a prioritized queue and suspended.

(3) While a job holds the group lock its priority will be raised to the highest priority of any task currently blocked waiting for the lock to be released. This is in accordance to PIP, P-PCP or LB-PCP. Another possibility would be to always execute the critical section with an effective priority equal to the priority ceiling of the resource group. This would avoid penalties due to dynamically changing priorities and would reduce also the implementation costs but may increase delays of higher priority tasks that are not blocked on the same resource group.

(4) If the capacity of the server executing the job is exhausted whilst the job is holding a group lock the server is suspended. If another server in the application's server set is still available (i.e. its capacity is not finished) then it is possible for the job to continue its execution in this server.

The set of rules above can be combined with either of the protocols PIP, P-PCP or LB-PCP.

In a HSF using P-HRP, for each task τ_i in an application there are two sources of indirect blocking. First of all, a lower priority task using a local resource group may run at increased effective priority, preventing τ_i from running if its priority is lower than the priority ceiling of the resource group. Second, when a low priority task uses a global resource group it becomes non-preemptable and can block any higher priority task from running, whatever its priority. For these reasons, we need to slightly adapt LB-PCP. These adaptations refer mostly to the way the counter LPR_i for the number of jobs with base priority lower than i but potential effective priority higher than i is updated, but also to the update rules for the IBC_i counter which holds the current number of times that jobs with priority higher than i can be indirectly blocked. Moreover, since in a hierarchical system, we cannot know a-priori the number of minimum running servers at all time instants, we have to remove the assumption that $IBT_1 = ... = IBT_m = 0$. In this case, we only know that $IBT_1 = 0$ and $IBT_1 \leq IBT_2 \leq ... \leq IBT_n$ and we leave the system designer choose the values for all IBT_is, $i \geq 2$.

In what follows, Algorithms 3, 4, 5 and 6 give the LB-PCP resource sharing protocol as it should be applied by an application level scheduler in a Hierarchical Scheduling Framework.

The major difference from Algorithm 1 lies in the way global resources are handled. When a job τ_i^j uses a global resource it runs non-preemptively and as a result can block any higher priority job, not just the ones with priority lower than the priority ceiling of the used group. This is why, in Algorithm 6 we check the values of the blocking counters of all higher priority tasks and, in Lines 4–6 of Algorithm 5, we update the LPR_i counter for all higher base priority tasks.

Depending on the characteristics of each application (i.e. number of servers in the associated set, taskset utilization, taskset cardinality), one can choose the protocol that will give the best performance. Furthermore, the choice of one of these protocols in one application is independent of the protocols used in the other applications in the system. Based on the performance evaluation in Section 6.6. we make the following recommendations:

- If the application consists of a large number of tasks, each using several local or global resources then LB-PCP should be used no matter the number of servers in the application's set.
- If the number of servers in the application's set and the number of available processors is large but the applications consists of just a few tasks, the application level scheduler should use P-PCP for handling resource sharing.
- If the number of servers in the application's set and the number of available processors is large but the applications consists of many tasks, the application level scheduler should use LB-PCP for handling resource sharing.

7.5. Response Time Analysis under the Parallel Hierarchical Resource Policy

In this section we calculate an upper bound on the response time of servers and tasks in a two-level HSF. The servers and tasks are scheduled on a m processor platform, using a global preemptive fixed-priority algorithm and resource sharing is according to the rules of P-HRP. We will first determine these upper bounds assuming

Algorithm 3 Resource sharing under P-HRP with LB-PCP

for each job τ_i^j in decreasing effective priority order **do**
  if there are unassigned running servers **then**
    if τ_i^j is not requesting a resource **then**
      assign server to τ_i^j
    else
      Let g be the resource group owning the resource requested by τ_i^j
      if g is global **then**
        Run Algorithm 5
      else if g is locked **then**
        Let τ_l be the task currently holding resource group g
        if $l > i$ **then**
          $EP_l = i$
        end if
      else
        if g can be granted to τ_i^j according to rules in Algorithm 4 **then**
          assign server to τ_i^j
          for $ind = LPC(g) + 1$ **to** $i - 1$ **do**
            $LPR_{ind} = LPR_{ind} + 1$
          end for
        else
          suspend τ_i^j
        end if
      end if
    end if
  else if there is a lower base priority job $\tau_l^{j'}$ running with priority $EP_l < i$ or non-preemptively **then**
    for $ind = i + 1$ **to** n **do**
      $IBC_{ind} = IBC_{ind} - 1$
    end for
    Set LBT_i to the base priority of the job blocking τ_i^j
  end if
end for

Algorithm 4 Access rules to local resource group g by job τ_i^j

for $ind = LPC(g) + 1$ **to** $i - 1$ **do**
  if $(IBC_{ind+1} - IBC_{ind}) - LPR_{ind} \leq 0$ **then**
    return false
  end if
end for
return true

Algorithm 5 Global resource sharing with LB-PCP for group g and job τ_i^j

```
1: if g is unlocked then
2:   if g can be granted to τ_i^j according to rules in Algorithm 6 then
3:     assign server to τ_i^j
4:     for ind = 1 to i − 1 do
5:       LPR_ind = LPR_ind + 1
6:     end for
7:   else
8:     suspend τ_i^j
9:   end if
10: end if
```

Algorithm 6 Access rules to global resource group g by job τ_i^j

```
1: for ind = 1 to i − 1 do
2:   if (IBC_{ind+1} − IBC_ind) − LPR_ind ≤ 0 then
3:     return false
4:   end if
5: end for
6: return true
```

that the PIP protocol is used at both levels of the hierarchy and then, we will extend the results for the case when LB-PCP is used at application level.

7.5.1. Notations

We define $B(\sigma_s, g)$ as the longest time for which any task in a server σ_s may access a resource in a global resource group $g \in \mathcal{G}_g$.

$$B(\sigma_s, g) = \max\{CT_{i,g} : \tau_i \in \sigma_s \wedge \tau_i \in \mathcal{A}(g)\}, \tag{7.9}$$

where by $\mathcal{A}(g)$ we denote the set of tasks that may require access to a resource in group g. The maximum overrun time of server σ_s can then be computed as:

$$B(\sigma_s) = \max\{B(\sigma_s, g) : g \in \mathcal{G}(\sigma_s)\}, \tag{7.10}$$

where $\mathcal{G}(\sigma_s)$ is the set of all global resource groups accessed by a task that may execute in σ_s. Note that for all servers of the same application the maximum overrun time will be the same.

Also, we define $B_{\overline{x}}(\sigma_s)$ as the maximum time for which a task in σ_s accesses a resource group that is not also accessed by any task in server σ_x:

$$B_{\overline{x}}(\sigma_s) = \max\{B(\sigma_s, g) : g \in \mathcal{G}(\sigma_s) \wedge g \notin \mathcal{G}(\sigma_x)\}, \tag{7.11}$$

We also define $B(\tau_i, g) = \max\{CT_{i,g} : g \in \mathcal{G}(\tau_i)\}$ as the longest time for which a job of task τ_i may access a resource in resource group g, local or global:

$$B(\tau_i, g) = \max\{CT_{i,g} : g \in \mathcal{G}_g(\tau_i) \cup \mathcal{G}_l(\tau_i)\}, \tag{7.12}$$

where by $\mathcal{G}_g(\tau_i)$ and $\mathcal{G}_l(\tau_i)$ we denote the sets of global and local resource groups accessed by τ_i with $\mathcal{G}(\tau_i) = \mathcal{G}_g(\tau_i) \cup \mathcal{G}_l(\tau_i)$.

7.5.2. Parallel Hierarchical Resource Policy with PIP

7.5.3. Server Response Time

In this section we calculate the worst-case response time RT_s of a server σ_s scheduled using a global fixed-priority preemptive scheduler with resource sharing under P-HRP. There are four parameters that influence the value of RT_s: (1) the server's capacity C_s, (2) the amount of time server σ_s is blocked waiting for the resources it requests to be granted, (3) the amount of execution that lower priority servers execute at a priority higher than that of σ_s, and (4) the amount of execution that servers with priority higher than σ_s can perform.

A server may overrun if the task within holds the lock of a global resource group. This has a negative impact on schedulability of server σ_s with worst case effects when: (1) for all servers that block σ_s directly, the task that executes within accesses a group for the longest time amongst all tasks of the application, (2) all lower priority servers, that execute at higher priority, access a group for the maximum possible amount, and (3) all releases of higher priority servers running between the release and finish of σ_s overrun by their maximum amount.

Under P-HRP, a server σ_s can suffer blocking delays when the task that executes within σ_s issues a request for a global resource group g. There are two possible sources of blocking for σ_s.

First, *direct blocking* occurs when σ_s is one of the m highest priority servers but the lock for group g is locked by some other server. The worst case scenario occurs when σ_s requests group g immediately after it has been granted to a lower priority server and all higher priority servers also request access to the resource. Therefore, the direct blocking time during an interval of length x can be expressed as follows:

$$DB_{\sigma_s}(x) = \sum_{g \in \mathcal{G}(\sigma_s)} \left(N_{s,g} \cdot \left(\max_{\sigma_l \in lp(s)} \{C_{l,g}\} - 1 \right) + \sum_{\sigma_h \in hp_g(\sigma_s)} I_s^{CI}(h, B(\sigma_h, g), x) \right) \tag{7.13}$$

where $N_{s,g}$ denotes the maximum number of times an instance of σ_s requires access to group g and $I_s^{CI}(h, B(\sigma_h, g), x)$ is the interference due to higher priority servers requesting the same resource group g and is computed using Equation (6.6) but considering only the set of higher priority servers that use g and assuming that the interference produced by a server σ_h is due to $B(\sigma_h, g)$ units of work:

$$I_s^{CI}(h, B(\sigma_h, g), x) = \|W_h^{CI}(B(\sigma_h, g), x)\|^{x - C_s + 1} \tag{7.14}$$

In the equations above we used $lp(s)$ to denote the set of servers with priorities lower than σ_s, $hp(s)$ to denote the set of higher priority servers and $hp_g(s)$ for the set of all higher priority servers that use resource group g.

The second source of blocking is *indirect* and occurs when σ_s cannot execute because a lower priority server σ_l executes with a priority higher than σ_s while it accesses a resource group. Under PIP, the indirect blocking time for a sever IB_{σ_s} states that a lower priority server σ_l may interfere with σ_s multiple times and may delay the execution of σ_s just as much as any higher priority server. The indirect blocking time of σ_s can be upper bounded as follows:

$$IB_{\sigma_s}(x) = \frac{1}{m} \Omega_s^{lp}(x) \tag{7.15}$$

where $\Omega_s^{lp}(x)$ is the total interference from lower priority servers while accessing global

resources with priority ceiling greater than s. In the computation of $\Omega_s^{lp}(x)$ we consider that only the $\theta(l,s) = \sum_{g \in \mathcal{G}(\sigma_l) \wedge GPC(g) < s} B(\sigma_l, g)$ execution units are relevant:

$$\Omega_s^{lp}(x) = \max_{(\sigma^{NC}, \sigma^{CI}) \in \mathcal{Z}_{lp(s)}} \left(\sum_{\sigma_l \in \sigma^{NC}} I_s^{NC}(l, \theta(l,s), x) + \sum_{\sigma_l \in \sigma^{CI}} I_s^{CI}(l, \theta(l,s), x) \right) \tag{7.16}$$

where $\mathcal{Z}_{lp(s)}$ is the set of all partitions of the set of servers with priority lower than σ_s.

The worst-case effects on the response time of a server σ_s due to interference on its execution from higher priority servers that execute within critical sections of resource groups others than the ones shared with σ_s, occur when all releases of such servers succeeding the release of σ_s overrun by their maximum amount due to accesses to other global resources. When the *overrun and payback* mechanism is enabled, the first invocation of such higher priority servers σ_h will execute for $C_h + B_{\overline{s}}(\sigma_h)$ time units while next invocations will have an execution time of only C_h even though they may also overrun. The interference of these servers can be expressed as:

$$I_{s,osh}^X(h, \beta(h,s), x) = \left\| W_h^X(\beta(h,s), x) + B_{\overline{s}}(\sigma_h) \right\|^{x - C_s + 1} \tag{7.17}$$

where X is a placeholder for NC and CI, $\beta(h,s) = \sum_{g \in \mathcal{G}(\sigma_h) \wedge g \notin \mathcal{G}(\sigma_s)} B(\sigma_h, g)$ and W_h^X can be computed as in Equations (6.2) and (6.3).

Furthermore, a job of σ_s suffers interference from higher priority servers when these execute outside any critical section. The interference caused by such situations can be computed as follows:

$$I_{s,nsh}^X(h, \gamma(h), x) = \left\| W_h^X(\gamma(h), x) \right\|^{x - C_s + 1} \tag{7.18}$$

where $\gamma(h) = C_h - \sum_{g \in \mathcal{G}(\sigma_h)} B(\sigma_h, g)$.

These new definitions of $I_{s,osh}^X(h,c,x)$ and $I_{s,nsh}^X(h,c,x)$ will be used for computing the total interference in the relation below (due to space considerations, we dropped the middle parameter):

$$\begin{aligned} \Omega_s(x) = \max_{(\sigma^{NC}, \sigma^{CI}) \in \mathcal{Z}} \Bigg(& \sum_{\sigma_h \in \sigma^{NC}} \left(I_{s,osh}^{NC}(h,x) + I_{s,nsh}^{NC}(h,x) \right) + \\ & \sum_{\sigma_h \in \sigma^{CI}} \max\left(I_{s,osh}^{NC}(h,x) + I_{s,nsh}^{CI}(h,x), I_{s,osh}^{CI}(h,x) + I_{s,nsh}^{NC}(h,x) \right) \Bigg) \end{aligned} \tag{7.19}$$

Server worst-case response time can be determined by incorporating the blocking and total interference factors into the recurrence relation defined by Equation (6.8):

$$RT_s = C_s + DB_{\sigma_s}(RT_s) + IB_{\sigma_s}(RT_s) + \left\lfloor \frac{\Omega_s(RT_s)}{m} \right\rfloor \tag{7.20}$$

The recurrence starts with $RT_s = C_s$ and ends when RT_s converges, in which case RT_s gives the worst-case response time of server σ_s, or when $RT_s > T_s$ which means that the server is unschedulable.

7.5.4. Task Response Time

In HSFs, the factors that influence the worst-case response time of a task scheduled according to a global fixed priority preemptive scheduler can be grouped based on the level where they manifest themselves. At the first level, we identify factors such as the availability and the response time of the servers that may execute the task, while at application level, the worst-case load and the resource requests handled between the release and the deadline of the task under analysis, are key factors.

The workload that must be executed before the completion of task τ_i is affected by task accesses to local and global resource groups. When a task holds the lock of a local resource group lg its priority may be raised to the priority ceiling $LPC(lg)$ which may be higher than the priority of τ_i. Furthermore, whenever a task requests access to a global resource group g it becomes non-preemptable which may also prevent higher priority tasks from executing. We call such situations *indirect task blocking*. Moreover, just like in the case of servers, a task τ_i may also be *directly blocked* by another task that requests the same local resource group lg as τ_i, or, in case τ_i issues a request for a global resource group g, it is possible for τ_i to be blocked by some task in another application using group g.

The worst case delay caused by indirect blocking to a task τ_i is:

$$IB_{\tau_i}(x) = \Omega_i^{lp}(x) \tag{7.21}$$

Above, we assume that the lower priority tasks execute in the lowest priority server σ_{l_0} and express the blocking as the total interference from the lower priority tasks with increased priorities while executing local and global critical sections as follows:

$$\Omega_i^{lp}(x) = \max_{(\tau^{NC},\tau^{CI})\in\mathcal{Z}_{lp(i)}} \left(\sum_{\tau_l\in\tau^{NC}} I_i^{NC}(l,\theta_\tau(l,i),x) + \sum_{\tau_l\in\tau^{CI}} I_i^{CI}(l,\theta_\tau(l,i),x) \right) \tag{7.22}$$

where $\theta_\tau(l,i) = \sum_{lg\in\mathcal{G}_l(\tau_l)\wedge LPC(lg)<i} B(\tau_l,lg) + \sum_{g\in\mathcal{G}_g(\tau_l)\wedge g\notin\mathcal{G}_g(\tau_i)} B(\tau_l,g)$, $lp(i)$ is the set of tasks with base priority lower than τ_i and $\mathcal{Z}_{lp(i)}$ is the set of all partitions of tasks with priority lower than τ_i which use local resources with priority ceiling greater than i or use global resources. Note that $|\tau^{CI}| \leq \min(|\sigma^a|, m) - 1$ where $|\sigma^a|$ is the number of servers in application's a set.

Like in the case of servers, every request for a local group $lg \in \mathcal{G}_l(\tau_i)$ of job τ_i^j may be directly blocked by at most one lower priority job. Moreover, in the worst case scenario, it is possible that jobs of all higher priority tasks also request lg at the same time with τ_i^j. Therefore task direct blocking time due to access to local resources can be defined as follows:

$$DB_L^{\tau_i}(x) = \sum_{lg\in\mathcal{G}_l(\tau_i)} \left(N_{i,lg} \cdot (\max_{\tau_l\in lp(i)} \{C_{l,lg}\} - 1) + \sum_{\tau_h\in hp_{lg}(i)} I_i^{CI}(h, B(\tau_h,lg), x) \right) \tag{7.23}$$

where $I_i^{CI}(h, B(\tau_h,lg), x)$ is computed as follows:

$$I_i^{CI}(h, B(\tau_h,lg), x) = \|W_h^{CI}(B(\sigma_h,lg), x)\|^{x-C_i+1} \tag{7.24}$$

In the equation above we consider the set $hp_{lg}(i)$ of all tasks with higher priority than τ_i which use local resource group lg.

Whenever a task uses a global resource it remains assigned to the server

where it is currently running and consequently, while the server is directly blocked by other servers, the task executing inside will also be blocked. This blocking time represents the task direct blocking time for global resource requests and is maximum when the task is assigned to the lowest priority server in the application's set. The *task direct blocking time* due to requests for global resource groups is equal to $DB_G^{\sigma_{l_0}}$:

$$DB_G^{\tau_i}(x) = \sum_{g \in \mathcal{G}_g(\tau_i)} \left(\left(\max_{\sigma_l \in lp(l_0)} \{C_{l,g}\} - 1 \right) + \sum_{\sigma_h \in hp_g(l_0)} I_{l_0}^{CI}\left(h, B(\sigma_h, g), \min\{x, RT_{l_0}\}\right) \right) \quad (7.25)$$

where $I_{l_0}^{CI}(h, B(\sigma_h, g), \min\{x, RT_{l_0}\})$ is computed according to Equation (7.14).

Furthermore, the worst-case response time of task τ_i is also influenced by interference from higher priority tasks in the same application that either execute outside critical sections or inside critical sections of resource groups not accessed by τ_i. We compute this interference for the case when these tasks have carry-in (CI) jobs relative to the problem window of τ_i and for the case when there is no carry-in (NC). Next, we express the interference due to executions with resource groups not shared with τ_i ($I_{i,osh}^X$) and executions with no shared resources ($I_{i,nsh}^X$) (again X is just a placeholder for NC and CI):

$$I_{i,osh}^X(h, \beta_\tau(h,i), x) = \left\| W_h^X(x, \beta_\tau(h,i)) \right\|^{x - C_i + 1} \quad (7.26)$$

$$I_{i,nsh}^X(h, \gamma_\tau(h), x) = \left\| W_h^X(x, \gamma_\tau(h)) \right\|^{x - C_i + 1} \quad (7.27)$$

where $\beta_\tau(h,i) = \sum_{g \in \mathcal{G}(\tau_h) \wedge g \notin \mathcal{G}(\tau_i)} B(\tau_h, g)$ and $\gamma_\tau(h) = C_h - \sum_{g \in \mathcal{G}(\tau_h)} B(\tau_h, g)$. We can compute the total interference as (due to space considerations we have omitted the middle parameter for I^X):

$$\begin{aligned} \Omega_i(x) = \max_{(\tau^{CI}, \tau^{NC}) \in \mathcal{Z}} \Bigg(& \sum_{\tau_h \in \tau^{NC}} \left(I_{i,nsh}^{NC}(h,x) + I_{i,osh}^{NC}(h,x) \right) + \\ & \sum_{\tau_h \in \tau^{CI}} \max\left(I_{i,nsh}^{NC}(h,x) + I_{i,osh}^{CI}(h,x), I_{i,nsh}^{CI}(h,x) + I_{i,osh}^{NC}(h,x) \right) \Bigg) \end{aligned} \quad (7.28)$$

where $|\tau^{CI}| \leq \min\left(|\sigma^a|, m\right) - 1$ (i.e. $|\sigma^a|$ is the number of servers in application's a set).

From the discussions above, we can determine the worst case response time of a task scheduled under P-HRP with PIP by incorporating the blocking factors into the recurrence relation defined by Equation (7.8):

$$RT_i = \left\lfloor \frac{\Omega_i(RT_i) + mp(C_i) + IB_{\tau_i}(RT_i)}{U_{\sigma^a}} \right\rfloor + \delta_\sigma + DB_L^{\tau_i}(RT_i) + DB_G^{\tau_i}(RT_i) + IB_{\sigma_{l_0}}(RT_i) \quad (7.29)$$

The recurrence relation in Equation (7.29) starts with $RT_i = C_i$ and ends when the value of RT_i converges or $RT_i > D_i$ in which case the task is unschedulable.

7.5.5. Parallel Hierarchical Resource Policy with LB-PCP

We derive in this section an upper bound on the response time of a task τ_i in an application of the HSF, scheduled with a global multiprocessor preemptive fixed-priority scheduler with resources shared under the rules of P-HRP combined with LB-PCP.

Note that, since we can only apply LB-PCP at application level, the RTA for servers requires no modifications and is similar to the one in Section 7.5.3. Therefore, in this section, only the RTA for application tasks is modified.

The local and global direct blocking time ($DB_L^{\tau_i}$ and $DB_G^{\tau_i}$) is identical to that under PIP described by equations (7.23) and (7.25), respectively. This follows from the fact that, under LB-PCP, as well as under PIP, when a job τ_i^j requests a local resource group it has to wait for at most one lower priority job and for all higher priority jobs to release the resource group. Similarly, when it requests a global resource it has to wait the release of the group lock by at most one job, of the same or of another application, running in a lower priority server, and by all jobs running in higher priority servers.

Higher priority interference from no shared resource execution ($I_{i,nsh}^X$) is also identical to that under PIP because LB-PCP behaves the same way as PIP when handling executions without shared resources.

Under LB-PCP a job τ_i^j can be indirectly blocked at most $IBT_{i+1} - IBT_i$ times by any lower priority task in the same application executing with higher effective priority, while accessing a local resource group, or non-preemptively, while holding the lock of a global resource group. Therefore, the indirect blocking time under LB-PCP is lower than under PIP and, in the worst case, it is upper bounded by:

$$IB_{\tau_i}(x) = \left\lfloor \frac{(IBT_{i+1} - IBT_i) \max_{\tau_l \in lp(i) \wedge ((g \in \mathcal{G}_l(\tau_l) \wedge LPC(g) < i) \vee (g \in \mathcal{G}_g(\tau_l) \wedge g \notin \mathcal{G}_g(\tau_i)))} (C_{l,g})}{\min(U_{\sigma_a}, IBT_{i+1} - IBT_i)} \right\rfloor \tag{7.30}$$

For each request to a resource group g by a job τ_i^j, the job may be suspended when either of the Algorithms 6 or 4 deems that letting the job enter a critical section may cause disallowed indirect blockages to one or several higher priority jobs.

For each request of job τ_i^j to a local resource group lg, in the worst case scenario, the job will be suspended during all shared resource executions $lg' \in \mathcal{G}_l(\tau_h)$ of every job with priority h, $LPC(lg) + 1 < h < i$, with $lg' \neq lg$ and $LPC(lg') < h$. Therefore, the suspension time determined by local resource requests from jobs of task τ_i depends on the interference from local resource accesses of tasks with priority h, $\min_{lg \in \mathcal{G}_l(\tau_i)}\big(LPC(lg)\big) + 1 < h < i$, and can be bounded by:

$$\Omega_{i,hp}^{lsus}(x) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}^{LPC(lg)}} \left(\sum_{\tau_h \in \tau^{NC}} I_i^{NC}(h, \beta_h^l, x) + \sum_{\tau_h \in \tau^{CI}} I_i^{CI}(h, \beta_h^l, x) \right) \tag{7.31}$$

where $\mathcal{Z}^{LPC(lg)}$ is the set of all task partitions with higher priority h, $\min_{lg \in \mathcal{G}_l(\tau_i)}\big(LPC(lg)\big) + 1 < h < i$, and

$$\beta_h^l = \sum_{lg \in \mathcal{G}_l(\tau_h) \wedge lg \notin \mathcal{G}_l(\tau_i) \wedge LPC(lg) < h} B(\tau_h, lg). \tag{7.32}$$

Additionally, in the worst case scenario, τ_i^j will also be suspended during all global resource accesses of tasks with higher base priority. Hence, suspension time due to interference from global resource accesses can be bounded by:

$$\Omega_{i,hp}^{gsus}(x) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}} \left(\sum_{\tau_h \in \tau^{NC}} I_i^{NC}(h, \beta_h^g l, x) + \sum_{\tau_h \in \tau^{CI}} I_i^{CI}(h, \beta_h^g l, x) \right) \tag{7.33}$$

where $\mathcal{Z}$ is the set of all partitions of the set of higher priority tasks, and

$$\beta_h^g l = \sum_{g \in \mathcal{G}_g(\tau_h) \wedge g \notin \mathcal{G}_g(\tau_i)} B(\tau_h, g). \tag{7.34}$$

Furthermore, since the tasks currently executing with effective priority higher than their base priority or executing non-preemptively, may lead to the scenario where indirect blocking is not allowed anymore for some higher priority jobs ($IBC_{h+1} - IBC_h = 0$), job τ_i^j will also have to wait until the last of these jobs finishes its execution. Therefore, the total amount of suspension time of job τ_i^j can be upper bounded as follows:

$$susp_i(x) = \left\lfloor \frac{\Omega_{i,hp}^{lsus}(x) + \Omega_{i,hp}^{gsus}(x)}{U_{\sigma_a}} \right\rfloor + \sum_{lg \in \mathcal{G}_l(\tau_i)} N_{i,lg} \max_{\substack{\tau_h \in hp(i) \wedge \\ i > h > LPC(lg)}} (C_h) + \sum_{g \in \mathcal{G}_g(\tau_i)} N_{i,g} \max_{\tau_h \in hp(i)} (C_h) \tag{7.35}$$

Besides the higher priority shared resource executions considered in the estimation of the upper bound on the suspension time, the execution of a job τ_i^j is interfered also by shared resource executions that are outside the domain considered there. We refer here to the interference from: (1) tasks with base priority $h < i$, $\min_{lg \in \mathcal{G}_l(\tau_i)}\big(LPC(lg)\big) + 1 < h < i$, during shared resource executions with priority ceiling equal to h, and (2) tasks with base priority $h < i$, $h \leq \min_{lg \in \mathcal{G}_l(\tau_i)}\big(LPC(lg)\big) + 1$, during any local shared resource executions. The interference due to these executions can be computed as follows:

$$I_{i,osh}^X(h,x) = \begin{cases} \left\| W_h^X(x, \beta_\tau^1(h,i)) \right\|^{x - C_i + 1} & \text{if } \min_{lg \in \mathcal{G}_l(\tau_i)} LPC(lg) + 1 < h < i, \\ \left\| W_h^X(x, \beta_\tau^2(h,i)) \right\|^{x - C_i + 1} & \text{if } h \leq \min_{lg \in \mathcal{G}_l(\tau_i)} LPC(lg) + 1. \end{cases} \tag{7.36}$$

with $\beta_\tau^1(h,i) = \sum_{\substack{lg \in \mathcal{G}_l(\tau_h) \wedge lg \notin \mathcal{G}_l(\tau_i) \\ \wedge LPC(lg) = h}} B(\tau_h, lg)$ and $\beta_\tau^2(h,i) = \sum_{\substack{lg \in \mathcal{G}_l(\tau_h) \\ \wedge lg \notin \mathcal{G}_l(\tau_i)}} B(\tau_h, lg)$.

Summing up, the total interference on task τ_i from higher priority tasks can be computed using Equation (7.28), where the $I_{i,nsh}^X(h,x)$ term is computed according to Equation (7.27) and the $I_{i,osh}^X(h,x)$ term is computed according to Equation (7.36).

Using the results above, we can determine the worst case response time of a task scheduled under P-HRP with LB-PCP by incorporating the suspension factors into the recurrence relation defined by Equation (7.29):

$$\begin{aligned} RT_i = & \frac{\Omega_i(RT_i) + mp(C_i)}{U_{\sigma^a}} + \delta_\sigma + susp_i(RT_i) \\ & + IB_{\tau_i}(RT_i) + DB_L^{\tau_i}(RT_i) + DB_G^{\tau_i}(RT_i) + IB_{\sigma_{l_0}}(RT_i) \end{aligned} \tag{7.37}$$

where $IB_{\tau_i}(RT_i)$ and $susp_i(RT_i)$ are computed using Equations (7.30) and (7.35), $\Omega_i(RT_i)$ is defined as explained in the previous paragraph and the rest of the terms are similar to those defined under PIP. The recurrence relation in Equation (7.37) starts with $RT_i = C_i$ and ends when the value of RT_i converges or $RT_i > D_i$, in which case the task is unschedulable.

Example 10. *We consider two applications, A_1 and A_2, each scheduled using a set of servers with parameters given in Table 7.1 (all time values are given in μs and servers are ordered based on priority) and we assume $m = 3$. Tasks in both applications share*

a global resource group with one resource and a maximum access time of $350\mu s$, a value quite large when compared to the capacity of the first server. Table 7.1 gives the worst-case response times of the servers for the case when P-HRP is used. For comparison purpose we also show what is the worst-case response time assuming that no resources are shared between applications.

Table 7.1: Numeric example for server response times under P-HRP

Server	Appl.	Capacity	Period	U	No resources	P-HRP
σ_1	A_1	500	2000	25%	500	849
σ_2	A_2	1000	4000	25%	1000	2147
σ_3	A_1	2500	10000	25%	2500	5182
σ_4	A_1	3000	12000	25%	4000	9181
σ_5	A_2	4000	16000	25%	7166	14895
σ_6	A_2	4500	30000	15%	9500	25963

It is obvious that global resource accesses have a cumulative effect on server response time. The response times of servers σ_2 to σ_5 are approximatively 2 times greater than for the non-blocking case (6^{th} column of Table 7.1) while for the last server this ratio increases. This is caused by the higher direct blocking of server σ_6 which leads us to the conclusion that, at least under this priority assignment, the achieved system utilization may be quite low.

Next, we consider task response times. Table 7.2 presents the task parameters for application A_1 (all time values are given in μs). Besides global resource access, two pairs of tasks also access a locally shared resource for at most $500\mu s$. The last two columns of Table 7.2 give the worst-case response times for the tasks assuming local and global resource sharing using the P-HRP with PIP and LB-PCP, respectively.

Table 7.2: Numeric example for task response times under P-HRP

Task	C	P	D	U	Res.	No resources	P-HRP	
							PIP	LB-PCP
τ_1	2800	40000	40000	7.00%	lg_1	23200	27547	27547
τ_2	3500	70000	70000	5.00%	lg_2	26000	36879	33147
τ_3	4000	85000	85000	4.71%	-	28000	47446	35348
τ_4	2300	100000	100000	2.30%	lg_2	21200	50044	33548
τ_5	4500	125000	125000	3.60%	lg_1	30000	60611	43514

There are several conclusions to be drawn from Table 7.2. First of all, resource access under PIP increases significantly the worst-case response time: with at least 18% for the highest priority task and with more than 100% for the lowest priority one). For LB-PCP, resource sharing is less expensive and we get an increase of the response time of at most 60%. These differences between the performances of the two protocols result from the lower indirect blocking time in LB-PCP compared to PIP.

Secondly, non-preemptive execution of low priority tasks during global resource accesses causes large blocking times for all higher priority tasks. This is especially visible for tasks τ_1 and τ_2. A second issue causing high delays for high priority tasks is the high indirect blocking time of these tasks when they request a local resource group. Note that indirect blocking is caused by lower priority tasks using the

local resource but may be caused also by usage of global resources, which determines non-preemptive execution of the tasks.

Finally, it can be seen that even when no resource sharing takes place, the worst-case response time is quite large. This is because in Equation (7.8), used for computing the response time in this case, we assume that servers running in parallel with the problem job's are idle and so the upper bound on allowed total interference is lowered significantly.

7.6. Performance Evaluation

In this section we present the results of the tests on the effectiveness of our protocol in scheduling workloads consisting of six hard real-time tasks scheduled on the set of servers $\sigma_{A_1} = \{\sigma_1, \sigma_2, \sigma_3\}$. These six tasks are part of an application A_1 executing on a shared platform with 3 processors. On the same platform there is a second application A_2 using the set of servers $\sigma_{A_2} = \{\sigma_4, \sigma_5, \sigma_6\}$. The parameters of the servers in the two sets are summarized in Table 7.3. The servers are ordered decreasingly by their priority.

Table 7.3: Server sets used for evaluating the performance of P-HRP

Server	**Application**	**Capacity**[μs]	**Period**[μs]	**U**
σ_1	A_1	500	2000	25%
σ_4	A_2	1000	4000	25%
σ_2	A_1	2500	10000	25%
σ_3	A_1	3000	12000	25%
σ_5	A_2	4000	16000	25%
σ_6	A_2	4500	30000	15%

For evaluating the performance of P-HRP we follow a simulative approach that considers the number of schedulable tasksets detected by the schedulability tests detailed in Section 7.5. among a randomly generated distribution of tasksets. The evaluation of the protocol uses a tool capable of determining if a set of tasks sharing a specified set of local and global resources is schedulable under P-HRP, for the cases when either LB-PCP or PIP is used at the second level of the HSF. Similar to the tool developed for testing schedulability under simple LB-PCP, this tool is also platform independent Java code and can be used as a stand-alone application or as a library, providing an interface which can be used by other applications. Basically, the tool receives as input the path of a file containing the set of servers in the system, the path to the file with the sets of tasks of each application running in the system, the path of a file containing the definitions of local and global shared resources and describing which tasks use which resources, the number of processors on which the tasks shall be scheduled and the name of the protocol to be used of arbitrating access to shared resources. Using this information, the tool tests if the taskset is schedulable using the equations presented in Section 7.5.

The experiments described further use this tool and test the schedulability of randomly generated tasksets containing six tasks each, with a variable taskset utilization. Each tasksets is assumed to represent application A_1 above.

For each experiment we generate 1000 tasksets. Every task is generated as follows:

- Task utilization is generated randomly using the UUnifast algorithm [49] adapted task utilizations in some specified interval. In our case, the taskset utilization increases from 10% to 75%, in steps of 5% and the task utilization U_i is between 1% and maximum taskset utilization.
- Depending on the exact experiment, task period T_i ($1 \leq i \leq n$) is generated according to a uniform random distribution between minimum or maximum server budget in the application and 100ms.
- The execution time C_i is computed as $C_i = U_i * T_i$.
- Task deadline D_i is generated using a uniform random distribution, in the range $[C_i + 0.9 * (T_i - C_i), T_i]$.
- The priorities of the tasks were set according to the RM order.
- For LB-PCP, we assume that a job can be indirectly blocked at most once: $IBT_{i+1} - IBT_i \leq 1, \forall\, i$.

7.6.1. Experiment 1: Global Critical Section Duration

One of the main factors that influence the response time of both servers and tasks is the duration of a global resource access. It is clear from Equations (7.20), (7.13) and (7.17) that whenever the duration of a global critical section increases, the response time of the servers may increase. Moreover, from Equations (7.29) and (7.37) we can see that any increase of the length of a global critical section will also lead to an increase in the response time of a task, no matter if it uses the global resource or not.

In this first experiment we consider that each task in application A_1 and each task in application A_2 uses a global resource for an amount of time varying between 10% and 100% of the minimum capacity of any server in the application's set. The number of tasks using the resource is 1, 3, 4 and 6.

In Figure 7.6 we see that if a single task (the medium priority one) in each application uses the global shared resource, the schedulability of the tasksets decreases very little as the length of the critical section increases, when either of the PIP or LB-PCP protocols is employed. This decrease is more visible for low taskset utilization and almost unobserved for high utilization. The task using the resource can affect the other tasks in the application only through indirect blocking and, if the overall taskset utilization is small, the ratio between the blocking time and actual execution time of each task is higher than for large taskset utilization values. For high priority tasks with early deadlines this leads to incapacity of meeting those deadlines.

In Figures 7.7 and 7.8, we see that as the number of application tasks using the same global resource is increased, increasing the duration of a global critical section decreases the percent of schedulable tasksets also for higher taskset utilizations. This is due to the additional direct blocking costs which influence the response time of the tasks using the resource, although the overall indirect blocking costs are reduced compared to the case in Figure 7.6.

Regarding the performance gap between the two protocols used together with P-HRP, namely PIP and LB-PCP, we note that LB-PCP always performs better than PIP. For example, with all tasks sharing the global resource, under PIP almost 34% of the tasksets with utilization 0.3 are schedulable, while, under LB-PCP, 34% of the tasksets with utilization 0.4 are schedulable. This means 33% better utilization of the processor.

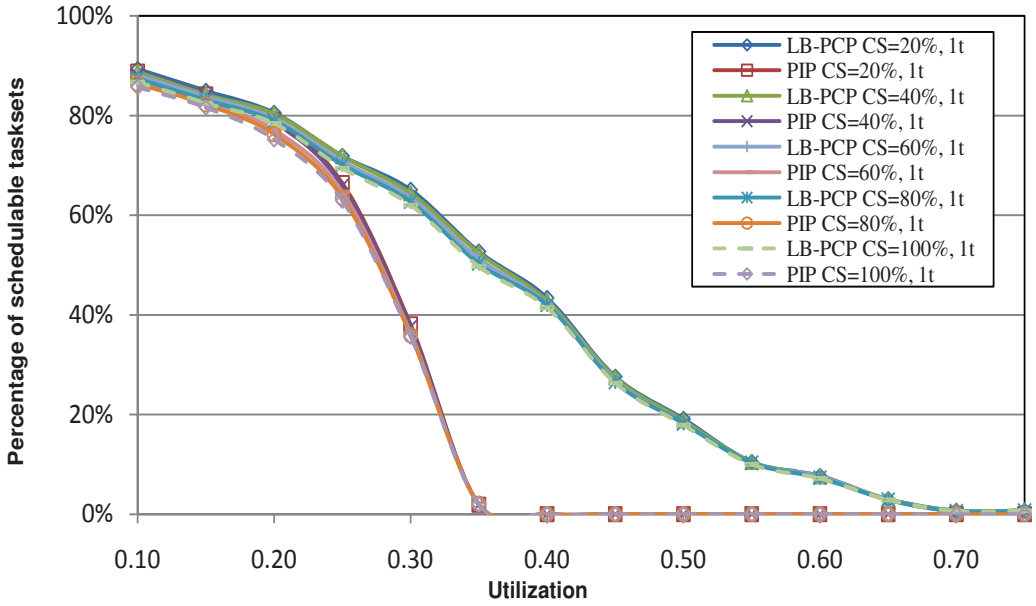


Figure 7.6: Percentage of schedulable tasksets for different critical section durations when a single task in each application uses the global resource

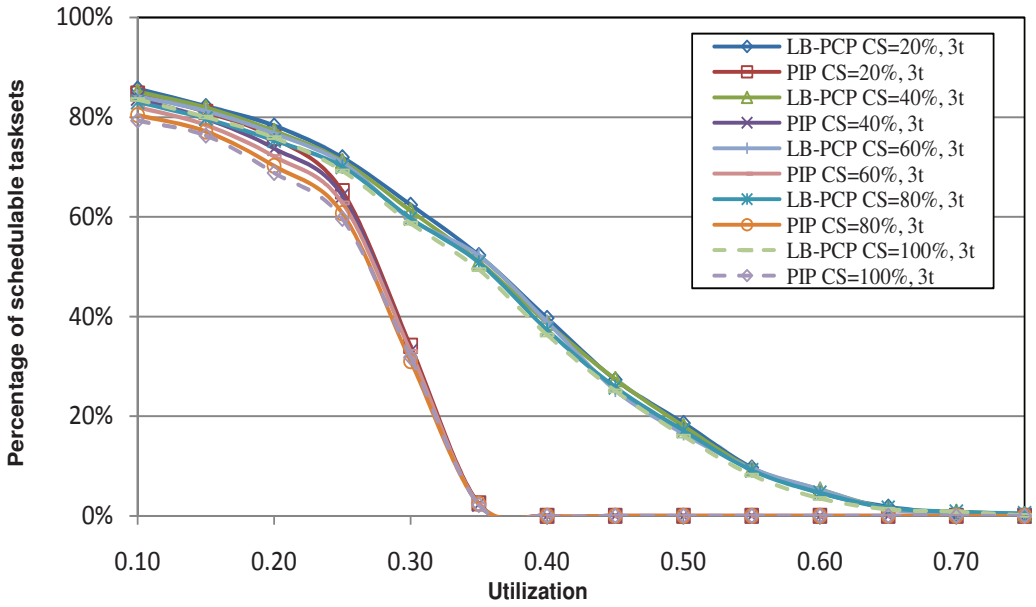


Figure 7.7: Percentage of schedulable tasksets for different critical section durations when half of the tasks in each application use the global resource

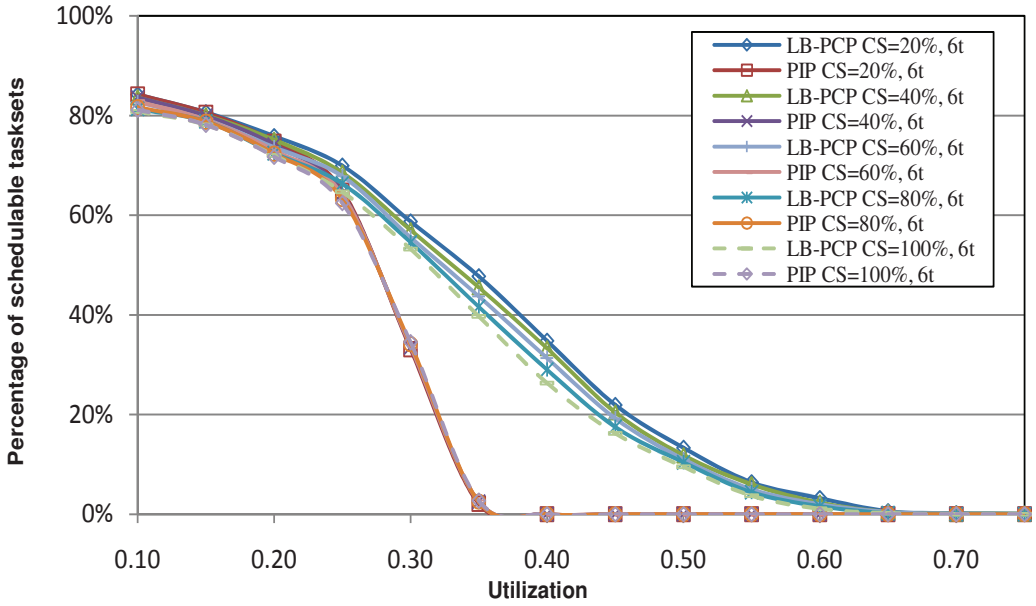


Figure 7.8: Percentage of schedulable tasksets for different critical section durations when all tasks in each application use the global resource

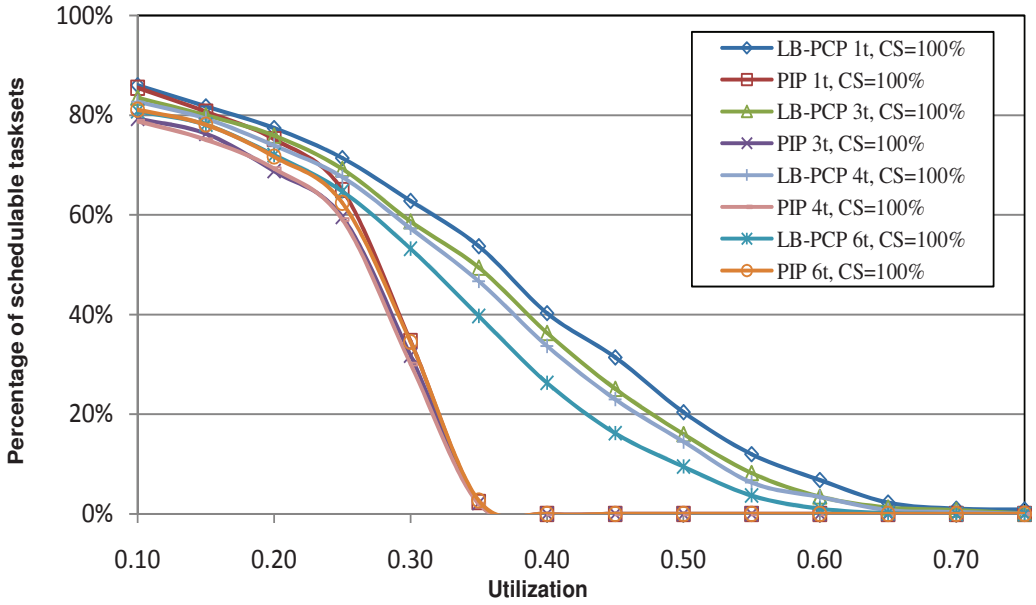


Figure 7.9: Percentage of schedulable tasksets for different number of tasks sharing a global resource

If we analyze schedulability as a function of tasks sharing the same resource for a time equal to the minimum capacity of any application server (see Figure 7.9), the effects of direct blocking are more obvious, with minimum percent of schedulable tasksets when all tasks use the resource and maximum when only one task uses the resource.

7.6.2. Experiment 2: Task Priority

Looking at Equations (7.22), (7.25), (7.28) and (7.33) it is clear that the priority of a task using a global resource is of great importance. A low priority task using a global resource will execute non-preemptively and may prevent a high priority one from running and finally may lead to at least one missed deadline. A high priority task may also delay other tasks since it keeps a server busy while waiting for the resource, a server which may have been used more efficiently by other tasks in the absence of other available servers. However, the number of tasks affected in the two scenarios is different. Another factor that influences schedulability is the lowest period of any task in the taskset. If this period is very close to the lowest period of any server in the application's set then we expect a lower schedulability success rate.

The graphs in Figure 7.10 show the percent of schedulable tasksets when the highest (HP) or the lowest (LP) priority task uses a global resource for a time equal to the minimum capacity of any application server. Furthermore, we test the case when the minimum task period in the taskset is equal to the minimum server period (MIN) and also when it is equal to the maximum server period (MAX). First thing to note is that when at least one task period is equal to the smallest server period, the schedulability rate drops with 50% under both LB-PCP and PIP, regardless the priority of the task using the global resource. It is fairly easy to estimate that if the task period is small its workload is high and consequently the resource demands of the application are high. As the resource supply provided by the server set is rather small, the application will need a lot of extra resources to be schedulable.

Furthermore, for this length of the critical section we can not see any major difference between the scenarios when the lowest priority task uses the global resource and those when the highest priority one does it. Figure 7.11 shows that even if the highest priority task uses the global resource for less than the entire budget of the server with the smallest period, the schedulability results do not change very much.

7.6.3. Experiment 3: Local Critical Section Duration

An important factor that influences the response time of a task is the duration of a local resource access. It is clear from almost all equations in Sections 7.5.4. and 7.5.5. that whenever the duration of a local critical section increases, the response time of the application tasks may increase.

In this experiment we consider that each task in application A_1 uses a local resource for an amount of time varying between 10% and 100% of the minimum capacity of any server in the application's set. The number of tasks using the resource is 3, 4 and 6.

Figure 7.12 shows the percentage of schedulable tasksets when a half of the tasks, selected randomly, use the locally shared resource. Note that the schedulability of the tasksets decreases very little as the length of the critical section increases, when either of the PIP or LB-PCP protocols is employed. Comparing with the situation when the shared resource was global, we can see that local resource sharing determines

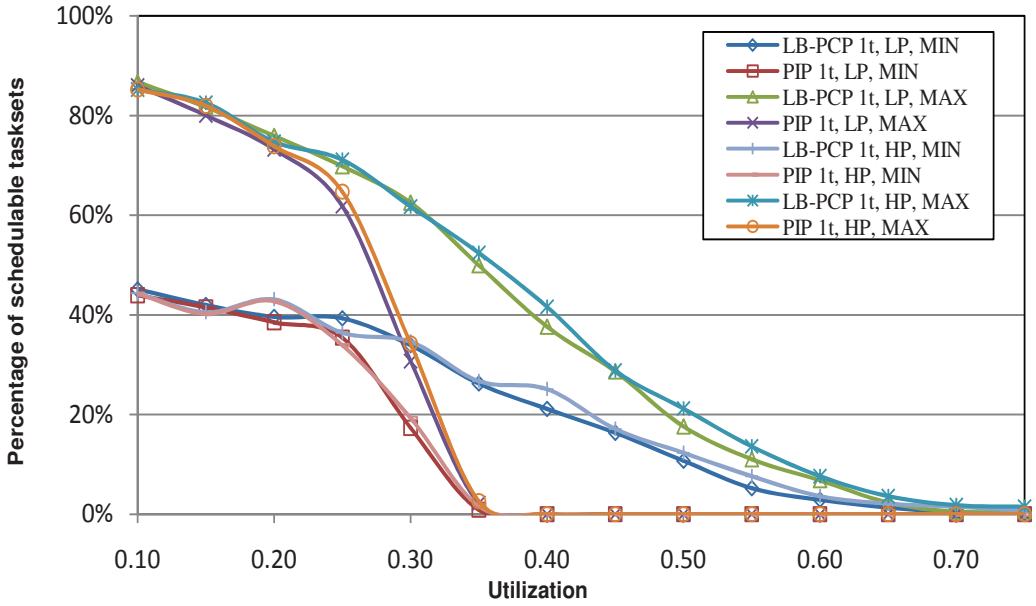


Figure 7.10: Percentage of schedulable tasksets when the task using a global resource has the highest or lowest priority

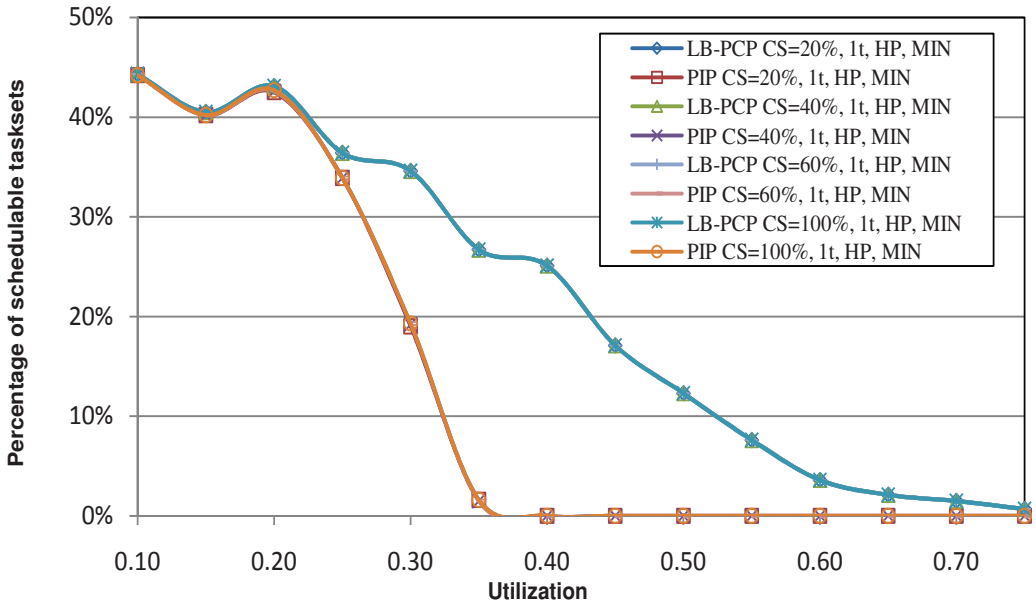


Figure 7.11: Percentage of schedulable tasksets when the highest priority task uses a global resource and its period is close to the minimum server period in the application's set

lower overheads. This was expected since in this case, delays may be induced only by tasks in the same application and are not influenced by other applications.

The effects of local direct blocking become more visible in Figure 7.13 where all tasks use the local resource. In Figure 7.13, we see that as the number of application tasks using the same local resource is increased, increasing the duration of a local critical section decreases the percent of schedulable tasksets for medium and high taskset utilization values. This is due to the additional direct blocking costs which influence the response time of the tasks using the resource. In previous cases, the cost of indirect blocking dominated the one of direct blocking, while in this situation we have no indirect blocking at all.

The performance gap between the PIP and LB-PCP protocols used together with P-HRP is similar to the global resource sharing case. Again, we note that LB-PCP always performs better than PIP. For example, with all tasks sharing the local resource for a small amount of time (20% of the smallest server budget), under PIP almost 39% of the tasksets with utilization 0.3 are schedulable, while, under LB-PCP, 39% of the tasksets with utilization 0.4 are schedulable. This means 33% better utilization of the processor.

If we compare schedulability rates based on the number of tasks sharing the same resource for a time equal to the minimum capacity of any application server (see Figure 7.14), the effects of direct blocking are more obvious, with minimum percent of schedulable tasksets when all tasks use the resource and maximum when only half of the tasks use the resource.

7.6.4. Experiment 4: Multiple Local Critical Sections

In this experiment we compare the cases when tasks in an application access multiple local critical sections. For this purpose we test what happens when different number of tasks use 2, 4, 6, 8 and 10 local resources. For each task the length of each critical section is generated as a percent of the task's execution time such that the sum of all critical sections durations equals the task's execution time. Furthermore, we test the cases when 50%, 75% and 100% of the tasks share the same resource.

Figure 7.15 shows the percentage of schedulable tasksets when 3 tasks (50%) share the same local resource. It can be noted that for 2 up to 6 critical sections per task, under LB-PCP the schedulability success rate is higher than under PIP, while for 8 and 10 critical sections PIP outperforms LB-PCP for low and medium utilization tasksets, but for higher utilization LB-PCP is still better. This is the effect of the high suspension time for each of the tasks. If we increase the number of tasks using the same resource to 6 (see Figure 7.16) we see that in the best case, LB-PCP is just as good as PIP, but almost always PIP is better than LB-PCP.

If we analyze the schedulability as a function of the number of tasks using the same resource (Figures 7.17 and 7.18), we see that for a small number of allowed indirect blockages (1) and a large percent of tasks using the resource, with PIP we get a higher number of schedulable tasksets than when LB-PCP is used.

In the worst case, with 8 locally shared resources and all 6 tasks using each resource, under PIP approximatively 49% of the tasksets with utilization 0.2 are schedulable, while under LB-PCP, 49% of the tasksets with utilization 0.15 are schedulable. This means 33% better processor utilization under PIP. In the best case, when just half of the tasks share 4 local resources, under LB-PCP we get 33% better processor utilization since approximatively 20% of the tasksets with 0.35 utilization are schedulable, in opposition with just 0.3 taskset utilization achieved with PIP.

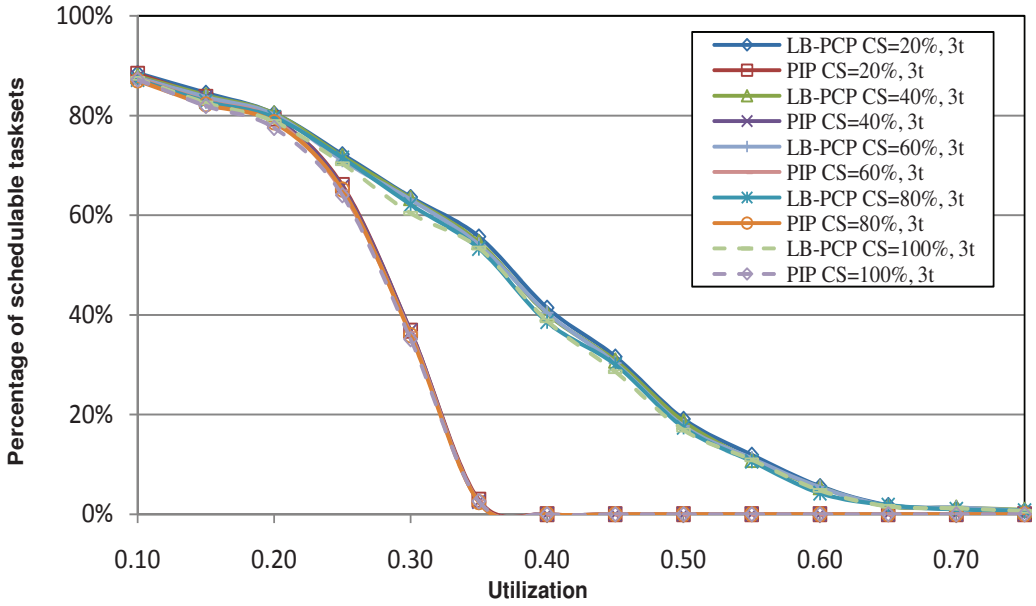


Figure 7.12: Percentage of schedulable tasksets for different critical section durations when half of the tasks in each application use the local resource

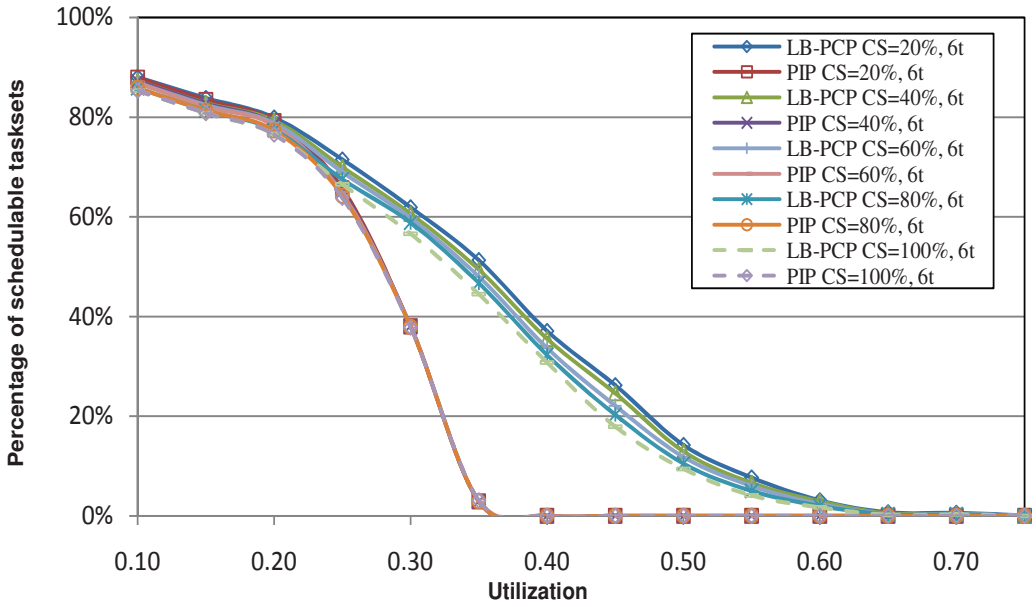


Figure 7.13: Percentage of schedulable tasksets for different critical section durations when all tasks in each application use the local resource

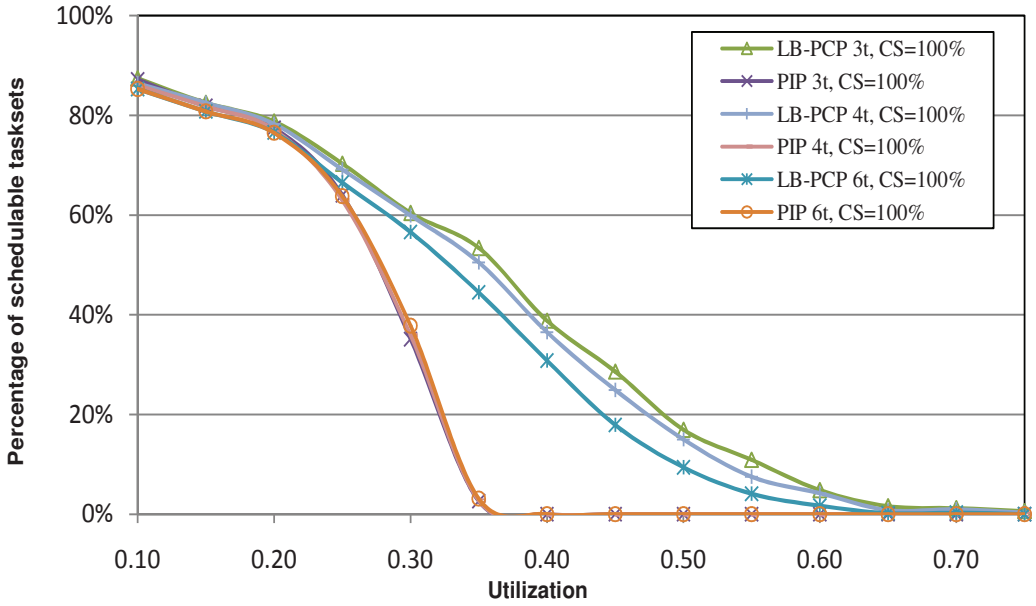


Figure 7.14: Percentage of schedulable tasksets for different number of tasks sharing a local resource

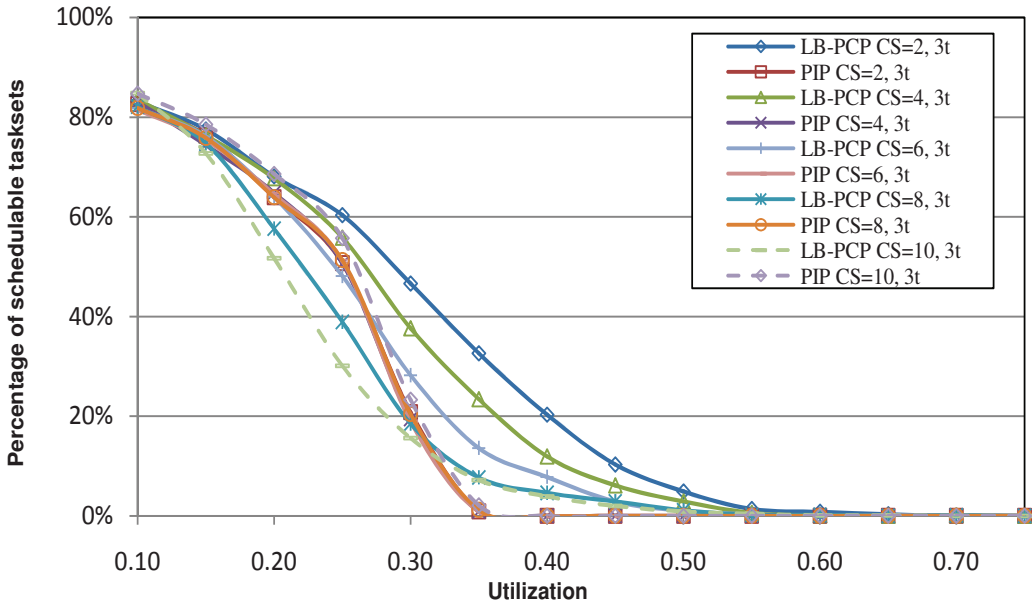


Figure 7.15: Percentage of schedulable tasksets for different numbers of critical sections when half of the tasks in each application use each local resource

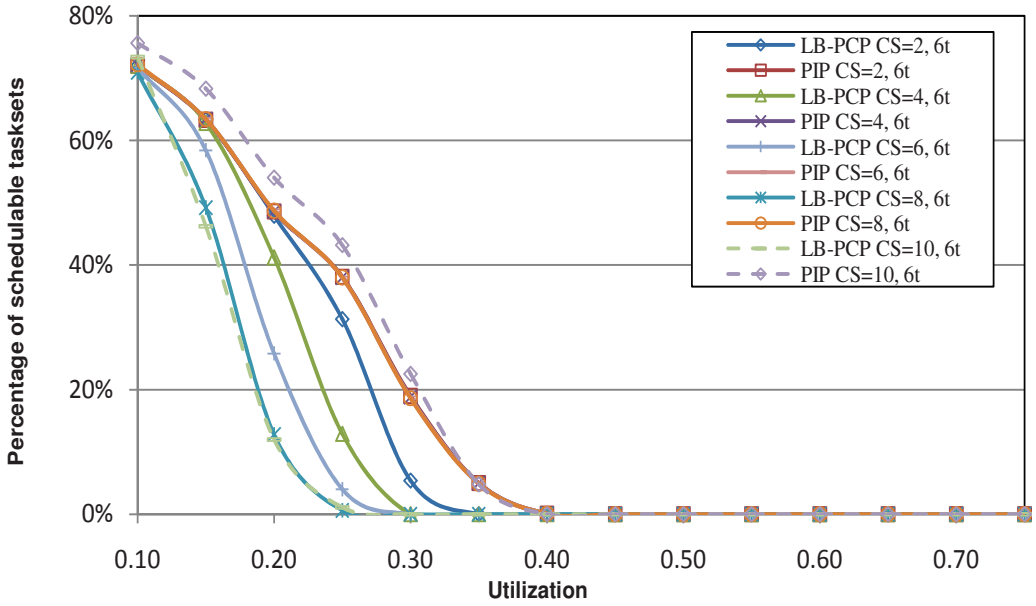


Figure 7.16: Percentage of schedulable tasksets for different numbers of critical sections when all tasks in each application use each local resource

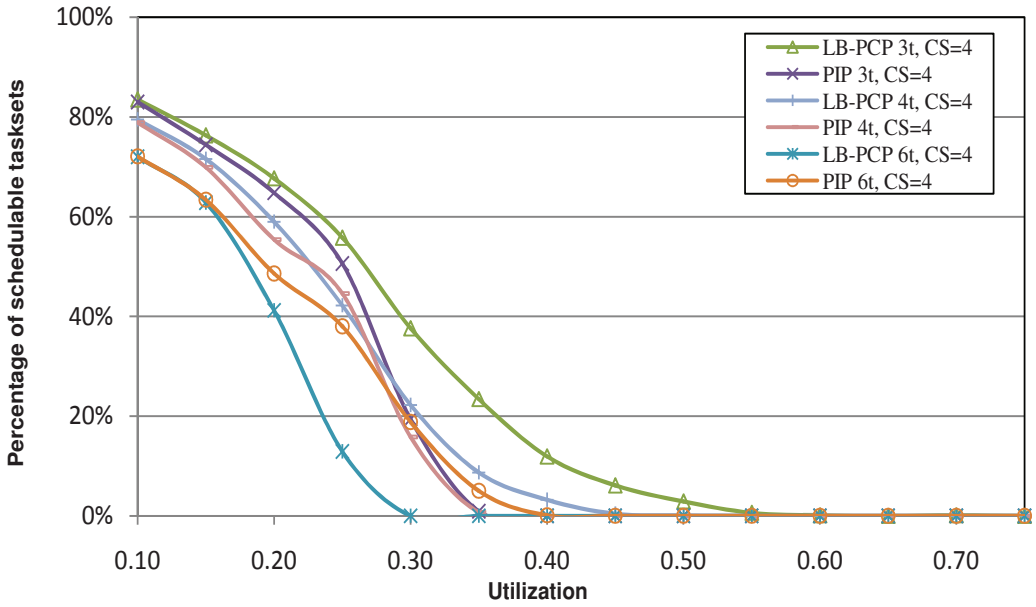


Figure 7.17: Percentage of schedulable tasksets for different number of tasks sharing 4 local resources

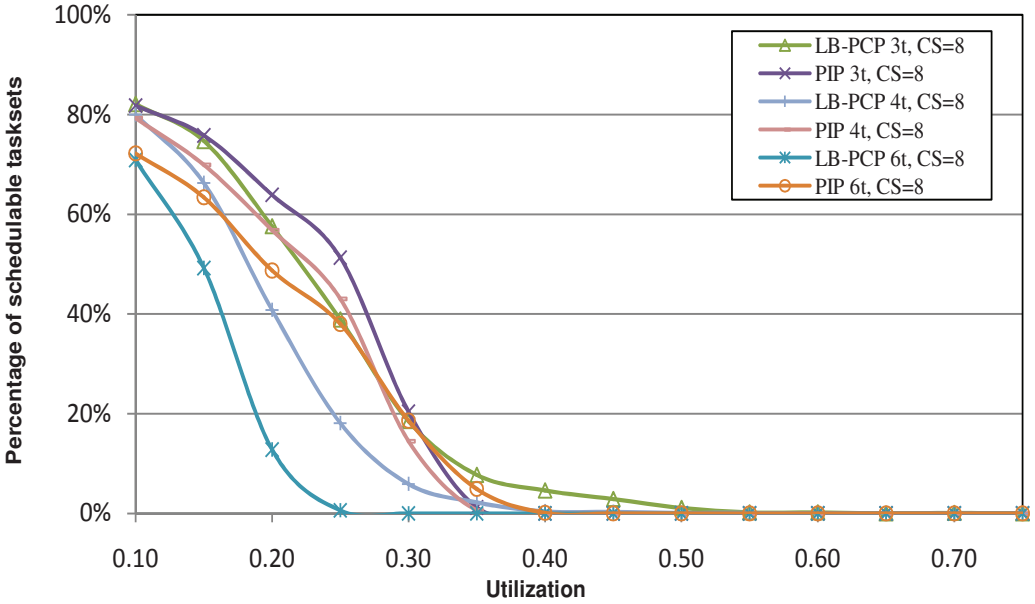


Figure 7.18: Percentage of schedulable tasksets for different number of tasks sharing 8 local resources

From this experiment we can conclude that LB-PCP behaves well for tasksets with large utilization even when several resources are shared, but for low and medium taskset utilizations is better than PIP only if the number of critical sections is not very high and the resource contention is not at its peek.

7.6.5. Experiments Summary

In previous sections we presented the results of a set of experiments performed in order to give some insight on the performance of P-HRP when either LB-PCP or PIP is used as resource sharing protocol by the application level scheduler. Based on all these experiments, we can draw several conclusions:

- For global resource accesses, the performance of P-HRP decreases as the duration of critical sections increases and also as the number of tasks requesting a resource increases. Further, P-HRP associated with LB-PCP always performs better than when associated with PIP.
- The priority of the task using a global resource has little influence on the number of tasksets deemed schedulable by P-HRP.
- In case of global resource accesses, if the period of at least one application task is almost equal to the smallest server period in the application's set, the number of tasksets deemed schedulable is almost half the number of schedulable tasksets when the minimum task period is much greater than the minimum server period.
- For local resource accesses, the performance of P-HRP decreases as the duration of critical sections increases and also as the number of tasks requesting a re-

source increases. Further, P-HRP associated with LB-PCP always performs better than when associated with PIP.

- If tasks use a large number of local resources and each resource is requested by many tasks, P-HRP associated with PIP performs better than if associated with LB-PCP

7.7. Concluding Remarks

In this chapter we have introduced a new resource sharing protocol for multi-core hierarchical systems, called Parallel Hierarchical Resource Policy (P-HRP), and developed the response-time analysis for it assuming that global fixed-priority preemptive multiprocessor scheduling is used. The RTA shows that local and global resource accesses have a cumulative effect on the schedulability of both tasks and servers and that lower priority interference can lead to important increases of the response times of high priority tasks.

The proposed protocol distinguishes itself from existing work in the research area by the followings:

c1. It allows application tasks running in a multi-core HSF to make mutually exclusive accesses to locally or globally shared resources.

c2. It allows parallel processing within each application and thus enables a more efficient use of the processing capabilities of multi-core processors.

c3. It can be used along with existing multiprocessor resource sharing protocols like PIP, P-PCP and LB-PCP.

8. CONCLUSIONS AND FUTURE WORK

8.1. Summary

This thesis is structured in six main chapters.

Chapter 2. provides the background on the concepts and terminology regarding real-time scheduling and shows the current status of the multiprocessor real-time scheduling area. The presentation covers a description of the principal task models used in real-time scheduling analysis and the significant task parameters. The notions necessary to understand the domain are defined and a taxonomy of multiprocessor scheduling algorithms is presented. Although one would expect that by making the step from uniprocessor to multiprocessor systems, it would be easier to guarantee systems' timing constraints, a series of anomalies can emerge at this transition. These anomalies along with eloquent examples are also introduced in this chapter. Having laid the foundations, we take a further step into a more restricted area of hierarchical real-time scheduling as a mean to attain temporal isolation in shared multiprocessor/multi-core platforms. The chapter ends with a presentation of existing resource sharing protocols for multiprocessor systems. Most of these protocols are designed for partitioned multiprocessor real-time systems and only few address the issues specific to globally scheduled systems.

Most of the multiprocessor scheduling techniques presented in Chapter 2. are not exact and may take false negative scheduling decisions. Chapter 3. proposes formal verification as an alternative solution. In order to study schedulability analysis using formal verification and modeling techniques, a definition of the main challenges and problems that need to be addressed is given in the opening of Chapter 3. Furthermore, the chapter gives an overview of three main formalisms used for schedulability analysis: timed automata, Petri nets and ACSR. The features of these formalisms are analyzed to show how appropriate is each of them for modeling multiprocessor real-time systems. Besides these three formalisms, several other formal approaches are presented. As the method proposed in this thesis uses timed automata, we also dedicated a section for introducing their syntax and semantics and also related our methods to previous work using timed automata for real-time scheduling.

Our objective was to benefit from this experience and define a formal model for multi-core contract-based scheduling. We introduced the concept of contract-base scheduling as a hierarchical scheduling scheme aiming at ensuring temporal isolation of real-time and non-real-time independent applications. The model was used for providing a method for schedulability analysis using model checking for multi-core systems with preemption and migration enabled. Based on the defined timed automata model, we give an exact schedulability test for multi-core contract-based scheduling with applications encompassing only independent tasks and fixed execution time. Furthermore, we extended the method for applications with task precedence constraints and variable task execution requirements. This new method gives only a sufficient schedulability test, but is based on a model closer to real applications. The methods proposed in this chapter can be applied for testing schedulability of real-time component-based systems. The scalability of the methods shows that they can be applied with success for real applications. In this sense we also presented a case study for an

H.264 decoder.

Using the formal model introduced in Chapter 4., in Chapter 5. we develop a method for generating the time partitions necessary to guarantee timing constraints of applications scheduled using a two-level scheduling hierarchy. The method assumes a simple task model, with independent tasks with fixed execution requirement, and uses fixed priorities for these tasks. The proposed method uses the simulation feature of UPPAAL to find an initial time partition which is refined in further iterations through model checking until the resulted time partitions satisfy the timing constraints of the application. As the method requires a simulation stage we had to find a time limit of the simulation. Consequently, we extended previous results in the real-time scheduling theory in order to find a feasibility interval for the special case of multiprocessor hierarchical systems.

This thesis was also dedicated to resource sharing in multi-core/multiprocessor real-time systems in general and, hierarchical multi-core systems, in particular. We have focused on globally scheduled systems using a fixed-priority preemptive scheduling algorithm.

Chapter 6. proposes the Limited Blocking Priority Ceiling Protocol (LB-PCP), a resource sharing protocol for simple, globally scheduled systems. The main idea of LB-PCP came from analyzing the limitations of PIP in globally scheduled multiprocessor systems, namely the potentially unbounded number of blockages of high priority tasks due to shared resource executions of low priority ones. Consequently, LB-PCP limits these blockages by controlling the instant when a low priority task may access a resource. To enable the schedulability analysis under LB-PCP, we also presented a schedulability test for it based on response time analysis of the tasks to be scheduled. The performance evaluation presented later in Chapter 6. showed that, in many situations, the proposed protocol is superior to the existing P-PCP protocol.

Further, Chapter 7. extends LB-PCP for a two-level hierarchical scheduling framework. We introduced the Parallel Hierarchical Resource Policy (P-HRP), which defines a set of rules for resource sharing at each level of the hierarchy. These rules are based on PIP, for the first scheduling level, and on PIP or LB-PCP for the second scheduling level. Because a protocol is not complete without a schedulability test, we also proposed such a test. Using this test we evaluated the performance of P-HRP for various test cases.

8.2. Contributions

One of the major research directions in this thesis focused on the usage of model-based techniques for real-time scheduling. As a result, the thesis proposes a model-based methodology to address a major problem in the formal analysis of contract-based systems: verification of real-time properties to prove whether individual deadlines for tasks are satisfied. The key contributions in this direction are:

- An exact model checking method using timed automata for the real-time verification of preemptive multi-core contract-based-scheduling with independent tasks and fixed execution time:
  - it formalizes multi-core contract-based scheduling using timed automata with discrete time semantics,
  - the method is exact, while tests provided by classical scheduling theory are only sufficient,
  - the method can be used with any global multiprocessor scheduling algorithm.

- A conservative approximation method for verification of preemptive multi-core contract-based-scheduling with task dependence relations and variable execution time:
  - considering variable execution time improves the confidence in the proposed method since several anomalies can affect the scheduling process,
  - the method uses a model of the application where tasks have precedence constraints between them, being closer to a real application.
- A method for generating the multi-core temporal partitions that can satisfy the computing requirements of an application or component for a two-level hierarchical system:
  - the method uses the characteristics of the tasks encompassing the application and determines low processor utilization time partitions,
  - we consider that such a method is required since currently such temporal partitions are designed manually and consequently are subject to human error.
- We prove that a feasible schedule obtained with a preemptive fixed-priority scheduling algorithm on periodic deadline constrained tasksets is periodic on a multi-core time partition and we also give the feasibility interval for these schedules.

A second research direction of the thesis concentrated on resource sharing in globally scheduled multiprocessor systems. First, the thesis proposes a protocol for resource sharing in a single level multiprocessor real-time system and then extends the contributions to hierarchical multiprocessor systems. The key contributions in this area are:

- A protocol for mutually exclusive access to shared data in a real-time system scheduled according to a global preemptive fixed priority policy:
  - the protocol is based on priority inheritance but reduces the negative effects of PIP protocols on multiprocessors, as it controls the blocking time of each task,
  - the protocol is configurable, such that one can increase or decrease the level of parallelism in the system, depending on the characteristics of the workload,
  - a schedulability test accompanies the protocol, such that one can give pre-runtime guarantees that an application will meet its timing constraints,
  - the performance evaluation shows that the proposed protocol is better than other state-of-the-art protocols.
- A protocol for mutually exclusive access to shared data in a multi-core two-level hierarchical real-time system which uses at each scheduling level a global preemptive fixed priority policy:
  - it allows parallel executions at both scheduling levels, a feature that no other existing resource sharing protocol for hierarchical frameworks presents,
  - the protocol can be used along with existing multiprocessor resource sharing protocols like PIP, P-PCP and LB-PCP,
  - two schedulability tests are given for the protocol, one for when simple PIP is used at both levels and, one for the case when LB-PCP is employed for controlling resource sharing at the second level of the hierarchical framework.

8.3. Next Steps

The next steps relate to the model-based analysis of the proposed resource sharing protocols. Further, to prove the efficiency of the proposed protocols, a prototype implementation of them will be developed in a real operating system. Using this implementation, through extensive benchmarking, we expect to show that they can be employed to increase the performance of real-time applications running on multicore systems. Future research directions also include extending the Limited Blocking Priority Ceiling Protocol for dynamic priority assignment policies.

Bibliography

[1] ***, FFmpeg, http://ffmpeg.org/.

[2] ***, ISO/IEC 14496-10. Coding of audio-visual objects. Part 10: Advanced Video Coding.

[3] ***, Memtime utility, http://freshmeat.net/projects/memtime/.

[4] ***, Video Trace Library - YUV 4:2:0 Video Sequences, http://trace.eas.asu.edu/yuv.

[5] Distributed Real-time Embedded Analysis Method - DREAM, http://dre.sourceforge.net/, January 2010.

[6] ***, LG launches first and fastest dula-core smartphone, Online, December 2010.

[7] ***, ARM Cortex-A15 MPCore, http://www.arm.com/products/processors/cortex-a/cortex-a15.php, January 2011.

[8] Y. Abdeddaïm, A. Kerbaa, and O. Maler, Task Graph Scheduling Using Timed Automata, in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pp. 237–244, IEEE Computer Society, Washington, DC, USA, 2003, ISBN 0-7695-1926-1.

[9] Y. Abdeddaïm and O. Maler, Preemptive Job-Shop Scheduling Using Stopwatch Automata, in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 113–126, Springer-Verlag, London, UK, 2002, ISBN 3-540-43419-4.

[10] L. Abeni and G. Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, in *RTSS '98: Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 4–13, IEEE Computer Society, Washington, DC, USA, 1998, ISBN 0-8186-9212-X.

[11] L. Almeida and P. Pedreiras, Scheduling within Temporal Partitions: Response-Time Analysis and Server Design, in *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded software*, pp. 95–103, ACM, New York, NY, USA, 2004, ISBN 1-58113-860-1.

[12] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, The Algorithmic Analysis of Hybrid Systems, *Theoretical Computer Science*, vol. 138: pp. 3–34, 1995.

[13] R. Alur and D. L. Dill, A Theory of Timed Automata, *Theoretical Computer Science*, vol. 126 (2): pp. 183–235, 1994, ISSN 0304-3975.

[14] R. Alur and T. A. Henzinger, Modularity for Timed and Hybrid Systems, in *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pp. 74–88, Springer-Verlag, London, UK, 1997, ISBN 3-540-63141-0.

[15] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, TIMES - A Tool for Modelling and Implementation of Embedded Systems, in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 460–464, Springer-Verlag, London, UK, 2002, ISBN 3-540-43419-4.

[16] J. H. Anderson and J. M. Calandrino, Parallel Real-Time Task Scheduling on Multicore Platforms, in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pp. 89–100, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2761-2.

[17] J. H. Anderson, J. M. Calandrino, and U. C. Devi, Real-Time Scheduling on Multicore Platforms, in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 179–190, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2516-4.

[18] B. Andersson, *Static-Priority Scheduling on Multiprocessors*, Phd thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2003.

[19] B. Andersson, S. Baruah, and J. Jonsson, Static-Priority Scheduling on Multiprocessors, in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pp. 193–202, IEEE Computer Society, Washington, DC, USA, 2001, ISBN 0-7695-1420-0.

[20] B. Andersson and K. Bletsas, Sporadic Multiprocessor Scheduling with Few Preemptions, in *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pp. 243–252, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3298-1.

[21] B. Andersson and J. Jonsson, Fixed-Priority Preemptive Multiprocessor Scheduling: to Partition or not to Partition, in *RTCSA '00: Proceedings of the 7th International Conference on Real-Time Systems and Applications*, pp. 337–346, IEEE Computer Society, Washington, DC, USA, 2000, ISBN 0-7695-0930-4.

[22] B. Andersson and J. Jonsson, Some Insights on Fixed-Priority Preemptive Non-Partitioned Multiprocessor Scheduling, in *RTSS '00: Proceedings of the 21st IEEE International Real-Time Systems Symposium*, pp. 53–56, IEEE Computer Society, Washington, DC, USA, November 2000, ISBN 0-7695-0900-2.

[23] A. E. E. C. (ARINC), Avionics Application Software Standard Interface (Arinc Specification 653), March 2006.

[24] T. P. Baker, Stack-Based Scheduling of Realtime Processes, *Real-Time Systems Journal*, vol. 3 (1): pp. 67–100, March 1991.

[25] T. P. Baker, An Analysis of Fixed-Priority Schedulability on a Multiprocessor, *Real-Time Systems*, vol. 32 (1-2): pp. 49–71, 2006, ISSN 0922-6443.

[26] S. Baruah, Techniques for Multiprocessor Global Schedulability Analysis, in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pp. 119–128, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3062-1.

[27] S. Baruah, An Improved Global EDF Schedulability Test for Uniform Multiprocessors, in *RTAS '10: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 184–192, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-0-7695-4001-6.

[28] S. Baruah and T. Baker, Global EDF Schedulability Analysis of Arbitrary Sporadic Task Systems, in *ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 3–12, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3298-1.

[29] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, Proportionate Progress: a notion of Fairness in Resource Allocation, *Algorithmica*, vol. 15: pp. 600–625, July 1995.

[30] S. Baruah and N. Fisher, The Partitioned Multiprocessor Scheduling of Sporadic Task Systems, in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pp. 321–329, IEEE Computer Society, Washington, DC, USA, 2005, ISBN 0-7695-2490-7.

[31] S. Baruah and J. Goossens, The EDF Scheduling of Sporadic Task Systems on Uniform Multiprocessors, in *RTSS '08: Proceedings of the 29th Real-Time Systems Symposium*, pp. 367–374, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3477-0.

[32] S. Baruah, J. Goossens, and G. Lipari, Implementing Constant-Bandwidth Servers upon Multiprocessor Platforms, in *RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 154–163, IEEE Computer Society, Washington, DC, USA, 2002, ISBN 0-7695-1739-0.

[33] S. Baruah and G. Lipari, Executing Aperiodic Jobs in a Multiprocessor Constant-Bandwidth Server Implementation, in *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, vol. 0, pp. 109–116, IEEE Computer Society, Los Alamitos, CA, USA, 2004, ISSN 1068-3070.

[34] S. Baruah and G. Lipari, A Multiprocessor Implementation of the Total Bandwidth Server, in *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*, vol. 1, pp. 40–49, IEEE Computer Society, Los Alamitos, CA, USA, 2004, ISBN 0-7695-2132-0.

[35] A. Bastoni, B. Brandenburg, and J. Anderson, Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability, in *OSPERT '10: Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 33–44, 2010.

[36] A. Bastoni, B. Brandenburg, and J. Anderson, An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers, in *RTSS '10: Proceedings of the 31st IEEE Real-Time Systems Symposium*, pp. 14–24, IEEE Computer Society, Washington, DC, USA, December 2010.

[37] M. Behnam, T. Nolte, M. Asberg, and R. J. Bril, Overrun and Skipping in Hierarchically Scheduled Real-Time Systems, in *RTCSA '09: Proceedings of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 519–526, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3787-0.

[38] M. Behnam, T. Nolte, and R. J. Bril, A New Approach for Global Synchronization in Hierarchical Scheduled Real-Time Systems, in *ECRTS'09: Work-in-Progress (WiP) session of the 21st Euromicro Conference on Real-Time Systems*, pp. 41–44, IEEE Computer Society, Washington, DC, USA, July 2009.

[39] M. Behnam, I. Shin, T. Nolte, and M. Nolin, SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems, in *EMSOFT '07: Proceedings of the 7th International Conference on Embedded Software*, pp. 279–288, ACM, New York, NY, USA, 2007, ISBN 978-1-59593-825-1.

[40] H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie, A Process Algebraic Approach to the Schedulability Analysis of Real-Time Systems, *Real-Time Systems*, vol. 15 (3): pp. 189–219, 1998, ISSN 0922-6443.

[41] J. Bengtsson and W. Yi, *Lectures on Concurrency and Petri Nets*, *Lecture Notes in Computer Science*, vol. 3098, chap. Timed Automata: Semantics, Algorithms and Tools, pp. 87–124, Springer Berlin / Heidelberg, July 2004, ISBN 978-3-540-22261-3.

[42] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat, Reachability Problems and Abstract State Spaces for Time Petri Nets with Stopwatches, *Discrete Event Dynamic Systems*, vol. 17 (2): pp. 133–158, 2007, ISSN 0924-6703.

[43] M. Bertogna, *Real-Time Scheduling Analysis for Multiprocessor Platforms*, PhD Thesis, Scuola Superiore Sant'Anna, Pisa, Italy, May 2008.

[44] M. Bertogna and M. Cirinei, Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms, in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pp. 149–160, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3062-1.

[45] M. Bertogna, M. Cirinei, and G. Lipari, Improved Schedulability Analysis of EDF on Multiprocessor Platforms, in *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pp. 209–218, IEEE Computer Society, Washington, DC, USA, 2005, ISBN 0-7695-2400-1.

[46] M. Bertogna, N. Fisher, and S. Baruah, Resource-Sharing Servers for Open Environments, *IEEE Transactions on Industrial Informatics*, vol. 5 (3): pp. 202–219, August 2009.

[47] E. Bini, M. Bertogna, and S. Baruah, Virtual Multiprocessor Platforms: Specification and Use, in *RTSS '09: Proceedings of the 30th IEEE Real-Time Systems Symposium*, pp. 437–446, IEEE Computer Society, Washington, DC, USA, December 2009.

[48] E. Bini, G. Buttazzo, and M. Bertogna, The Multi Supply Function Abstraction for Multiprocessors, in *RTCSA '09: Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 294–302, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3787-0.

[49] E. Bini and G. C. Buttazzo, Measuring the Performance of Schedulability Tests, *Real-Time Systems Journal*, vol. 30 (1-2): pp. 129–154, 2005, ISSN 0922-6443.

[50] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, A Flexible Real-Time Locking Protocol for Multiprocessors, in *RTCSA '07: Proceedings of the 13th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 47–56, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2975-5.

[51] M. Bordin, M. Panunzio, and T. Vardanega, Fitting Schedulability Analysis Theory into Model-Driven Engineering, in *ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 135–144, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3298-1.

[52] B. B. Brandenburg and J. H. Anderson, Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors, in *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pp. 61–70, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2914-3.

[53] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario, Modeling Flexible Real Time Systems with Preemptive Time Petri Nets, *ECRTS '03: Proceedings of the 15th Euromicro Conference on Real-Time Systems*, vol. 0: pp. 279–286, 2003.

[54] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd ed., 2001, ISBN 0201729881.

[55] J. M. Calandrino and J. H. Anderson, Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study, in *ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 299–308, IEEE Computer Society, Washington, DC, USA, 2008, ISSN 1068-3070.

[56] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms, in *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pp. 247–258, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2914-3.

[57] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, $LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers, in *RTSS '07: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pp. 111–126, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2761-2.

[58] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah, *Handbook on Scheduling Algorithms, Methods, and Models*, chap. A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms, Chapman Hall/CRC, Boca, 2004.

[59] F. Cassez and K. G. Larsen, The Impressive Power of Stopwatches, in *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pp. 138–152, Springer-Verlag, London, UK, 2000, ISBN 3-540-67897-2.

[60] Y. Chang, R. Davis, and A. Wellings, Schedulability Analysis for a Real-time Multiprocessor System Based on Service Contracts and Resource Partitioning, Technical Report YCS-2008-432, Computer Science Department, University of York, 2008.

[61] C.-M. Chen and S. K. Tripathi, Multiprocessor Priority Ceiling Based Protocols, Technical Report UMIACS-TR-94-42, 1994.

[62] H. Cho, B. Ravindran, and E. D. Jensen, An Optimal Real-Time Scheduling Algorithm for Multiprocessors, in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pp. 101–110, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2761-2.

[63] A. Cimatti, L. Palopoli, and Y. Ramadian, Symbolic Computation of Schedulability Regions Using Parametric Timed Automata, in *RTSS '08: Proceedings of the 29th Real-Time Systems Symposium*, pp. 80–89, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3477-0.

[64] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine, TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems, in *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pp. 391–395, Springer-Verlag, London, UK, 2001, ISBN 3-540-42345-1.

[65] S. Collette, L. Cucu, and J. Goossens, Integrating Job Parallelism in Real-Time Scheduling Theory, *Information Processing Letters*, vol. 106 (5): pp. 180–187, 2008, ISSN 0020-0190.

[66] L. Cucu and J. Goossens, Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors, in *ETFA '06: IEEE Conference on Emerging Technologies and Factory Automation*, pp. 397–404, September 2006, ISBN 0-7803-9758-4.

[67] L. Cucu and J. Goossens, Feasibility Intervals for Multiprocessor Fixed-Priority Scheduling of Arbitrary Deadline Periodic Systems, in *DATE '07: Proceedings of the 10th Conference on Design, Automation and Test in Europe*, pp. 1635–1640, EDA Consortium, San Jose, CA, USA, 2007, ISBN 978-3-9810801-2-4.

[68] R. I. Davis and A. Burns, Resource Sharing in Hierarchical Fixed Priority Pre-Emptive Systems, in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pp. 257–270, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2761-2.

[69] R. I. Davis and A. Burns, Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems, *RTSS '09: Proceedings of the 30th IEEE International Real-Time Systems Symposium*, pp. 398–409, 2009, ISSN 1052-8725.

[70] Z. Deng and J. W.-S. Liu, Scheduling real-time applications in an open environment, in *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 308–319, IEEE Computer Society, Washington, DC, USA, 1997, ISBN 0-8186-8268-X.

[71] U. C. Devi, H. Leontyev, and J. H. Anderson, Efficient Synchronization under Global EDF Scheduling on Multiprocessors, in *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pp. 75–84, IEEE Computer Society, Washington, DC, USA, 2006, ISBN 0-7695-2619-5.

[72] F. Dewan and N. Fisher, Approximate Bandwidth Allocation for Fixed-Priority-Scheduled Periodic Resources, in *RTAS '10: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 247–256, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-1-4244-6690-0, ISSN 1080-1812.

[73] S. K. Dhall and C. L. Liu, On a Real-Time Scheduling Problem, *Operations Research*, vol. 26 (1): pp. 127–140, January–February 1978.

[74] A. Easwaran and B. Andersson, Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling, in *RTSS '09: Proceedings of the 30th Real-Time Systems Symposium*, pp. 377–386, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3875-4.

[75] A. Easwaran and B. Andersson, Scheduling Sporadic Tasks on Multiprocessors with Mutual Exclusion Constraints, in *ICPPW '09: Proceedings of the International Conference on Parallel Processing Workshops*, pp. 50–57, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3803-7.

[76] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, A Compositional Scheduling Framework for Digital Avionics Systems, in *RTCSA '09: Proceedings of the 15th International Workshop on Real-Time Computing Systems and Applications*, pp. 371–380, IEEE Computer Society, Los Alamitos, CA, USA, 2009, ISSN 1533-2306.

[77] A. Easwaran, I. Shin, and I. Lee, Optimal Virtual Cluster-Based Multiprocessor Scheduling, *Real-Time Systems*, vol. 43 (1): pp. 25–59, 2009, ISSN 0922-6443.

[78] D. Faggioli, G. Lipari, and T. Cucinotta, The Multiprocessor Bandwidth Inheritance Protocol, in *ECRTS '10: Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pp. 90–99, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-0-7695-4111-2.

[79] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, Task Automata: Schedulability, Decidability and Undecidability, *Information and Computation*, vol. 205 (8): pp. 1149–1172, 2007, ISSN 0890-5401.

[80] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, Schedulability Analysis Using Two Clocks, in *TACAS '03: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science*, vol. 2619, pp. 224–239, Springer-Verlag, 2003.

[81] E. Fersman, P. Pettersson, and W. Yi, Timed Automata with Asynchronous Processes: Schedulability and Decidability, in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 67–82, Springer-Verlag, London, UK, 2002, ISBN 3-540-43419-4.

[82] N. Fisher and S. Baruah, The Global Feasibility and Schedulability of General Task Models on Multiprocessor Platforms, in *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pp. 51–60, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2914-3.

[83] N. Fisher, M. Bertogna, and S. Baruah, The Design of an EDF-Scheduled Resource-Sharing Open Environment, in *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pp. 83–92, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-3062-1.

[84] N. Fisher and F. Dewan, Approximate Bandwidth Allocation for Compositional Real-Time Systems, in *ECRTS '09: Proceedings of the 21st Euromicro Conference on Real-Time Systems*, vol. 0, pp. 87–96, IEEE Computer Society, Los Alamitos, CA, USA, 2009, ISBN 978-0-7695-3724-5.

[85] N. Fisher, J. Goossens, and S. Baruah, Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible, *Real-Time Systems Journal*, vol. 45: pp. 26–71, June 2010, ISSN 0922-6443.

[86] S. Funk, J. Goossens, and S. Baruah, On-Line Scheduling on Uniform Multiprocessors, in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pp. 183–192, IEEE Computer Society, Washington, DC, USA, 2001, ISBN 0-7695-1420-0.

[87] 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 *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pp. 73–83, IEEE Computer Society, Washington, DC, USA, 2001, ISBN 0-7695-1420-0.

[88] P. Gai, M. D. 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 *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 189–198, IEEE Computer Society, Washington, DC, USA, 2003, ISBN 0-7695-1956-3.

[89] G. Gardey, D. Lime, M. Magnin, and O. H. Roux, Romeo: A Tool for Analyzing Time Petri Nets, in *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, vol. 3576, pp. 418–423, Springer-Verlag, July 2005.

[90] J. Goossens and C. Macq, Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation, in *Proceedings of the 9th Conference on Real-Time Systems*, pp. 133–148, Paris France, March 2001, ISBN 2-87717-078-0.

[91] R. Graham, *Computer and Job Shop Scheduling Theory*, chap. Bounds on the Performance of Scheduling Algorithms, pp. 165–227, John Wiley and Sons, Chichester, 1976.

[92] U. R.-T. Group, $LITMUS^{RT}$ homepage, http://www.cs.unc.edu/ anderson/litmus-rt/.

[93] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking, in *SEUS'07: Proceedings of the 5th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, vol. 4761, pp. 263–272, Springer Berlin/Heidelberg, September 2007.

[94] N. Guan, M. Stigge, W. Yi, and G. Yu, New Response Time Bounds for Fixed Priority Multiprocessor Scheduling, in *RTSS '09: Proceedings of the 30th IEEE Real-Time Systems Symposium*, pp. 387–397, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3875-4.

[95] M. G. Harbour, Architecture and Contract Model for Processors and Networks, Technical Report D-AC1, Universidad de Cantabria, 2006.

[96] F. Heitmann and D. Moldt, Petri Nets Tools Database Quick Overview, online: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html, January 2010.

[97] J. Held and S. Koehl, Introducing the Singlechip Cloud Computer: Exploring the Future of Many-core Processors, White Paper, Intel Labs, Tera-scale Computing Research, May 2010.

[98] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, HYTECH: A Model Checker for Hybrid Systems, in *CAV '97: Proceedings of the 9th International Conference on*

Computer Aided Verification, pp. 460–463, Springer-Verlag, London, UK, 1997, ISBN 3-540-63166-6.

[99] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, What's Decidable about Hybrid Automata?, in *STOC '95: Proceedings of the 27th annual ACM Symposium on Theory of Computing*, pp. 373–382, ACM, New York, NY, USA, 1995, ISBN 0-89791-718-9.

[100] K. S. Hong and J. Y.-T. Leung, On-Line Scheduling of Real-Time Tasks, *IEEE Transactions on Computers*, vol. 41 (10): pp. 1326–1331, 1992, ISSN 0018-9340.

[101] W. A. Horn, Some Simple Scheduling Algorithms, *Naval Research Logistics Quarterly*, vol. 21 (1): pp. 177–185, 1974.

[102] M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani, Schedulability of Asynchronous Real-Time Concurrent Objects, *Journal of Logic and Algebraic Programming*, vol. 78 (5): pp. 402–416, 2009.

[103] L. Ju, A. Roychoudhury, and S. Chakraborty, Schedulability Analysis of MSC-based System Models, in *RTAS '08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 215–224, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3146-5.

[104] R. Kaiser, Combining Partitioning and Virtualization for Safety-Critical Systems, White Paper, SYSGO AG, 2007.

[105] S. Kato and N. Yamasaki, Semi-partitioned Fixed-Priority Scheduling on Multiprocessors, in *RTAS '09: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 23–32, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3636-1.

[106] X. Ke, K. Sierszecki, and C. Angelov, COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems, in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 199–208, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2975-5.

[107] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine, Decidable Integration Graphs, *Information and Computation*, vol. 150 (2): pp. 209–243, 1999, ISSN 0890-5401.

[108] P. Krčál, M. Stigge, and W. Yi, Multi-processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times, in *FORMATS '07: Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, vol. 4763, pp. 274–289, Springer Berlin/Heidelberg, October 2007.

[109] P. Krčál and W. Yi, Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata, in *TACAS '04: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science*, vol. 2988, pp. 236–250, Springer, Berlin, 2004.

[110] H.-H. Kwak, I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky, Symbolic Schedulability Analysis of Real-Time Systems, in *RTSS '98: Proceedings of the 19th Real-Time Systems Symposium*, pp. 409–418, IEEE Computer Society, Washington, DC, USA, 1998, ISBN 0-8186-9212-X.

[111] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors, in *ECRTS '09: Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pp. 239–248, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3724-5.

[112] K. G. Larsen, P. Pettersson, and W. Yi, UPPAAL in a Nutshell, *International Journal on Software Tools for Technology Transfer*, vol. 2 (1): pp. 134–152, 1997.

[113] S. Lauzac, R. Melhem, and D. Mossé, Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor, *ECRTS '98: Proceedings of the 10th Euromicro Conference on Real-Time Systems*, pp. 188–195, 1998, ISSN 1068-3070.

[114] I. Lee, A. Philippou, and O. Sokolsky, Resources in Process Algebra, *Journal of Logic and Algebraic Programming*, vol. 72 (1): pp. 98–122, May 2007.

[115] S. K. Lee, On-line multiprocessor scheduling algorithms for real-time tasks, in *TENCON '94. Proceedings of the IEEE Region 10's Ninth Annual International Conference.*, vol. 2, pp. 607–611, 1994, ISBN 0-7803-1862-5.

[116] H. Leontyev and J. H. Anderson, A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees, in *ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 191–200, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3298-1.

[117] J. Y. T. Leung and J. Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, *Performance Evaluation*, vol. 2: pp. 237–250, 1982.

[118] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling, in *ECRTS '10: Proceedings of the of the 22nd Euromicro Conference on Real-Time Systems*, pp. 3–13, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 78-1-4244-7546-9, ISSN 1068-3070.

[119] D. Lime and O. H. Roux, Expressiveness and Analysis of Scheduling Extended Time PEtri Nets, in *FET '03: Proceedings of the 5th IFAC conference on fieldbus and their applications*, Elsevier Science, July 2003.

[120] D. Lime and O. H. Roux, Formal Verification of Real-Time Systems with Preemptive Scheduling, *Real-Time Systems*, vol. 41 (2): pp. 118–151, 2009, ISSN 0922-6443.

[121] G. Lipari, G. Lamastra, and L. Abeni, Task Synchronization in Reservation-Based Real-Time Systems, *IEEE Transactions on Computers*, vol. 53 (12): pp. 1591–1601, 2004, ISSN 0018-9340.

[122] C. L. Liu, Scheduling Algorithms for Hard Real-Time Multiprogramming of a Single Processor, *JPL Space Programs Summary*, vol. 2 (28–31): pp. 37–60, November 1969.

[123] C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM*, vol. 20 (1): pp. 46–61, 1973, ISSN 0004-5411.

[124] 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*, vol. 28 (1): pp. 39–68, 2004, ISSN 0922-6443.

[125] G. Macariu, A Model Checking Approach for Multi-core Time Partitions Design, in *ICESS '10: Proceedings of the 7th IEEE International Conference on Embedded Software and Systems*, pp. 1910–1917, IEEE Computer Society, Los Alamitos, CA, USA, June 2010, ISBN 978-0-7695-4108-2.

[126] G. Macariu and V. Cretu, Model-based Analysis of Contract-based Real-Time Scheduling, in *SEUS '09: Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, *Lecture Notes in Computer Science*, vol. 5860, pp. 227–239, Springer Berlin/Heidelberg, November 2009, ISBN 978-3-642-10264-6, ISSN 0302-9743 (Print) 1611-3349 (Online).

[127] G. Macariu and V. Cretu, Timed Automata Model for Component-based Real-Time Systems, in *ECBS '10: Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pp. 121–130, IEEE Computer Society, March 2010, ISBN 978-1-4244-6537-8.

[128] G. Macariu and V. Cretu, Enabling Parallelism and Resource Sharing in Multi-core Component-based Systems, in *ISORC '11: Proceedings of the 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, IEEE Computer Society, March 2011, in print.

[129] G. Madl and S. Abdelwahed, Model-based analysis of distributed real-time embedded system composition, in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pp. 371–374, ACM, New York, NY, USA, 2005, ISBN 1-59593-091-4.

[130] G. Madl, N. Dutt, and S. Abdelwahed, A Conservative Approximation Method for the Verification of Preemptive Scheduling Using Timed Automata, in *RTAS '09: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 255–264, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3636-1.

[131] A. Masrur, S. Chakraborty, and G. Färber, Constant-time admission control for deadline monotonic tasks, in *DATE '10: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 220–225, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2010, ISBN 978-3-9810801-6-2.

[132] E. Massa and G. Lima, A Bandwidth Reservation Strategy for Multiprocessor Real-Time Scheduling, in *RTAS '10: Proceedings of the 16th Real-Time and Embedded Technology and Applications Symposium*, pp. 175–183, IEEE Computer Society, Washington, DC, USA, 2010, ISBN 978-1-4244-6690-0, ISSN 1080-1812.

[133] C. W. Mercer, S. Savage, and H. Tokuda, Processor Capacity Reserves for Multimedia Operating Systems, Tech. rep., Pittsburgh, PA, USA, 1993.

[134] P. Merlin and D. Farber, Recoverability of Communication Protocols - Implications of a Theoretical Study, *IEEE Transactions on Communications*, vol. 24 (9): pp. 1036–1043, September 1976, ISSN 0090-6778.

[135] A. K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*, Phd thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[136] A. K. Mok, X. A. Feng, and D. Chen, Resource Partition for Real-Time Systems, in *RTAS '01: Proceedings of the 7th Real-Time Technology and Applications Symposium*, pp. 75–84, IEEE Computer Society, Washington, DC, USA, 2001.

[137] F. Nemati, M. Behnam, and T. Nolte, An Investigation of Synchronization under Multiprocessors Hierarchical Scheduling, in *ECRTS '09: Proceedings of the Work-In-Progress (WIP) session of the 21st Euromicro Conference on Real-Time Systems*, pp. 49–52, July 2009.

[138] F. Nemati, M. Behnam, and T. Nolte, Multiprocessor Synchronization and Hierarchical Scheduling, in *ICPPW '09: Proceedings of the International Conference on Parallel Processing Workshops*, pp. 58–64, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-0-7695-3803-7.

[139] F. Nemati, M. Behnam, T. Nolte, and R. J. Bril, Investigation of implementing a synchronization protocol under multiprocessors hierarchical scheduling, in *ETFA '09: Proceedings of the 14th IEEE international conference on Emerging technologies & factory automation*, pp. 1670–1673, IEEE Press, Piscataway, NJ, USA, 2009, ISBN 978-1-4244-2727-7.

[140] J. E. Nielsen and K. S. Knudsen, *MoVES - A Tool for Modelling and Verification of Embedded Systems*, Master thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2007, URL http://www2.imm.dtu.dk/pubdb/p.php?5267.

[141] C. A. Phillips, C. Stein, E. Torng, and J. Wein, Optimal Time-Critical Scheduling via Resource Augmentation (extended abstract), in *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 140–149, ACM, New York, NY, USA, 1997, ISBN 0-89791-888-6.

[142] A. Rahni, E. Grolleau, and M. Richard, An efficient response-time analysis for real-time transactions with fixed priority assignment, *Innovations in Systems and Software Engineering*, vol. 5: pp. 197–209, 2009, ISSN 1614-5046, URL http://dx.doi.org/10.1007/s11334-009-0095-2, 10.1007/s11334-009-0095-2.

[143] R. Rajkumar, Real-time Synchronization Protocols for Shared Memory Multiprocessors, in *ICDCS'90: Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 116–123, May–June 1990.

[144] K. Ramamritham and J. A. Stankovic, Scheduling Algorithms and Operating Systems Support for Real-Time Systems, *Proceedings of the IEEE*, vol. 82 (1): pp. 55–67, January 1994, ISSN 0018-9219.

[145] C. Ramchandani, *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, Phd thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[146] O. H. Roux and A.-M. Déplanche, A T-time Petri Net Extension for Real-Time Task Scheduling Modeling, *European Journal of Automation (JESA)*, vol. 36 (7): pp. 973–987, 2002.

[147] O. H. Roux and D. Lime, Time Petri Nets with Inhibitor Hyperarcs. Formal Semantics and State Space Computation, in *ICATPN '04: Proceedings of the International Conference on Applications and Theory of Petri Nets*, *Lecture Notes in Computer Science*, vol. 3099, pp. 371–390, Springer-Verlag, June 2004.

[148] C. Seceleanu, P. Pettersson, and H. Hansson, Scheduling Timed Modules for Correct Resource Sharing, in *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pp. 102–111, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3127-4.

[149] S. Sentilles, A. Pettersson, D. Nystrom, T. Nolte, P. Pettersson, and I. Crnkovic, Save-IDE - A Tool for Design, Analysis and Implementation of Component-based Embedded Systems, in *ICSE '09: Proceedings of the IEEE 31st International Conference on Software Engineering*, pp. 607–610, IEEE Computer Society, Washington, DC, USA, 2009, ISBN 978-1-4244-3453-4.

[150] L. Sha, T. Abdelzaher, K. E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok, Real Time Scheduling Theory: A Historical Perspective, *Real Time Systems*, vol. 28: pp. 101–155, 2004.

[151] L. Sha, R. Rajkumar, and J. P. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers*, vol. 39 (9): pp. 1175–1185, 1990, ISSN 0018-9340.

[152] I. Shin, A. Easwaran, and I. Lee, Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors, in *ECRTS'08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 181–190, IEEE Computer Society, Washington, DC, USA, 2008, ISBN 978-0-7695-3298-1.

[153] I. Shin and I. Lee, Periodic Resource Model for Compositional Real-Time Guarantees, in *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pp. 2–13, IEEE Computer Society, Washington, DC, USA, 2003, ISBN 0-7695-2044-8.

[154] B. Sprunt, *Aperiodic task scheduling for real-time systems*, Ph.D. thesis, Pittsburgh, PA, USA, 1990.

[155] A. Srinivasan and S. Baruah, Deadline-based scheduling of periodic task systems on multiprocessors, *Information Processing Letters*, vol. 84: pp. 93–98, October 2002, ISSN 0020-0190.

[156] J. K. Strosnider, J. P. Lehoczky, and L. Sha, The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, *IEEE Transactions on Computers*, vol. 44 (1): pp. 73–91, 1995, ISSN 0018-9340.

[157] D. Tudor, G. Macariu, C. Jebelean, and V. Cretu, Load Balancing in Modern Multi-Core Mobile Communication Systems, in *CSCS 17: Proceedings of 17th International Conference on Control Systems and Computer Science*, vol. 1, pp. 255–261, Bucharest, Romania, May 2009, ISSN 2066-4451.

[158] D. Tudor, G. Macariu, C. Jebelean, and V. Cretu, Towards a load balancer architecture for multi-core mobile communication systems, in *SACI '09: Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics*, pp. 391–396, May 2009, ISBN 978-1-4244-4477-9.

[159] E. Vicario, Static Analysis and Dynamic Steering of Time-Dependent Systems, *IEEE Transactions on Software Engineering*, vol. 27: pp. 728–748, 2001, ISSN 0098-5589.

[160] A. J. Wellings, Y. Chang, and T. Richardson, Enhancing the platform independence of the real-time specification for Java, in *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pp. 61–69, ACM, New York, NY, USA, 2009, ISBN 978-1-60558-732-5.

[161] S. Yovine, Kronos: A Verification Tool for Real-Time Systems, *International Journal of Software Tools for Technology Transfer*, vol. 1 (1/2): pp. 123–133, October 1997.

[162] D. Zhang and R. Cleaveland, Fast On-the-Fly Parametric Real-Time Model Checking, in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pp. 157–166, IEEE Computer Society, Washington, DC, USA, 2005, ISBN 0-7695-2490-7.

[163] A. Zuhily and A. Burns, Exact Response Time Scheduling Analysis of Accumulatively Monotonic Multiframe Real Time Tasks, in *ICTAC '08: Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pp. 410–424, Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 978-3-540-85761-7.