Scheduling and Synchronization for Multi-core Real-time Systems

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Karthik S. Lakshmanan

Thesis Committee:
Advisor: Prof. Ragunathan Rajkumar
Prof. John Lehoczky
Prof. Daniel Mossé
Prof. Onur Mutlu

Carnegie Mellon University
Pittsburgh, PA

September, 2011

Copyright © 2011 Karthik S. Lakshmanan

Keywords: Real-Time Systems, Scheduling, Multi-core Computing, Synchronization, Parallel Processing, Operating Systems, Mixed-Criticality Systems, Performance Analysis

To my dad and mom,
for holding me close to their heart,
yet setting me free to follow my own

Abstract

Multi-core processors are already prevalent in general-purpose computing systems with manufacturers currently offering up to a dozen cores per processor. Real-time and embedded systems adopting such processors gain increased computational capacity, improved parallelism, and higher performance per watt. However, using multi-core processors in real-time applications also introduces new challenges and opportunities for efficient scheduling and task synchronization. In this dissertation, we study this problem, characterize the design space, and develop an analytical and systems framework for multi-core real-time scheduling.

Exploiting the co-located nature of processor cores, the general principle adopted in this thesis is to statically partition tasks among processor cores, co-allocate synchronizing tasks when possible, and introduce limited inter-core task migration and synchronization for improving system utilization as necessary. We model the multi-core real-time scheduling problem as a bin-packing problem and develop an object splitting algorithm for scheduling tasks on multi-core processors. We develop Highest-Priority Task Splitting (HPTS) to schedule *independent sequential* tasks on multi-core processors. We then analyze the overheads of inter-core task synchronization and provide mechanisms to efficiently allocate *synchronizing sequential* tasks on multi-cores by co-locating such tasks. We then generalize this approach to provide early solutions for scheduling *parallel* real-time tasks using the fork-join model. Next, we develop mechanisms to use such techniques in *mixed-criticality* systems. Finally, we describe the distributed resource kernel framework, where we demonstrate the practical feasibility of our approach.

The results of this dissertation contribute to a system that can *efficiently* utilize multi-core processors to *predictably* execute periodic tasks with well-defined deadlines and *analytically* guarantee such deadlines. We provide a utilization bound of 65% for independent sequential tasks, demonstrate up to 50% reduction in the required number of cores using synchronization-aware allocation, and prove a 3.42 resource augmentation bound for parallel real-time task scheduling.

Acknowledgements

In India, since the Vedic ages, schools are referred to as *Gurukuls*, where students are considered as the *extended family* (kula) of their *teachers* (gurus). I am humbled to have Raj as my *guru*, and be a part of his academic family. His passion, dedication, and energy never cease to amaze me, and I hope that I gained a fraction of those over the course of my graduate studies. I am deeply indebted to Raj for making my stay at CMU as comfortable as it could have *ever* been.

I would like to thank my thesis committee members, Prof. John Lehoczky, Prof. Daniel Mossé, and Prof. Onur Mutlu for their time and effort in helping me bring this dissertation to life. I feel privileged to have worked with Prof. Lehoczky, and I will always look up to his precision and clarity of thought. I have enjoyed the lively conversations that I have had with Prof. Mossé, and his view of the big picture and academic rigor. I admire Prof. Mutlu for his work on multi-core architectures and his perspectives as a computer architect on this dissertation.

If there is one person, who has made me more productive and pushed my limits, then that would be Dionisio de Niz. I would like to thank Dio for our early sessions at the Software Engineering Institute (SEI). We have shared lots of coffee, interesting discussions, and some very good work. I also like to thank his wife Chelo for packing some awesome breakfast.

My decision to work towards a Ph.D. stemmed from the respect I have for my Undergraduate thesis advisor Prof. Ranjani Parthasarathi. I like to thank Prof. Parthasarathi for her inspiration. I would also like to thank Prof. A.P.Shanthi for my first exposure to research.

I have spent almost the entirety of my past five years at the Real-Time and Multimedia systems Laboratory (RTML), and it has been *home* thanks to Gaurav Bhatia, Anthony Rowe, Rahul Mangharam, Arvind Kandhalu Raghu, Junsung Kim, Vikram Gupta, Maxim Buevich, Shinpei Kato, Haifeng Zhu, Yong-Hoon Choi, and Reza Azimi. I could not have asked for for better company, be it for coffee or late-night bug fixes. Gaurav always has the answers for my questions, be it for work or *life*. I really cherish the memories of my Half-Marathon training with Anthony, and hope that we get to train together for a race sometime. Rahul has been a great mentor and colleague from the very early stages of my Ph.D. I have found great friends in Arvind, Junsung, and Vikram. I like to thank them for keeping me company, be it 3 in the morning or 3 in the afternoon.

Shinpei has been a wonderful colleague, and I will always think of him when I need *focus* and *determination*. Lydia Corrado, Jennifer Engleson, Tonya Bordonaro, and Elaine Lawrence have always been there for me to take care of my issues and help me focus on my work.

Vishal Mhatre, Rashmi Sachdeva, Aman, Mihir, and Aditya Shah worked with me during their Masters degree. I wish them all the very best and thank them for their support. I am happy to have had a chance to work with Ricardo Marau and Prof. Luis Almeida.

A special thanks to my mentors at the SEI for their support and interesting discussions. I am grateful to Mark Klein, Jeff Hansen, Gabriel Moreno, and Sagar Chaki for providing feedback on my work.

I have received generous support from the industry during both during my graduate studies and my summer internships. I like to thank General Motors (GM) for their continued support and mentorship. Paolo Giusto, Massimo Osella, and Dr. Nady Boules at GM for helping me drive this thesis forward. Paolo has been extremely helpful in organizing many of my ideas and finding practical applications within GM to motivate my work. At Texas Instruments (TI), I like to thank Xiaolin Lu, Se-Joong Lee, Don Shaver, and Soon-Hyeok Choi for two wonderful summers of great practical experience in applying multi-core real-time scheduling algorithms on real systems. At Microsoft, eXtreme Computing Group, I like to thank Ajith Jayamohan, Alexey Pakhunov, and Suyash Sinha for hosting me for a great summer, and enabling me to experience the joy of working on a full-blown 48-core state-of-the-art processor.

My stay in Pittsburgh would be largely routine had it not been for my friends Arjun Rathi Vijayakumar, Siva Paramesh, Saketh Chemuru Muni, Prashanth Jampani, Rajesh, Vamshi Ambati, and Pooja, who have breathed life into my stay. They have shared many of my laughs and wonderful memories over the past few years. I thank Rajan and Keya Pandia for making my summers in Dallas eventful, and as always, my undergraduate friends, Madhan and Prasad for keeping me great company in Seattle.

Finally, I would like to thank my wonderful family. My dad Mr. Lakshmanan is a great inspiration for me and will always be the source of my courage. My mom Mrs. Vijaya has been as caring and concerned as one could ever hope for. My love goes to my little sister Kavitha, for all her support and encouragement. My wife Alagu, for her love and understanding, and agreeing to keep me company from India, while I finish writing my dissertation. Last but not the least, my uncle Mr. Gunasekar, for supporting me with my application to graduate school, helping me settle down in Pittsburgh, and always providing me with valuable advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor manufacturers have widely adopted multi-core technologies to effectively utilize continuously increasing transistor densities. Multi-core technology is considered as a practical alternative to increasing the processor clock frequency, which is limited by available instruction-level parallelism and leads to challenging power/thermal requirements [52]. Commercial vendors such as Intel [58], AMD [1], and FreeScale [49], already offer various processor solutions with multiple cores on the same package. Processors with up to a dozen cores per package are already employed in production systems [1], while research prototypes with 80 cores-per-chip have been successfully developed by Intel [124]. Tilera has also demonstrated a 100-core chip with shared caches, interconnects, and memory controllers [119]. Researchers have even predicted that chips with hundreds of processing cores on the same package could become available in the future [28].

Given the proliferation of multi-core processors in general-purpose computing, embedded and real-time applications such as automotive, avionics, and smart phones are actively considering the use of such processors. These application domains would benefit from the additional computational capacity made available at a lower power requirement. However, effectively utilizing multi-core processors in traditional real-time applications requires new tools and techniques for programming, scheduling, synchronization, certification, and runtime support. In this dissertation, we address the scheduling and synchronization challenges arising in the context of multi-core real-time systems. A more detailed description of the scope is available in Section 1.3.

The main motivation for this dissertation arises from the heavy computational requirements of many hard real-time applications such as automotive engine control, avionics flight control, and driver assistance. These systems often execute control loops and other time-critical tasks, which require absolute guarantees on meeting deadlines. The heavy computational demand motivates the need for employing multi-core processors, which could enable additional functionality such as smarter algorithms for fuel injection, smoother control, and fully autonomous operation. The increasing portfolio of multi-core processors also ensures long-term hardware support for such applications, which acts as a forcing function for systems with long expected lifetimes.

In the real-time systems literature, there has been significant research on scheduling real-time tasks on Symmetric Multi-Processors (SMPs). Although multi-core processors largely resemble SMPs, there are some key differences such as (i) the presence of fast interconnects between the processor cores due to their co-located nature, (ii) the potential availability of multiple levels of on-chip shared cache, and (iii) the shared nature of the off-chip pins connecting to memory and I/O devices. Traditional real-time multiprocessor scheduling algorithms are classified as either *partitioned*, where tasks are not allowed to migrate across processor cores, or *global*, where tasks are allowed to unrestrictedly migrate across processor cores. The architectural characteristics of multi-core processors make them more amenable to semi-partitioned scheduling [3], where a limited number of tasks are allowed to migrate across core boundaries. In this dissertation, we propose a semi-partitioning framework for scheduling *periodic* real-time tasks, where a coordinated approach is adopted for allocating tasks to processor cores, scheduling tasks within processor cores, and synchronizing with other tasks. The proposed approach limits inter-core task migrations for reducing scheduling overheads and reduces inter-core task synchronization for containing synchronization costs in real-time multi-core systems.

1.1 Thesis Statement

This dissertation studies the problem of *efficiently utilizing multi-core processors to predictably execute periodic tasks and analytically guarantee task deadlines*. To address this problem, we develop a comprehensive analytical framework and operating systems infrastructure that supports efficient task allocation, fixed-priority scheduling, predictable task synchronization, effective task parallelization, and criticality-aware overload handling. The proposed approach leverages statically defined task migrations to achieve high utilization bounds for task allocation. Coordination among task allocation, scheduling, and synchronization is used to manage blocking delays and reduce the utilization loss resulting from inter-core task synchronization. Task transformations are employed to effectively schedule parallel real-time tasks with fork-join structures. Mixed-criticality scheduling algorithms are introduced to manage system overloads in a criticality-aware fashion. All these solutions are developed in the context of a Distributed Resource Kernel implementation, which provides appropriate abstractions and primitives for distributed resource management. This demonstrates the practical feasibility of our proposed approach and serves as a basis for measuring implementation overheads.

1.2 Thesis Organization

The rest of this dissertation is organized as follows:

- A detailed summary of the relevant background material and related work is provided in Chapter 2. In that chapter, we discuss the existing literature in multi-processor and multi-core scheduling, describe the specific approaches adopted in this dissertation, and discuss the advances brought about by this work compared to the existing state-of-the-art.
- In Chapter 3, we consider the problem of scheduling independent periodic real-time *sequential tasks* on multi-core processors. We provide a solution that employs off-line task allocation and statically defined migrations. The proposed algorithm is analyzed for its

worst-case utilization bounds and average-case schedulable utilization.

- In Chapter 4, we relax the assumption of *independent* tasks and study the problem of sequential task synchronization in multi-core systems. We analytically bound the overheads of inter-core task synchronization and then develop a coordinated approach to task allocation, scheduling, and synchronization, which leverages the multiprocessor priority ceiling protocol to provide bounded blocking delays and avoids inter-core task synchronization when possible. We also quantitatively evaluate the proposed algorithm for its utilization benefits.
- Chapter 5 relaxes the assumption of *sequential* tasks and considers the problem of scheduling periodic parallel real-time tasks. In this dissertation, we restrict our attention to independent parallel real-time tasks in a single-criticality domain, and other generalizations are considered as future work. We first discuss the *fork-join* task model and propose a task transformation based approach to scheduling such tasks on multi-core processors. This transformation is shown to efficiently convert the parallel task scheduling problem into the previously studied sequential scheduling problem. We also provide analytical results for bounding the competitiveness of this contribution with respect to the ideal scheduling algorithm for fork-join task sets.
- Next in Chapter 6, we propose a *Mixed-Criticality* task allocation and develop multi-core scheduling algorithms for handling overload scenarios with multiple criticality levels. We illustrate this approach using a real-world case-study of a radar surveillance application, where tasks have different criticality levels. In this chapter, we also extend existing task synchronization protocols with bounded blocking delays to include the proposed mixed-criticality scheduling algorithms.
- Finally, in Chapter 7, we describe a comprehensive practical framework for task allocation, scheduling, synchronization, parallelization, and overload provisioning in distributed multi-core processors. This framework is realized by developing the proposed solutions in

the context of a Distributed Resource Kernel implementation for systems with potentially multiple multi-core processors. The implementation describes our task splitting approach, synchronization library, and parallel real-time task scheduling considerations. Using our implementation, we have evaluated the task splitting approach on an experimental platform for measuring its system-level overheads. We have also implemented our synchronization protocols as a part of a userspace library called RT-MAP, which we use to quantify the implementation costs of inter-core synchronization.

- Chapter 8 provides the concluding remarks and summarizes the major contributions of this dissertation. Possible avenues of future work and extensions to this dissertation are also provided. Specifically, we discuss problems in the domains of heterogeneous multi-core processors, energy management, Graphics Processing Unit (GPU) scheduling, and servicing Soft Real-Time and Aperiodic real-time tasks.

1.3 Dissertation Scope

In this dissertation, we focus on systems with only the following characteristics:

1. *Homogeneous Uniform Multi-core Processors*, where each processor core has the same micro-architecture and operating speed as any other in the same package. This effectively implies that the processor cores are interchangeable from a functional perspective.
2. *Periodic Task Sets*, where each task is an infinite sequence of jobs released exactly at periodic intervals. Algorithms in this dissertation can are also be applied for Sporadic task sets, often with no modifications. However, for the simplicity of presentation and discussion, we restrict our scope to strictly periodic task sets. We do *not* study *servers* to handle aperiodic tasks in multi-core processors, which constitutes key future work.
3. *Hard Real-Time Systems*, where a job missing its deadline constitutes a failure of the corresponding task. There is no accrued value for jobs completing late or occasionally missing

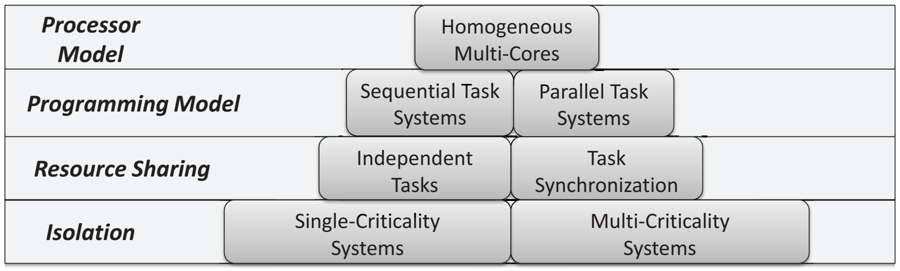


Figure 1.1: Dissertation Scope

their deadlines. However, in mixed-criticality setups, we do consider that the deadlines of higher critical tasks are more important than those of lower critical tasks.

4. *Implicit Deadlines*, where task periods are equal to their deadlines. Although algorithms presented in this dissertation can be extended to *constrained-deadline* task sets with deadlines shorter than periods, we do not consider such task sets in the scope of this dissertation.

1.3.1 Design Space

Real-time scheduling on multiprocessor systems is a well-studied problem in the literature. The scheduling algorithms developed for this problem are classified as partitioned (static binding) and global (dynamic binding) approaches, with each category having its own merits and de-merits. Multi-core processors with co-located processor cores and shared levels of cache are amenable to a new class of scheduling algorithms called semi-partitioned scheduling, where a limited subset of tasks are allowed to migrate across cores to improve the overall system utilization. In this dissertation, we focus on the *semi-partitioned approach*, and also assume that uniprocessor *fixed-priority preemptive scheduling* schemes are used. We perform task splitting where a task can be split to run across more than one processor[1]. This allows us to overcome the 50% bound imposed by the underlying bin-packing problem of allocating tasks to processors. For instance, in a hypothetical system where all tasks have 51% utilization, under traditional bin packing, each

[1]In the rest of this dissertation, we will use the terms *processor*, *core* and *processor core* interchangeably.

processor core will have 49% unused utilization. Task splitting takes advantage of the co-located nature of the processing cores to increase the overall system utilization. Our algorithms presented in this dissertation do not split more than one task per processor, and therefore minimize any penalties of task splitting.

Tasks in the system could synchronize with each other at multiple points. In this context, we do assume that all the *critical sections are non-nested* [2]. As we show later, we will use synchronization-aware allocation to co-locate tasks accessing shared resources, thereby reducing the scheduling penalties associated with global task synchronization. Adopting the spirit of semi-partitioning, we *separate* synchronizing tasks on different processor cores when necessary to improve overall system utilization.

From the perspective of programming models, we will study both *Sequential* and *Parallel* tasks. Sequential programming models proved to be quite useful when processor manufacturers pushed for faster and faster processor clock speeds. As the semiconductor vendors shift the scaling trends towards more and more processor cores, the benefits of sequential programming start to diminish in comparison to the inability to take advantage of the available parallelism. Parallel programming models such as *OpenMP* [92] are promising candidates for taking advantage of future massive multi-core processors. These models have the capability to parallelize specific segments of tasks, thereby leading to shorter response times when possible.

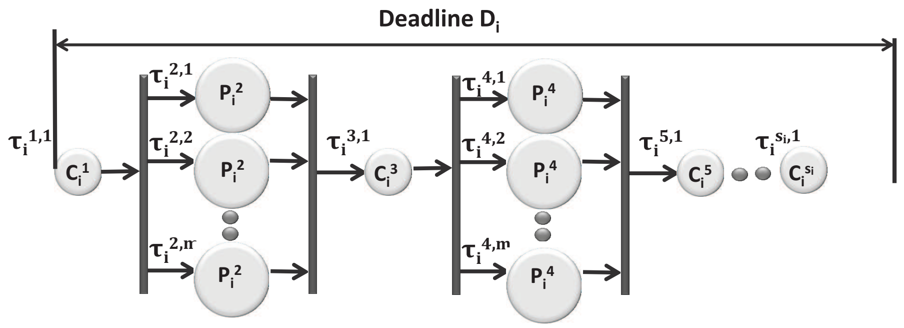


Figure 1.2: Fork-Join Task Model.

[2]Nested critical sections may be accommodated for example using aggregate locks on the outermost critical section [106].

In this dissertation, we focus on the *fork-join programming model* for parallel tasks. Fork-Join is a popular parallel programming paradigm employed in systems such as Java [68] and OpenMP. In this work, we study *basic* fork-join tasks as shown in Figure 1.2. Each basic fork-join task begins as a single master thread that executes sequentially until it encounters the first *fork* construct, where it splits into multiple parallel threads which execute the parallelizable part of the computation. After the parallel execution region, a *join* construct is used to synchronize and terminate the parallel threads, and resume the master execution thread. This structure of *fork* and *join* can be repeated multiple times within a job execution. A *general* fork-join task is one where the parallel execution regions themselves can be fork-join structures. A study of such nested fork-join structures is beyond the scope of this work, and presents a direction in which our results can be extended. In the context of this work, we also assume that the parallel execution regions themselves can be preempted individually on their respective processors. This assumption may not hold good in certain systems, where approaches such as gang scheduling [63] may be required. Henceforth, when we use the term *fork-join*, it denotes this *basic* fork-join task model.

We also restrict our attention to tasks with a *pre-specified static number of threads*, as dictated by the `OMP_NUM_THREADS` environment variable in OpenMP. Although the number of threads in a parallel region can be dynamically adjusted in specific implementations of OpenMP, we do not consider such a task model in this work. Analyzing such dynamic *fork-join* task structures forms an important aspect of future work.

In the later part of this dissertation, we generalize our approach to include mixed-criticality systems. In such systems, we consider tasks with different criticality levels being hosted on the same multi-core processor. In this dissertation, we restrict our focus to systems where high-criticality tasks are considered *absolutely* more important than low-criticality tasks. The scheduling guarantee that we look for in such systems is that any task can enter an overload situation as long as the tasks with higher criticality do not get affected. We illustrate this guarantee later with a radar surveillance case-study, where the benefits of this approach are more apparent.

1.4 System Model and Notation

The system is a collection of tasks $\tau : \{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is an infinitely recurring sequence of jobs, defined by the following properties:

- *Period* T_i: Representing the inter-arrival time between jobs of the task τ_i.
- *Deadline* D_i: Denoting the relative deadline for jobs of task τ_i i.e. Any job J of τ_i released at time t should complete by time $(t + D_i)$. Unless explicitly specified, we assume $D_i = T_i$ and *do not explicitly* represent D_i in the task parameters. If D_i is indeed specified then $D_i \leq T_i$ for the scope of this dissertation.
- *Criticality* κ_i: Signifying the semantic importance of the task τ_i. We will use lower numbers to denote higher criticality. We will also assume an absolute criticality value, where it is always more *important* for a task with higher criticality to meet its deadlines than a task with lower criticality. We *do not explicitly* represent the criticality of a task in single-criticality systems.
- *Parallelism* m_i: Providing the number of parallel threads spawned by the task τ_i in its parallel segments. For sequential tasks $m_i = 1$ and does not get represented explicitly.
- *Computation* $< \Psi_i^1, \Psi_i^2, ..., \Psi_i^{s_i} >$: A sequence of s_i *computational segments*, which alternate between sequential and parallel segments. Each *computational segment* in turn could comprise of multiple *execution blocks* alternating between normal and critical section execution.

Each individual segment Ψ_i^j is of the form $[C_{i,1}^j, C_{i,1}'^j, C_{i,2}^j, C_{i,2}'^j, ..., C_{i,n_{i,j}-1}'^j, C_{i,n_{i,j}}^j]$, where $n_{i,j}$ is the number of normal execution blocks of Ψ_i^j and $(n_{i,j} - 1)$ is the number of critical section blocks of Ψ_i^j.

when j is odd, $C_{i,k}^j$ denotes the worst-case execution time of the kth normal execution block of a *sequential* execution segment Ψ_i^j, while $C_{i,k}'^j$ denotes the worst-case execution time of the kth critical section block of a *sequential* execution segment Ψ_i^j.

when j is even, $C^j_{i,k}$ denotes the worst-case execution time of the kth normal execution block of a *parallel* execution segment Ψ^j_i, while $C'^j_{i,k}$ denotes the worst-case execution time of the kth critical section block of a *parallel* execution segment Ψ^j_i. We will also use the notation of $P^j_{i,k}$ for $C^j_{i,k}$, with even j, to explicitly represent the *parallel* nature of this execution segment.

Each parallel execution segment Ψ^j_i, where j is even, is a collection of m_i computation segments that can be executed in parallel.

1.4.1 Architectural Assumptions

Although our task model abstracts the tasks for providing analysis and timing guarantees, it relies heavily on task worst-case execution time parameters. Obtaining these parameters require specific architectural support, which bounds the pessimism in worst-case execution times. A few of the key design choices considered for embedded multi-core processors are listed below:

- *Cache partitioning or Scratch pad memory*: From the worst-case execution time analysis perspective, application developers would benefit from analyzing their tasks in isolation and measuring execution times. It would be ideal for the hardware architecture to faithfully maintain these execution times when executed with other tasks. Adopting cache partitioning or scratch pad memory is one decision to avoid interference from other cores.
- *Pinned pages and Memory Controller fairness*: Memory stalls could take orders of magnitude longer than cache accesses. Given that demand paging and swapping mechanisms add unacceptable delays, it would be ideal to pin pages to memory. Given that different threads from different cores could be issuing memory requests, it would be useful to have a notion of fairness and bounded service times at the memory controller level.
- *Deterministic Execution Pipelines*: Various mechanisms such as out-of-order execution, hardware multi-threading, and super-scalar processors already introduce non-determinism in the uni-core processor context. These issues transcend into multi-core embedded processors as well. Architectures with bounded interference would significantly reduce the

pessimism in worst-case execution time analysis.

1.5 Approach Overview

The approach adopted in this dissertation is to view the multi-core scheduling problem as a bin-packing problem. The resulting solutions take the form of tasks that are allocated to processors off-line with statically defined migration points (if any). Within each individual processor, we adopt fixed-priority preemptive scheduling with priority ceiling and inheritance protocols for efficient task synchronization.

The adopted bin-packing approach to multi-core scheduling requires us to strategically define *objects*, which can then be packed on to the bins i.e. *processor cores*. In this dissertation, we define the following *objects*:

- In the case of *independent sequential tasks*, we use the individual tasks themselves as *objects*.
- For *synchronizing sequential tasks*, we combine all the tasks that synchronize with each other to create a *composite* task, which constitutes a single *object*.
- Each *parallel fork-join task* is considered to be an *object*.
- In *mixed-criticality* systems, objects are the tasks themselves. However, depending on whether they are executing in the *normal* mode or *critical* (overloaded) mode, they could have different sizes.

The above definition of *objects* will be adopted along the course of this dissertation as we move from independent sequential tasks to synchronizing sequential tasks, then from sequential tasks to parallel real-time tasks, and finally to mixed-criticality systems.

This definition of *objects* leads to two major issues: (i) large objects could lead to fragmentation of unused utilization across processor cores, and (ii) some objects might be too large to fit in any single processor core. This warrants the need for *object splitting*, where objects can be split to fill up the bins as needed. From the perspective of the bin-packing problem, allowing such splitting leads to us to an optimal solution where all the bins can be fully utilized. However, the objects being split here are *tasks* or *composite tasks*, which result in penalties from a scheduling penalties. The strategy then is to define a smart approach to *object splitting*. In this dissertation, we adopt the following approach:

- In the case of individual tasks, we split the task having the highest priority on a processor, which has the shortest worst-case response time. This allows us to maximize the deadline for the remaining segment of the task that needs to be allocated elsewhere. When it is not possible to split the highest priority task in the processor, we need to update the deadline of the remaining subtask accordingly.
- For composite tasks, we split the task such that the resulting tasks have the minimum global synchronization cost, when deployed on different processor cores. This ensures that the tasks that *heavily* synchronize with each other get allocated to the same processor core.
- For parallel tasks, we split the task into a single *master thread* and multiple *auxiliary* threads. The master thread is chosen so as to fully utilize a processor, resulting in no fragmentation on that processor. The remaining *auxiliary* threads can simply be treated as tasks with deadlines shorter than periods.

Having defined the *objects* themselves and the approach to *object splitting*, we are now left with defining the actual bin-packing algorithm. The details of the solution are available in the following chapters. The general approach is to order tasks by decreasing order of size within each criticality level. The intuition is that *larger objects are harder to pack, hence it is better to pack them earlier*.

Across criticality levels, we follow a criticality order from higher criticality levels to lower criticality levels. In this case the desired behavior is that we ensure the schedulability of high-criticality tasks even under worst-case overload conditions. The low-criticality tasks which get allocated later in the packing may have to give up their cycles to high-criticality tasks that are hosted on the same processor, when the system encounters an overload.

1.6 Approach Rationale

The approach of static task partitioning with statically defined migration points and fixed-priority scheduling is motivated by the following key advantages:

- We focus on partitioned fixed-priority scheduling due to:

  1) The simplicity of the scheduler: which enables it to be implemented on low-end embedded devices like interrupt controllers and micro-controllers

  2) Support in commercial real-time operating systems (VxWorks, ThreadX ([46])), and

  3) Readily available interfaces in industrial standards like POSIX and AUTOSAR [12]

  Given these advantages, our assumption of the fixed-priority context is both relevant and useful. Some users also perceive the criticality of a task as its scheduling priority, and fixed-priority scheduling naturally accommodates such schemes. For a detailed discussion, see [71].

- Per-processor run-queues are still used to schedule tasks and thus retain the cache locality properties associated with partitioning. Recent results [24], suggest that existing partitioned scheduling analysis is still superior to global scheduling analysis, and the caching benefits are still in favor of partitioned scheduling for hard real-time platforms.

- Given that the results in [6] already state and achieve the 50% bound for utilization under partitioned scheduling, the next logical step in improving the utilization bound would be to consider statically defined migrations. Using our analytical results and experimental evaluation, we will quantify the overheads of such migrations allowing system designers to still have a handle on the migration costs. A fully global approach with dynamic migrations could lead to significant and pessimistic bounds on inter-core migration costs.

- We adopt the priority ceiling and inheritance approach [105] due to its practicality of implementation and caching benefits. Other approaches would either lead to additional priority inversion for high-priority tasks or result in significant implementation complexity. Our implementation of the multiprocessor priority ceiling protocol as a `pthreads` library demonstrates this advantage.

Chapter 2

Background

Work related to this dissertation falls in four categories: (i) multi-processor and multi-core scheduling, (ii) multi-processor synchronization, (iii) parallel task scheduling, and (iv) mixed-criticality systems. We will now discuss the related work in each of these domains and describe the differences with our proposed approach.

2.1 Multi-processor and Multi-core Scheduling

The design space of the existing literature on multi-processor real-time scheduling algorithms is shown in Fig. 2.1. Multiprocessor scheduling schemes are classified into global (*1*-queue *m*-server) and partitioned (*m*-queue *m*-server) systems. It has been shown that each of these categories has its own advantages and disadvantages [67]. Global scheduling schemes can better utilize the available processors, as best illustrated by PFair [23] and LLREF [35]. These schemes appear to be best-suited for applications with small working-set sizes. Although the last level of on-chip shared cache ultimately determines the caching behavior of an application, task migrations tend to generate significant additional cache traffic due to invalidations and cache-consistency protocols. Weak processor affinity and preemption overheads therefore need to be managed to fully exploit the benefits of global approaches [111]. On the other hand, partitioned

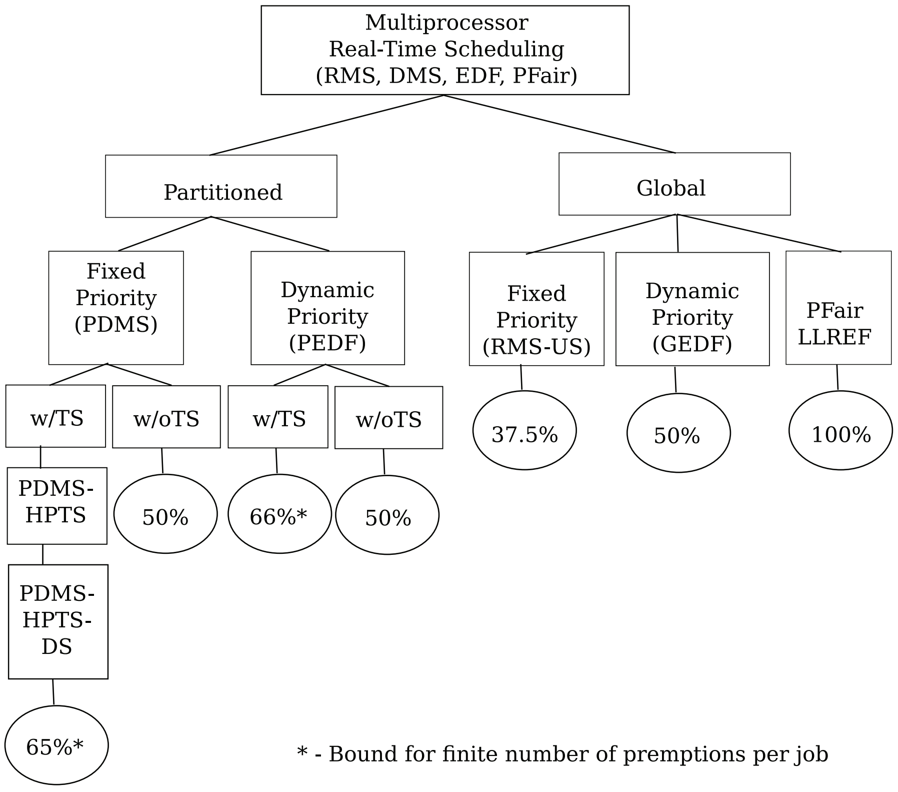


Figure 2.1: Design space of Multi-Processor Real-Time Scheduling

approaches are severely limited by the low utilization bounds associated with bin-packing problems. The advantage of these schemes is their stronger processor affinity, and hence they provide better average response times for tasks with larger working set sizes.

Global scheduling schemes based on rate-monotonic scheduling (RMS) and earliest deadline first (EDF) are known to suffer from the so-called Dhall effect [43]. When heavy-weight (high-utilization) tasks are mixed with lightweight (low-utilization) tasks, conventional real-time scheduling schemes can yield arbitrarily low utilization bounds on multi-processors. By dividing the task-set into heavy-weight and lightweight tasks, the RM-US [8] algorithm achieves a utilization bound of 33% for fixed-priority global scheduling. These results have been improved with a higher bound of 37.5% [80]. The global EDF scheduling schemes have been shown to possess a higher utilization bound of 50% [14]. PFair scheduling algorithms based on the notion of proportionate progress [17] can achieve the optimal utilization bound of 100%. Recent approaches [130] also reduce the number of migrations and preemptions incurred by fairness-based algorithms, further improving their overall performance. However, despite the superior performance of global schemes, significant research has also been devoted to partitioned schemes due to their appeal for a significant class of applications, and their scalability to massive multi-cores, while exploiting cache affinity.

Partitioned multiprocessor scheduling techniques have largely been restricted by the underlying bin-packing problem. The utilization bound of strictly partitioned scheduling schemes is known to be 50%. This optimal bound has been achieved for both fixed-priority algorithms [6] and dynamic-priority algorithms based on EDF [78]. Most modern multi-core processors provide some level of data sharing through shared levels of the memory hierarchy. Therefore, it could be useful to split a bounded number of tasks across processing cores to achieve a higher system utilization [3]. Partitioned dynamic-priority scheduling schemes with task splitting have been explored in this context [7, 61]. Fixed-priority scheduling with task-splitting support is relatively less analyzed in the literature. Recent results such as [55, 62] have provided task-splitting algorithms in the fixed-priority context. Optimal task-splitting algorithms have also been developed for tasks having the same period in [27]. In this dissertation, we propose the approach of highest-priority task splitting and demonstrate the benefits from a utilization bound perspective. Even though [55] has subsequently achieved a higher worst-case utilization bound, the average-case performance the proposed Highest-Priority Task Splitting algorithm is still better at around 88%.

In the area of real-time multi-core scheduling, there has also been previous work on cache-aware approaches to real-time scheduling [4]. We focus more on exploiting the shared caches to minimize the overhead of task splitting, rather than explicitly choosing cache-collaborative tasks to run in parallel. The partitioning algorithm may be modified to choose cache-collaborative tasks to be co-located on the same processing core. However, the effects of such partitioning schemes is the subject of future research.

2.2 Multi-processor Synchronization

Traditional real-time multiprocessor scheduling algorithms have dealt mostly with independent tasks having no interactions. de Niz and Rajkumar have previously developed a set of partitioning bin-packing algorithms to deploy groups of communicating tasks onto a network of processors [40]. The objective of these algorithms is to minimize the number of processors needed while trying to reduce the bandwidth required to satisfy the communication between these tasks. In this dissertation, we are concerned with tasks that need to synchronize on (potentially) globally shared resources, which could be executing on different processor cores.

Task synchronization in real-time systems is a well-known problem. Fixed-priority scheduling schemes employ techniques like priority inheritance and priority ceiling protocols to enable resource sharing across real-time tasks. Dynamic-priority scheduling schemes also use mechanisms like the Stack-based Resource Policy (SRP) [123] to handle real-time task synchronization. In the context of fixed-priority multiprocessor scheduling, the priority ceiling protocol has been extended to realize the multiprocessor priority ceiling protocol (MPCP) [105]. Synchronization schemes have also been developed for other related scheduling paradigms like PFair [94]. Multiprocessor extensions to SRP have also been considered and performance comparisons have been done with MPCP [93]. This dissertation adopts a more holistic approach to partitioned task scheduling by explicitly considering MPCP synchronization penalties during task allocation and investigating the impact of different Execution Control Policies (ECPs).

Recent studies [25, 26] have investigated the performance differences between spin-based and suspension-based synchronization protocols. In these studies, their authors found that spin-based protocols impose a smaller scheduling penalty than suspension-based ones, even under zero preemption costs. While these studies present interesting results, the analyses they used on suspension-based protocols can be substantially improved. In this dissertation, we have developed new schedulability analysis for these protocols that are less pessimistic and includes our improvements based on [85]. With this new analysis, we found that the suspension-based protocols in fact behave better than spin under low preemption costs (less than 160μs per preemption) and longer critical sections (15 μs) than those studied in [25].

2.3 Parallel Task Scheduling

Most of the results in classical multiprocessor real-time scheduling theory [34, 42, 43, 64, 72, 101] are focused on the sequential programming model, where the problem is to schedule many sequential real-time tasks on multiple processor cores. Parallel programming models introduce a new dimension to this problem, where jobs may be split into parallel execution segments at specific points. The parallel task scheduling problem is largely neglected or treated as scheduling multiple sequential tasks with predetermined offsets and precedence constraints. As computational requirements continue to increase, processor speeds are saturating and hardware architectures are shifting towards multi-cores, hence sequential programming is expected to yield diminishing results.

Scheduling of tasks with divisible loads was studied in [73], where Divisible Load Theory (DLT) was integrated with EDF and compared against a heuristic-based approach. The task model assumed arbitrarily divisible loads and the system model incorporated costs for computation and communication. Recent results [37, 63] have also considered alternative task models for parallel programming. In this work, we focus on the *fork-join* programming model [68] employed by *OpenMP* [92]. A Best-Fit scheduling heuristic with objective functions for Task Graphs (BFTG) was proposed in [2]. Although the authors also assumed fork-join task structures, the focus was on deriving an off-line schedule table as opposed to a priority-driven scheduling approach, which we propose to adopt.

Recent results [37, 63] have considered different task models for parallel programming. In this work, we focus on the *fork-join* programming model employed by the *OpenMP* system.

In this dissertation, we will focus on understanding the scheduling penalties imposed by the constrained deadline nature of parallel real-time tasks. We also propose developing task transformations to avoid these parallel structures when possible, such that the remaining tasks can be scheduled using standard priority-driven Deadline Monotonic Scheduling (DMS).

To the best of our knowledge, there is no serious prior research on the fork-join task model in the context of real-time systems. *OpenMP* is a mature system for parallel programming, and is expected to play a pivotal role in shaping future programming paradigms for multi-core processors.

2.4 Mixed-Criticality Systems

Mixed-criticality systems require that under overload conditions the high-criticality tasks are guaranteed to meet their deadlines, even at the expense of low-criticality tasks missing their deadlines. The related problem of off-line overload scheduling is well-studied in the real-time systems literature [32, 81]. Additional studies have also been done on online scheduling of overloads [22, 51]. These algorithms lay the foundational work in the area of dealing with task overloads, however, in the context of this dissertation we are more interested in criticality-aware servicing of such overloads.

Scheduling algorithms based on the elastic task model have also been proposed in [33], where tasks with higher elasticity are allowed to run at higher rates when required, whereas tasks with lesser elasticity are restricted to a uniform rate. In our case, the proposed algorithm deals with normal and overloaded execution requirements, and provides criticality-driven guarantees in terms of schedulability. The imprecise computational model is another approach to scheduling tasks that accrue additional utility when allowed to execute for longer [76].

Several important results in slack stealing have also been proposed in literature [102, 117, 118]. The key distinction of our results from existing results in overload scheduling and slack stealing is that we explicitly support the notion of task criticality, which is easier for system designers to deal with compared to derived attributes.

Chi-Sheng et al. present in [110] an approach to map the semantic importance of tasks in a task set into QoS service classes to improve the resource utilization. Their objective is to ensure that the allocated resources are never less for a high-criticality task than for a lower-criticality one. In our case, we focus on deadlines, and guarantee that whenever an overload reduces the resources available we ensure that a high-criticality task will not miss its deadline due to the allocation of more resources to a lower-criticality task.

Closely related to our proposal for dual-execution modes viz. normal mode (N) and critical mode (C) is the work on dual-priority scheduling [39]. Dual-priority scheduling is an effective technique for responsively scheduling soft tasks without compromising the deadlines of crucial hard real-time tasks. Rate-Based Earliest Deadline (RBED) [30] schedulers have also been developed to address this problem in dynamic-priority systems. In these results, while the notion of crucial (hard) tasks is somewhat similar to criticality there are two significant differences. First, non-crucial (soft and best-effort) tasks are not given any scheduling guarantees, whereas, in our scheme all criticality levels have a guarantee on execution times. Secondly, their limitation to two levels simplifies the analysis but also limits the utility. We allow an unlimited number of criticality levels with an implicit graceful degradation on overload in criticality order.

Another related approach to dual-priority scheduling is the earliest deadline zero laxity (EDZL) algorithm [36]. In EDZL, tasks are scheduled based on earliest deadline first (EDF) until some task reaches zero laxity (or zero slack), in which case its priority is elevated to the highest level. While the notion of zero slack is used in our solution, the existing EDZL results do not consider the notion of criticality and are not directly applicable to the mixed-criticality scheduling problem.

Period transformations offer a different solution for mixed-criticality systems [109, 125]. This technique has been subsequently extended and generalized in [18]. In these approaches, the disparity between overloaded computation time and normal computation time results in pessimistic assumptions. The proposed zero-slack scheduling algorithm switches tasks to their critical mode only when there would be zero slack remaining to meet their corresponding overloaded computation time requirements. Therefore, the zero-slack scheduling algorithm does not impose more interference than necessary to provide scheduling guarantees in mixed-criticality systems.

More recently, Baruah [20] deals with the certification problem in mixed-criticality systems. Their task model is defined from a certification perspective and is quite different from the one considered in this thesis. They consider both a reservation-based approach and a priority-based approach called Own Criticality Based Priority (OCBP) for addressing the problem. Resource augmentation results are also provided for the dual criticality version of OCBP. They also show the Mixed-Criticality schedulability to be NP-Hard in the strong sense. The analysis of OCBP was extended to an arbitrary number of criticality levels in [19].

To the best of our knowledge, we are the first to provide an implementation of mixed-criticality scheduling in the context of a real-time multi-core operating system. Our implementation leverages the existing Linux/RK infrastructure [91, 100].

2.5 Distributed Resource Kernel Framework

The Distributed Resource Kernel framework serves as the implementation platform for most of the algorithms described in this dissertation. Although many real-time kernels have been proposed in the past, we distinguish our resource kernel framework by considering resource isolation as an important requirement. Virtual machines like Xen [16] provide resource isolation at operating system granularity. Operating systems like QLinux [115] and Linux/RK [89] support performance isolation at the level of individual applications. Distributed RK framework extends this paradigm to provide support for end-to-end resource isolation for distributed applications.
Distributed operating systems have been well-studied in the literature. Many research systems like Amoeba [116], Locus [95], COSY [31] and Clouds [38] have established general principles like RPC, threads and group-communication, required to develop such systems. Communication infrastructures like MPI [60] and OpenMP [92] have also been developed for enabling parallel computing systems. Our framework focuses on distributed real-time systems, it therefore adapts these established distributed computing principles for the real-time environment and provides

temporal isolation through enforcement across the entire distributed system, which is not the major focus of the above-mentioned systems.

Many systems have looked at middleware-based approaches to developing end-to-end resource management in distributed systems [126] [86]. We follow the kernel-level approach primarily because: (i) middleware-based schemes require user-level participation and cooperation, therefore the fidelity of the system may be undermined, and (ii) the granularity of resource management supported by middle-ware may not be suitable to meet the performance requirements of all distributed real-time applications. Distributed RK operates at the kernel-level, and is therefore able to faithfully perform admission control, resource reservation and enforcement across all hosted system applications. Some of the services provided in our framework, like the global time service, also benefit greatly from kernel-level integration.

In the networking community, resource reservation protocols like RSVP [45] are popular for providing QoS guarantees over the Internet. Our idea of performing distributed admission control is similar to RSVP, but we are primarily interested in providing timing guarantees and hence follow a different admission control test. RSVP is restricted to managing the network resource, whereas our framework has the broader responsibility of managing all distributed system resources. The enforcement and replenishment mechanisms required in our context are also very different from those required for RSVP.

Distributed real-time scheduling approaches have been investigated since early distributed real-time operating systems like the Spring Kernel [112]. More recent systems like ARTS [122] have adopted fixed priority schemes for real-time scheduling in the distributed environment. Holistic schedulability analysis for such fixed-priority distributed real-time systems have also been developed [120]. In such systems, however, back-to-back scheduling problems may arise; Therefore, various solutions like Period Enforcers [96] and Release Guards [114] have been proposed. We employ a custom schedulability test that integrates computational deadlines, network latencies and time-synchronization error. This test is specific for the distributed task graph model and period enforcer technique that we use in our system. It also assumes bounded latency network reservation support.

Many commercial operating systems like Windows Vista [47] have started incorporating temporal resource reservation support. Distributed RK, however, is the first system to provide distributed temporal resource isolation through system-wide reservation and enforcement. Distributed RK brings together multiple concepts into a coherent and usable framework for realizing practical consolidated large-scale distributed real-time systems. The distributed RK abstraction of a distributed resource container can also be extended in the future to provide a unified interface for managing the timing, security and fault-tolerance properties of distributed real-time applications. More recently, $LITMUS^{RT}$ [29] has been developed as a platform for evaluating and testing multi-core scheduling algorithms. This platform shares many traits as the Resource Kernels implementation described in this dissertation in that they are both implemented for Linux and use minimally invasive extensions to the kernel. However, the key distinguishing factors of the Resource Kernel framework are the focus on resource isolation and its availability for distributed real-time platforms comprising of potentially multiple multi-core processors.

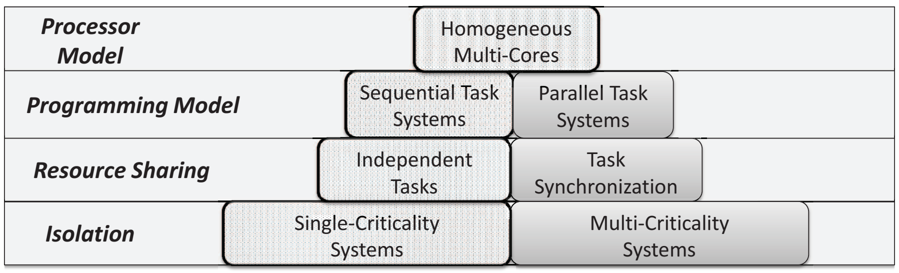
2.6 Summary

There are various categories of related work such as multi-processor and multi-core scheduling, multi-processor task synchronization, parallel processing, mixed-criticality systems, and real-time operating systems. The approach adopted in this dissertation is to use fixed-priority scheduling with static task partitioning and statically determined migration points. The benefits of this approach are its practicality and associated analytical guarantees. The main contribution of this dissertation are the utilization bound analyses, response-time analyses, and resource augmentation bounds for this specific approach. Our implementation in the distributed resource kernel framework also distinguishes our work from many existing results in literature by demonstrating its feasibility and quantifying system overheads.

Chapter 3

Independent Sequential Tasks

In this chapter, we consider the specific problem of scheduling *independent sequential tasks* on multi-core processors. In later chapters, we generalize the system model to include tasks with *synchronization* requirements and tasks with parallel execution segments. For the purposes of this chapter, each task τ_i has a single computation segment $< \psi_i^1 >$ (i.e. $s_i = 1$), where ψ_i^1 is of the form $[C_{i,1}^1]$ (i.e. $n_{i,j} = 1$). This representation corresponds to each task τ_i being both *sequential* and *independent*. For the purposes of this chapter, we will simplify the notation to use C_i to represent the worst-case computation time $C_{i,1}^1$ of the single computation segment of the independent task τ_i. Each task τ_i is therefore represented as (C_i, T_i, D_i), where C_i is the computation time, T_i is the period, and D_i is the relative deadline. Even though we are only concerned with scheduling implicit-deadline task sets (with $T_i = D_i$), during the course of task splitting we will introduce subtasks having $D_i \leq T_i$, hence we explicitly represent D_i. In this



chapter, we maintain an assumption of a single criticality domain.

In order to address the problem of scheduling independent sequential tasks on multi-core processors, we extend the class of partitioned deadline-monotonic scheduling (PDMS) algorithms by allowing the *highest-priority* task on a processor core to be split (HPTS) across more than one core. We prove that this extension achieves a schedulable per-processor utilization bound of 60% on implicit-deadline task-sets, under deadline-monotonic priority assignment. Under HPTS, a previously split task may be chosen again for splitting if it has the highest priority on a given core. A specific instance of this class of algorithms, in which tasks are allocated in the decreasing order of size (PDMS_HPTS_DS), can achieve a per-processor utilization bound of 65% on implicit-deadline task-sets. The behavior of PDMS_HPTS_DS on lightweight tasks is better, and a worst-case schedulable utilization of 69% on implicit-deadline task-sets is achieved when the individual task utilizations are less than 41.4%. When exact schedulability tests are used on the individual processors, PDMS_HPTS_DS is seen to achieve a much higher chedulable per-processor utilization (see Section 3.3.2).

The rest of this chapter is organized as follows. First, we provide a brief overview of the bin-packing problem, and show how splitting objects can achieve improved packing. We next translate the notion of dividing objects into splitting tasks, and compute the penalty of splitting the highest-priority task (HPTS). Utilization bounds are then developed for the class of PDMS algorithms under HPTS. The PDMS_HPTS_DS algorithm is then presented, its utilization bounds analyzed and its average-case performance evaluated.

In this chapter, we characterize the behavior of partitioned deadline-monotonic scheduling (PDMS) with task-splitting. Specifically, we focus on the effects of deadline-monotonic priority assignments and splitting the highest-priority task (HPTS), and show that this can lead to a utilization bound of 60% for this category of algorithms on implicit-deadline task-sets. A specific instance of this class, where tasks are allocated in the decreasing order of size using PDMS_HPTS_DS, is shown to have a higher utilization bound of 65%. This algorithm also performs better on lightweight task-sets achieving a utilization bound of 69%, when the individual task utilizations are restricted to be less than 41.4%.

The main advantages of PDMS_HPTS_DS are:

1. a higher utilization bound than conventional partitioned multiprocessor scheduling schemes,
2. a tight bound on the number of tasks straddling processing core boundaries (1 per core),
3. the ability to use both utilization-based tests and exact schedulability conditions,
4. high average-case utilization compared to non-splitting, and
5. strong processor affinity due to the partitioned nature.

3.1 Task Partitioning

The task partitioning problem is that of dividing the given set of tasks, $\{\tau_1, \tau_2, \cdots, \tau_n\}$, into m subsets such that each subset is schedulable on a single processing core.

The problem of task partitioning involves two components:

1. **Bin-Packing**: Dividing the taskset into m sub-sets.
2. **Uniprocessor Schedulability**: The sub-problem of ensuring that each subset is schedulable.

We briefly describe the classical bin-packing problem. Each bin B_j is of unit size, and each object O_i in the list of objects $\{O_1, O_2, \cdots, O_n\}$ to be packed has a size S_i ranging from 0 to 1. The bin-packing problem is to allocate the n objects to m bins, subject to the following constraint:

$$\forall j (1 \leq j \leq m) \sum_{\forall O_i \in B_j} S_i \leq 1$$

The average size (AS) for a given bin-packing problem instance is defined as:

$$AS = \frac{\sum_{i=1}^{n} S_i}{m}$$

The worst-case size-bound SB for bin-packing is defined as the greatest lower bound on the average size AS of all instances of the bin-packing problem that are unsolvable. Therefore, given a problem instance with average size $AS \leq SB$, there exists a solution to the bin-packing problem.[1]

The underlying bin-packing problem becomes the bottleneck for partitioned real-time scheduling schemes. It restricts the worst-case schedulable utilization to 50% per processor (see [6]), although uniprocessor scheduling is known to have a higher utilization bound (69% for RMS, 100% for EDF).

The classical bin-packing problem is simple to solve when any object is permitted to be split into two pieces, the sizes of which sum to the size of the object. The objects can be packed in any order, one bin at a time. When an object does not fit, it is split so that the current bin is exactly filled, and the residual part is placed in the next bin. This continues with each bin being filled to capacity and containing at most one split per bin. The final bin(s) will hold the remaining objects without any further splitting, since $AS \leq 1$. The total number of split objects cannot exceed $m - 1$. This discussion outlines the motivation behind our approach of splitting one task per processor to achieve a utilization considerably higher than the 50% restriction placed by conventional bin-packing when objects are not allowed to be split. In our context of fixed-priority scheduling, each bin (i.e. processor) cannot be filled up to 100% utilization since deadlines can be missed earlier. Since a task τ has three parameters (C, T, D), we need to define an appropriate mechanism for splitting tasks. We would like to do this with an eye towards improving schedulability. In our greedy object-splitting algorithm, we assumed that there was no penalty while splitting. That is, after splitting, the total size of the split objects did not change from the original.

When our real-time tasks are considered for splitting, however, the penalty of splitting can be non-zero. If a task τ is split into τ' and τ'', the subtasks τ' and τ'' will be assigned to different cores and both should not execute at the same time. The first piece τ' must execute first, and only after the completion of τ', can τ'' execute. Furthermore, both τ' and τ'' must complete within the same relative deadline of task τ. While other approaches can be adopted, we assume that each of

[1]This formulation is somewhat different from the original formulation of bin-packing where the objective is to minimize the number of bins to be used. We prefer our formulation in the current context where the number of processing cores is likely to be fixed on a given platform.

these subtasks is assigned its own local deadline, and the second subtask will be released when the deadline of the first subtask is reached. For instance, task τ with parameters (C, T, D) will be split into τ' and τ'' such that:

$$\tau' : (C', T, D') \qquad \tau'' : (C - C', T, D - R')\ (D' \leq D, R' \leq D')$$

where, R' represents the worst-case response time of subtask τ' on its allocated processor. This value may be obtained through an analysis of the critical instant of τ', as performed in the exact schedulability test for fixed-priority scheduling. Subtask τ'' will be eligible to execute on its processor R' time-units after τ' becomes eligible on its corresponding processor.

In our scheme, if a task on a processor is to be split, the highest-priority task τ_h on the processor is always chosen as the candidate for splitting. We refer to this scheme as HPTS (Highest-Priority Task Splitting). Let the split instances of the highest-priority task τ_h on a processor of interest be τ_h' and τ_h'', with τ_h' being scheduled on the same processor. If τ_h : (C_h, T_h, D_h) is split, we generate:

$$\tau' : (C_h', T_h, D_h) \qquad \tau'' : (C_h - C_h', T_h, D_h - C_h')$$

The special property that we exploit is that the highest-priority task τ_h' on a processor under fixed-priority scheduling has its worst-case response time R_h' equal to its worst-case computation time C_h'. The deadline of the second subtask τ'' is therefore maximized.

3.2 PDMS_HPTS

In this section, we analyze the performance of Partitioned Deadline-Monotonic Scheduling (PDMS) under deadline-monotonic priority assignments, when used with Highest-Priority Task-Splitting (HPTS). Conventional partitioned deadline-monotonic scheduling algorithms consist of a bin-packing strategy, which allocates tasks to processing cores in some sequence. Each time a task is allocated to a core, a schedulability test is invoked for the individual core to ensure that all its allocated tasks are schedulable under deadline-monotonic scheduling.

We utilize a routine called $HPTaskSplit$, which is invoked, whenever the schedulability test on a particular core fails on trying to assign task τ_f to processor P_j. We define the following data structures and operations:

- $Unallocated\_Queue$: Data structure to hold unallocated tasks in the order required by the bin-packing scheme.
- $Allocated\_Queue_j$: Data structure that holds the tasks allocated to processor P_j in priority order.
- $Enqueue(Unallocated\_Queue, \tau)$: Enqueues the task τ back into the unallocated task queue.
- $\tau \leftarrow Dequeue(Unallocated\_Queue)$: Dequeues a task τ from the unallocated task queue.
- $IsEmpty(Unallocated\_Queue)$: Returns TRUE, whenever the unallocated task queue is empty, signaling that there are no more tasks to be scheduled.
- $EnqueueHP(Allocated\_Queue_j, \tau)$: Enqueues task τ in priority order to allocated task queue of processor P_j.
- $\tau \leftarrow DequeueHP(Allocated\_Queue_j)$: Dequeues the highest-priority task τ from allocated task queue of P_j.
- $IsSchedulable(Allocated\_Queue_j)$: Performs the exact schedulability test on the tasks allocated to processor P_j under deadline-monotonic scheduling.
- $(\tau', \tau'') \leftarrow MaximalSplit(Allocated\_Queue(j), \tau)$: Splits task $\tau : (C, T, D)$ into $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$, so that processor P_j is maximally utilized when task τ' is added to it.
- $Available_j$: A boolean flag used to indicate whether processor P_j is available for scheduling.

The $IsSchedulable$ algorithm is readily derived from the exact schedulability tests for fixed-priority preemptive scheduling. Trivial utilization bound tests can be used to determine `MaximalSplit`.

Algorithm 1 $HPTaskSplit$

```
Require: $Unallocated\_Queue, Allocated\_Queue_j, \tau_f$
  $Removed\_Size \leftarrow 0$
  $Old\_Unallocated\_Queue \leftarrow Unallocated\_Queue$
  $Old\_Allocated\_Queue_j \leftarrow Allocated\_Queue_j$
  $EnqueueHP(Allocated\_Queue_j, \tau_f)$
  while $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ do
    $\tau \leftarrow DequeueHP(Allocated\_Queue_j)$
    $Removed\_Size \leftarrow Removed\_Size + S(\tau)$
    if $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ then
      $Enqueue(Unallocated\_Queue, \tau)$
    end if
  end while
  $(\tau', \tau'') = MaximalSplit(Allocated\_Queue_j, \tau)$
  if $(Removed\_Size - S(\tau')) \geq S(\tau_f)$ then
    $Unallocated\_Queue \leftarrow Old\_Unallocated\_Queue$
    $Allocated\_Queue_j \leftarrow Old\_Allocated\_Queue_j$
    $Available_j \leftarrow FALSE$
    return
  end if
  $EnqueueHP(Allocated\_Queue_j, \tau')$
  $Enqueue(Unallocated\_Queue, \tau'')$
  $Available_j \leftarrow FALSE$
  return
```

Although such tests are faster to perform, they will significantly reduce the average-case performance. A better implementation of $MaximalSplit(Allocated\_Queue_j, \tau)$ could perform a binary search to find the maximum computation time (C') of the task τ, at which

$IsSchedulable(Allocated\_Queue_j)$ *test is exactly satisfied.*

A more sophisticated version of $MaximalSplit$ uses a variation of the exact schedulability test for fixed-priority preemptive scheduling. In order to compute the maximum-computation time C' sustainable for task τ', we consider each task τ_k in $Allocated\_Queue_j$ and compute the maximum value C' of τ' at which τ_k still meets its deadlines[2].The minimum value of C' over all tasks τ_k in $Allocated\_Queue_j$ gives the maximum sustainable computation time (C') of τ. We can then create τ' to be $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$ from the original task-parameters of $\tau : (C, T, D)$.

3.2.1 Analysis of HPTS

We now provide an analysis of HPTS when the deadlines of given task-set $\tau : \{\tau_1, \tau_2, .., \tau_n\}$ are such that $\forall i$, $D_i = T_i$. These task-sets are referred to as implicit-deadline task-sets.

When a task τ is split, it results in sub-tasks with $D \leq T$, which are referred to as constrained-deadline task-sets.

A given task-set $\{\tau\}$ is *schedulable* by a scheduling algorithm A, if all the jobs released by all the tasks in $\{\tau\}$ meet their deadlines. This is denoted as $Schedulable(A, \{\tau\})$.

Let $NS(ID, A)$ represent the collection of all finite sets of implicit-deadline (ID) tasks that are not schedulable by algorithm A.

Let $NS(CD, A)$ represent the collection of all finite sets of constrained-deadline (CD) tasks that are not schedulable by algorithm A.

The *utilization bound* UB of a scheduling algorithm A is defined as the greatest lower bound (*glb*) of utilizations of *all* the task-sets not schedulable by A. Formally, the utilization bound of

[2] C' can be obtained by computing the maximum computation time of τ at which the worst-case response-time of τ_k coincides with a feasible scheduling point (as defined in [71]).

RMS for implicit-deadline (ID) tasks is given by:

$$UB(ID, RMS) = glb\{\sum_{\forall \tau_i \in \{\tau\}} U(\tau_i) | \forall \{\tau\} \in NS(ID, RMS)\}$$

Also, $\forall \{\tau\} \in ID$,

$$\sum_{\forall \tau_i \in \{\tau\}} U(\tau_i) < UB(ID, RMS) \implies Schedulable(RMS, \{\tau\}) \tag{3.1}$$

The size bound SB of a scheduling algorithm A is defined as the greatest lower bound (*glb*) of sizes of *all* the task-sets not schedulable by A.

The size bound of DMS for implicit-deadline (ID) tasks is given by:

$$SB(ID, DMS) = glb\{\sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) | \forall \{\tau\} \in NS(ID, DMS)\}$$

Also, $\forall \{\tau\} \in ID$,

$$\sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) < SB(ID, DMS) \implies Schedulable(DMS, \{\tau\}) \tag{3.2}$$

The size bound of DMS for constrained-deadline (CD) tasks is given by:

$$SB(CD, DMS) = glb\{\sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) | \forall \{\tau\} \in NS(CD, DMS)\}$$

Also, $\forall \{\tau\} \in CD$,

$$\sum_{\forall \tau_i \in \{\tau\}} S(\tau_i) < SB(CD, DMS) \implies Schedulable(DMS, \{\tau\}) \tag{3.3}$$

The following lemmas describe the relationship between the size bound and the utilization bound of each processor.

Theorem 1. *The size bound for uniprocessor deadline-monotonic scheduling (DMS) on constrained-deadline (CD) task-sets is equal to the utilization bound of rate-monotonic scheduling (RMS) on implicit-deadline (ID) task-sets.*

$$(SB(CD, DMS) = UB(ID, RMS))$$

Proof. Considering implicit-deadline tasks, Rate-Monotonic Scheduling is identical to Deadline-Monotonic Scheduling, since task periods equal their deadlines. It follows that $UB(ID, RMS) = SB(ID, DMS)$, since implicit-deadline tasks have a size equal to their utilization.

For implicit-deadline task-sets,

$$\sum_{i=1}^{n} \frac{C_i}{T_i} < UB(ID, RMS) = SB(ID, DMS) \Longrightarrow \mathit{Schedulable}(RMS, \{\tau_1, \tau_2, ..., \tau_n\})$$

(from (3.1))

Recall that implicit-deadline task-sets (I) are a subset of constrained-deadline task-sets (C). Therefore,

$$SB(CD, DMS) \leq SB(ID, DMS) = UB(ID, RMS) \tag{3.4}$$

For constrained-deadline task-sets,

$$\sum_{i=1}^{n} \frac{C_i}{D_i} < SB(CD, DMS) \Longrightarrow \mathit{Schedulable}(DMS, \{\tau_1, \tau_2, ..., \tau_n\}) \text{ (from (3.3))}$$

Now, consider if there exists a constrained deadline task-set $\{\tau_1, \tau_2, ..., \tau_n\}$ that is *not* schedulable under DMS but satisfies

$$\sum_{i=1}^{n} \frac{C_i}{D_i} < UB(ID, RMS)$$

Each constrained-deadline task τ in $\{\tau_1, \tau_2, ..., \tau_n\}$ can be transformed into a pessimistic implicit-deadline task $\tau*$, by artificially shortening the task period T to equal its deadline D. Under this transformation (C, T, D) to (C, D, D), the task size remains unchanged as the deadlines are not modified. Rate-monotonic priority assignments of $\{\tau*\}$ correspond to the deadline-monotonic priority assignments of $\{\tau\}$. If the transformed task-set $\{\tau*\}$ has a total utilization less than UB(ID,RMS), it is schedulable under RMS. However, the nature of the transformation also implies that the original task-set $\{\tau\}$ should be schedulable under DMS. This contradicts the assumption that $\{\tau\}$ is not schedulable, therefore

$$SB(CD, DMS) \geq UB(ID, RMS) = SB(ID, DMS) \tag{3.5}$$

From Inequalities 3.4 and 3.5, it follows that

$$SB(CD, DMS) = UB(ID, RMS)(= SB(ID, DMS))$$

$\square$

We now provide a brief description of the `HPTaskSplit` algorithm. `HPTaskSplit` is invoked with τ_f, only when the schedulability test on a particular processor P_j fails on assigning task τ_f. `HPTaskSplit` forcefully enqueues task τ_f to the allocated queue of P_j. While P_j remains unschedulable, `HPTaskSplit` continuously removes tasks from the allocated queue of P_j in the order of decreasing priorities, till P_j becomes schedulable. `MaximalSplit` is then invoked on the task τ, which was the last to be removed from P_j. If the total size removed from P_j to allocate τ_f is greater than the size of τ_f itself, it is not beneficial to add τ_f to P_j. `HPTaskSplit` therefore restore the old queues under such a condition, and decides not to invoke `MaximalSplit`.

Lemma 1. *The total size on each processing core P_j determined to be unavailable for scheduling by $HPTaskSplit$ is greater than or equal to $SB(CD, DMS) - \epsilon$ where*[3] $\epsilon \to 0$.

$$(Available_j = FALSE) \implies$$

$$\sum_{\tau_i \in P_j} S(\tau_i) \geq SB(CD, DMS) - \epsilon, \textit{ where } (\epsilon \to 0)$$

Proof. Two cases must be considered.

Case 1: `HPTaskSplit` decides to perform `MaximalSplit` on task τ, allocates the first piece τ' to processing core P_j and adds the second piece τ'' back to the $Unallocated\_Queue$.

Since `MaximalSplit` was called, the processing core P_j is maximally utilized. This means that increasing the computation time of sub-task τ' any further would render the tasks allocated to P_j unschedulable. Therefore, from Equation 3.3, the total size MS_{tot} of tasks allocated to the maximally utilized processor P_j must be greater than or equal to $SB(CD, DMS) - \epsilon$ (where $\epsilon \to 0$).

[3]To improve readability, we will assume $\epsilon = 0$ in our subsequent usage of size bound.

$$MS_{tot} = \sum_{\tau_i^{case\,1} \in P_j} S(\tau_i) \geq SB(CD, DMS) - \epsilon$$

$$\text{where } \epsilon \to 0$$

Case 2: `HPTaskSplit` decides not to invoke `MaximalSplit`, and instead restores the old allocated and unallocated queues.

This scenario occurs when the total size S_{tot} of the currently allocated tasks to P_j is greater than or equal to the total size obtainable through `MaximalSplit` (MS_{tot}).

$$S_{tot} = \sum_{\tau_i^{case\,2} \in P_j} S(\tau_i) \geq MS_{tot}$$

Using the analysis from previous case for MS_{tot},

$$S_{tot} = \sum_{\tau_i^{case\,2} \in P_j} S(\tau_i) \geq MS_{tot} \geq SB(CD, DMS) - \epsilon \text{ where } \epsilon \to 0$$

□

We now prove some useful properties about splitting the highest-priority task. First, the next Lemma proves that whenever task-splitting is performed, the operation does not increase the size of the unallocated task-set.

Lemma 2. *When a task $\tau : (C, T, D)$ is split into sub-tasks $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$, the sizes of both τ' and τ'' are less than or equal to that of τ. That is, $S(\tau') \leq S(\tau)$ and $S(\tau'') \leq S(\tau)$.*

Proof. $(\forall C', 0 \leq C' \leq C$ and $\forall C, C \leq D)$

We have, $S(\tau) = \frac{C}{D}$,

$S(\tau') = \frac{C'}{D} \leq \frac{C}{D} = S(\tau)$ and $S(\tau'') = \frac{C-C'}{D-C'} \leq \frac{C}{D} = S(\tau)$

□

Lemma 3. *When a task $\tau : (C, T, D)$ is split into sub-tasks $\tau' : (C', T, D)$ and $\tau'' : (C - C', T, D - C')$,*

1. $S(\tau'') = \frac{S(\tau) - S(\tau')}{1 - S(\tau')}$

2. $S(\tau') + S(\tau'') \leq 2(1 - \sqrt{1 - S(\tau)})$

Proof. For simplicity of notation, we let $S(\tau) = S$, $S(\tau') = S' = x$, and $S(\tau'') = S''$

$$S'' = \frac{C - C'}{D - C'} = \frac{\frac{C}{D} - \frac{C'}{D}}{\frac{D}{D} - \frac{C'}{D}} = \frac{S - x}{1 - x}, \tag{3.6}$$

$$S' + S'' = x + \tfrac{S-x}{1-x}$$

(hence claim (1) is established)

Maximizing w.r.t. x subject to $0 \leq x \leq S$ yields,

$$S' = S'' = 1 - \sqrt{1 - S},$$

hence, $S(\tau') + S(\tau'') \leq 2(1 - \sqrt{1 - S(\tau)})$. □

The penalty due to task-splitting is the increase in the size of the split task, which is quantified as:

$$\delta = S(\tau') + S(\tau'') - S(\tau) \leq 2(1 - \sqrt{1 - S(\tau)}) - S(\tau) \tag{3.7}$$

The incurred penalty δ is an increasing function of the task size $S = S(\tau)$. Specifically this shows that the penalty of task splitting is as low as 2% when the maximum size of any task in the system is less than or equal to 25%.

3.2.2 Utilization Bound

The *utilization bound* $UB_m(A)$ of a multi-processor scheduling algorithm (A) is defined as the greatest lower bound (*glb*) of per-processor utilization of *all* the task-sets with individual task-utilizations less than or equal to 1, which are not schedulable by A.

Considering implicit-deadline task-sets (ID) scheduled on m processors,

$$UB_m(ID, A) = glb\{\frac{\sum\limits_{\forall \tau_i \in \{\tau\}} U(\tau_i)}{m} | \forall \{\tau\} \in NS(ID, A)\}$$

Also, $\forall\{\tau\} \in \{ID, m\}, \forall\tau_i \in \{\tau\}$,

$$U(\tau_i) \leq 1 \wedge \sum_{\forall\tau_i\in\{\tau\}} U(\tau_i) < m * UB_m(ID, A) \Longrightarrow$$
$$Schedulable(A, \{\tau\}) \tag{3.8}$$

The *size bound* $SB_m(A)$ of a multi-processor scheduling algorithm (A) is defined as the greatest lower bound (*glb*) of per-processor size of *all* the task-sets with individual task-sizes less than or equal to 1, which are not schedulable by A.

Considering constrained-deadline task-sets (CD) scheduled on m processors,

$$SB_m(CD, A) = glb\{\frac{\sum\limits_{\forall\tau_i\in\{\tau\}} S(\tau_i)}{m} | \forall\{\tau\} \in NS(CD, A)\}$$

Also, $\forall\{\tau\} \in \{CD, m\}, \forall\tau_i \in \{\tau\}$,

$$S(\tau_i) \leq 1 \wedge \sum_{\forall\tau_i\in\{\tau\}} S(\tau_i) < m * SB_m(CD, A) \Longrightarrow$$
$$Schedulable(A, \{\tau\}\}) \tag{3.9}$$

We now restrict our discussion to task-sets with task periods equal to their deadlines. We are interested in finding the utilization bound $UB_m(ID, PDMS\_HPTS)$ below which all given task-sets are schedulable by any multiprocessor algorithm in PDMS_HPTS.

This is equivalent to finding $SB_m(ID, PDMS\_HPTS)$ since periods equal deadlines.

Let us consider any task-set which is not schedulable by $PDMS\_HPTS$. The total size (S_{tot}) is given by:

$$S_{tot} = \sum_{i=1}^{n} S(\tau_i)$$

Task-splitting adds a maximum of $(m-1)$ new tasks (one per each core, except the last). Therefore, in the worst-case, it increases the total size of the task-set by $(m-1)*\delta$. The cumulative task size of the task-set after task-splitting is:

$$CS^{TS} = S_{tot} + (m-1)*\delta = \sum_{i=1}^{n} S(\tau_i) + (m-1)*\delta.$$

For the entire task-set to be unschedulable, the $Available_j$ flag must be $FALSE$ for all m processing cores. Invoking Lemma 1 however, we see that each processing core must have a total size at least equal to $SB(CD, DMS)$, therefore

$$\sum_{i=1}^{n} S(\tau_i) + (m-1) * \delta \geq m * SB(CD, DMS).$$

Invoking Theorem 1 and re-writing, we obtain

$$\sum_{i=1}^{n} S(\tau_i) \geq m * UB(ID, RMS) - (m-1) * \delta,$$

therefore $SB_m(CD, PDMS\_HPTS)$ is at least:

$$\begin{aligned} SB_m(CD, PDMS\_HPTS) &\geq \frac{\sum_{i=1}^{n} S(\tau_i)}{m} \\ &\geq UB(ID, RMS) - \delta + \frac{\delta}{m}. \end{aligned}$$

Recall that the collection of finite implicit-deadline task-sets (I) is a subset of the collection of finite constrained-deadline tasks (C), therefore

$$\begin{aligned} SB_m(ID, PDMS\_HPTS) &\geq \frac{\sum_{i=1}^{n} S(\tau_i)}{m} \\ &\geq UB(ID, RMS) - \delta + \frac{\delta}{m}. \end{aligned}$$

When all tasks have periods equal to their deadlines, the size-bound is the same as the utilization bound; therefore,

$$UB_m(ID, PDMS\_HPTS) \geq UB(ID, RMS) - \delta + \frac{\delta}{m}.$$

The utilization bound for RMS on implicit-deadline tasks is 0.6931 [74], therefore

$$UB_m(ID, PDMS\_HPTS) \geq 0.6931 - \delta + \frac{\delta}{m}.$$

As $m \to \infty$, we get the utilization bound to be $0.6931 - \delta$. As we have shown earlier, when the individual task sizes are less than or equal to 25%, the size penalty (δ) of task splitting is less than 2%, leading to an utilization bound of 67.31% for such task-sets.

3.2.3 General Case

Lemma 4. *Given a task-set consisting of $r+1$ tasks, one of which has size S and all of which have deadlines less than or equal to their periods (CD), then the corresponding size bound for Deadline Monotonic Scheduling, $SB(CD, DMS|S|r)$ is*

$$SB(CD, DMS|S|r) = r((\frac{2}{1+S})^{\frac{1}{r}} - 1) + S.$$

Letting $r \to \infty$, the asymptotic bound is given by

$$SB(CD, DMS|S) = \ln(\tfrac{2}{1+S}) + S.$$

Proof. Consider a pessimistic task-set $\{\tau*\}$ obtained by artificially shortening the task period of each task to equal its deadline. The utilization of the $\tau*$ task-set is the same as the size of the original τ task-set. Furthermore, the Rate Monotonic priorities for the $\tau*$ task-set are the same as the Deadline Monotonic priorities for the original τ task-set. One can use the results of [113] who analyzed the behavior of the polling server to find a utilization bound for the Rate Monotonic scheduling of the $\tau*$ task-set. Specifically, for $r + 1$ tasks, one of which has size S (utilization in the $\tau*$ task-set), the bound is given by

$$SB(CD, DMS|S, r) =$$

$$UB(ID, RMS|S, r) = r((\frac{2}{1+S})^{\frac{1}{r}} - 1) + S. \tag{3.10}$$

Letting $r \to \infty$, we find the asymptotic bound

$$SB(CD, DMS|S) = UB(ID, RMS|S) = \ln(\tfrac{2}{1+S}) + S.$$

□

Theorem 2. *The utilization bound of the PDMS_HPTS class of algorithms for task-sets with task deadlines equal to task periods is at least 60%. That is,*

$$UB_m(ID, PDMS\_HPTS) \geq 60\%$$

Proof. Since task deadlines equal periods, the size bound and the utilization bound for $PDMS\_HPTS$ are the same.

Tasks with individual sizes greater than or equal to 60% can be allocated to their own processors as they have a size greater than or equal to the claimed size bound. These tasks can then be ignored for analyzing the worst-case size bound, since they can only increase the overall utilization.

We now need to consider the two-cases of `HPTaskSplit`.

Case 1: `HPTaskSplit` does not call `MaximalSplit` and returns the original allocated and unallocated task-queues.

In this scenario, from Lemma 1, the total size S_{tot} allocated to P_j is at least $SB(CD, DMS)$. From Theorem 1,

$$S_{tot} \geq SB(CD, DMS) = UB(ID, RMS) = 0.6931$$

These processing cores are allocated total sizes greater than or equal to 60% and also can be ignored.

Case 2: `HPTaskSplit` decides to call `MaximalSplit`.

Consider a scenario in which a processor, P_j, overflows. Let the task being split be denoted by τ. The split piece of this task remaining on this processor is denoted by τ' and the remaining piece of the task is denoted by τ''. The algorithm ensures through task-splitting that the individual processor is maximally utilized, therefore the total size of the tasks allocated to the processor must be greater than or equal to the size-bound for DMS.

P_j has a given task τ' of size $S(\tau')$. Hence, from Lemmas 1 and 4, the total size S_{tot} of tasks allocated to the P_j satisfies

$$S_{tot} \geq SB(CD, DMS|S(\tau')) = \ln(\frac{2}{1+S(\tau')}) + S(\tau'). \tag{3.11}$$

Task-splitting increases the size of the task-set by δ, therefore subtracting (Equation 3.7) $(S(\tau') + S(\tau'') - S(\tau))$ from (Equation 3.11) gives the total size of the original task-set allocated per-processor (S_{orig}),

$$S_{orig} \geq \ln(\frac{2}{1+S(\tau')}) + S(\tau) - S(\tau''). \tag{3.12}$$

Using (claim 1) of Lemma 3, we obtain

$$S_{orig} \geq \ln(\tfrac{2}{1+S(\tau')}) + S(\tau) - \tfrac{S(\tau)-S(\tau')}{1-S(\tau')}.$$

For simplicity, let $S = S(\tau)$ represent the size of task τ and $x = S(\tau')$ represent the size of task τ'. Then,

$$SB_m(CD, PDMS\_HPTS) = \ln(\frac{2}{1+x}) + S - \frac{S-x}{1-x} \tag{3.13}$$

Observe that (3.13) is a non-increasing function of $S = S(\tau)$. The minimum lower bound of $SB_m(CD, PDMS\_HPTS)$, therefore, occurs at the maximum value of $S(\tau)$. We are allocating individual processors to all tasks with size greater than or equal to $SB_m(CD, PDMS\_HPTS)$, therefore the maximum value of $S(\tau)$ is $SB_m(CD, PDMS\_HPTS)$.

Therefore, using $SB_m(CD, PDMS\_HPTS) = S$

$$\ln(\frac{2}{1+x}) - \frac{S-x}{1-x} = 0. \tag{3.14}$$

Rewriting S as a function of x gives

$$S = (1-x)\ln(\frac{2}{1+x}) + x. \tag{3.15}$$

To find the minimum value of S, differentiate S w.r.t. x and equate it to 0, to find the minimizing value.

$$\ln(\frac{2}{1+x}) - \frac{2x}{1+x} = 0. \tag{3.16}$$

Substituting $t = \frac{2}{1+x}$,

$$\ln(t) - (2-t) = 0 \Longrightarrow \ln(t) + t = 2 \Longrightarrow te^t = e^2$$

Solving this equation gives[4] $t = LambertW(e^2)$

Re-substituting $x = \frac{2-t}{t}$, we get

$$x = S(\tau') = \frac{2 - LambertW(e^2)}{LambertW(e^2)} = 0.2837, \tag{3.17}$$

[4] $LambertW(t)$ is the inverse of the function $f(t) = te^t$.

The second derivative of (3.15) is positive, therefore the value of x given above corresponds to the minimum value of S.

Corresponding value of $S = S(\tau) = SB_m(CD, PDMS\_HPTS)$ is 60.03%. The initially given task-set is assumed to have deadlines equal to their period, the size bound for such task-sets is the same as the utilization bound. We have accommodated the penalties due to task-splitting on each processor, and therefore we conclude that the utilization bound of PDMS_HPTS on task-sets with implicit deadlines is at least 60.03% (with the exception of the last processor, which can achieve up to 69.31%).

$$UB_m(ID, PDMS\_HPTS) = 0.6003 + \tfrac{0.0928}{m}.$$

As $m \to \infty$, the $UB_m(ID, PDMS\_HPTS) \to 60.03\%$. $\square$

We have shown above that for the general class of PDMS algorithms, HPTS can achieve an utilization of at least 60.03%. Depending on the specific scheme used for bin-packing, higher utilization bounds may be achievable.

The usefulness of this result is the flexibility in choosing the bin-packing scheme. The task allocation phase can therefore be guided by multiple heuristics like (i) minimizing communication across cores, (ii) co-locating cache-collaborative tasks, and (iii) allocating replicas of the same task on different processing cores. The HPTS technique ensures that even when any of these different heuristics are used, the system can achieve a utilization bound of 60%. Task-splitting may displace previously allocated tasks under some bin-packing schemes; this is a trade-off to achieve higher system utilization. The bin-packing algorithm could decide to not use HPTS on certain processors and settle for a lower utilization value, whenever the already allocated tasks should not be displaced. We believe that this flexibility would be very useful in practical applications with various constraints on allocating tasks.

3.2.4 Recent Work on Task-Splitting

A brief comparison of PDMS_HPTS with recent work on task-splitting [5, 9, 62] follows:

1) Each processing core uses purely fixed-priority preemptive scheduling, in contrast to mixed-priority schemes considered elsewhere. Our approach is therefore readily usable on fixed-priority reservation-based OSs like Linux/RK [90].

2) Under HPTS, task-splitting represents purely sequential migration across processors. For split tasks, when the reserve allocated on a particular processor completes, it is subsequently released on the next processor. In our current implementation for Linux/RK, when the task reserve is exhausted, the OS scheduler explicitly uses the already existing `sched_set_affinity()` function to migrate the task. No other support for inter-processor synchronization is required to ensure that the same task is not scheduled in parallel.

3) No special dispatching support or explicit suspensions are required, thus tasks may be even split across critical sections. However, the migration overhead increases the duration of the critical section, and hence it adversely affects the other tasks waiting on the critical section. After migration, the critical section becomes a global critical section [97] and hence remote-blocking effects need to be accounted for. In practice, critical sections are small and splitting may therefore be avoided at the cost of a slight loss in utilization.

4) Our task-splitting involves a single split per-processor thus trivially bounding the additional number of total preemptions. In this respect, our approach is similar to that of [5], although we do not provide tunable parameters to adjust the number of preemptions.

3.3 PDMS_HPTS_DS

In this section, we present a specific instance of the $PDMS\_HPTS$ class of algorithms in which tasks are allocated in decreasing order of size ($PDMS\_HPTS\_DS$). In describing this algorithm, we will use the same data-structures and notation used in the previous section to analyze PDMS_HPTS. $Enqueue$ and $Dequeue$ are modified as follows to reflect the decreasing size order of task allocation:

- $EnqueueDS(Unallocated\_Queue, \tau)$: Enqueues task τ in decreasing size order to unallocated task queue.
- $\tau \leftarrow DequeueDS(Unallocated\_Queue)$: Dequeues task τ with largest size ($\frac{C}{D}$) from unallocated task queue.

Algorithm 2 PDMS_HPTS_DS

```
**Require:** $Unallocated\_Queue$
  $j \leftarrow 1$
  $Allocated\_Queue_j \leftarrow EMPTY$
  while $IsEmpty(Unallocated\_Queue) \neq TRUE$ do
    $\tau \leftarrow DequeueDS(Unallocated\_Queue)$
    $Old\_Allocated\_Queue_j \leftarrow Allocated\_Queue_j$
    $EnqueueHP(Allocated\_Queue_j, \tau)$
    if $IsSchedulable(Allocated\_Queue_j) \neq TRUE$ then
      $Allocated\_Queue_j \leftarrow Old\_Allocated\_Queue_j$
      $HPTaskSplit(Unallocated\_Queue,$
                                  $Allocated\_Queue_j, \tau)$
      $j \leftarrow j + 1$
    end if
  end while
```

3.3.1 Worst-Case Behavior

The following theorem shows that a larger utilization bound holds for the PDMS_HPTS_DS algorithm.

Theorem 3. *The utilization bound of PDMS_HPTS_DS for task-sets with implicit deadlines is at least 65.47%.*

Proof. As in the proof of Theorem 2, tasks with individual sizes greater than or equal to 65.47% are allocated to their own processors as they have a size greater than or equal to the size bound. These tasks do not affect the per-processor allocated sizes, and can be ignored. Now, consider the two cases of `HPTaskSplit`.

Case 1. `HPTaskSplit` decides to not perform `MaximalSplit`.

Theorem 1 and Lemma 1 show that the total size S_{tot} allocated to processor P_j is at least $SB(CD, DMS)$, that is

$S_{tot} \geq SB(CD, DMS) = UB(ID, RMS) = 0.6931$.

These processing cores are allocated cumulative sizes greater than or equal to 65.47%; therefore, they can be ignored in our analysis for the worst-case per-processor utilization.

Case 2: `HPTaskSplit` decides to perform `MaximalSplit` on processor P_j.

In this case, the processor overflows on allocating some task, say τ_f; therefore, the highest priority task τ in the processor P_j is split. Let the split pieces be denoted by τ' and τ''. Let the total number of tasks allocated to P_j be $(r + 1)$ including τ'. Task τ' is obtained through `MaximalSplit`; therefore, by definition the processor is maximally utilized w.r.t. τ', i.e. any increase in the computation time of τ' renders the processor to be unschedulable.

Let the cumulative sum of the task-sizes excluding τ and τ_f be CS_r, so

$$CS_r + S(\tau_f) + S(\tau') > CS_r + S(\tau), \tag{3.18}$$

where the left-hand size of the above expression gives the cumulative sum of task sizes after allocating τ_f, and the right-hand side gives a lower-bound on the cumulative task-sizes before allocation τ_f. If the inequality (3.18) does not hold good, `HPTaskSplit` would have decided to not invoke `MaximalSplit`.

Simplifying and re-writing (3.18) gives

$$S(\tau) - S(\tau_{'}) < S(\tau_f). \tag{3.19}$$

Let the size of the task τ be denoted by $S = S(\tau)$ and the size of sub-task τ' be denoted by $x = S' = S(\tau')$.

The task has deadlines D less than or equal to the period T, and the processor is maximally utilized under deadline-monotonic scheduling (after adding τ' with size x). Using Lemma 1 and Lemma 4, the total allocated size on the processor is greater than or equal to:

$$SB(CD, DMS|x|r) = r((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \tag{3.20}$$

First, let us consider the case when $r = 1$. Re-writing (3.20) gives

$$SB(CD, DMS|x|1) = \frac{1-x}{1+x} + x.$$

Accommodating the penalty of task-splitting from (3.7), we get the size bound of PDMS_HPTS_DS to be:

$$SB(CD, PDMS\_HPTS\_DS) = \frac{1-x}{1+x} + S - \frac{S-x}{1-x}.$$

Let $\beta = SB(CD, PDMS\_HPTS\_DS)$.$\beta$ is a non-increasing function in S ($\forall\ 0 \leq x \leq 1$),therefore we want to use the maximum value of S to find the minimum β. The maximum possible value of S in our case is 0.6547, since tasks greater than this value are allocated their own processing cores and task-splitting does not increase the size of tasks (from Lemma 2). We therefore have that,

$$\beta = \frac{1-x}{1+x} + 0.6547 - \frac{0.6547-x}{1-x}.$$

Minimizing this function w.r.t. x, we find that the minimum occurs at $x = 0.4129$.

The corresponding value of $SB(CD, PDMS\_HPTS\_DS)$ is given by

$$\beta = 0.6584 \geq 0.6547.$$

When $r = 1$, we see that $SB(CD, PDMS\_HPTS\_DS)$ is greater than or equal to 0.6547, which is the proposed size-bound for PDMS_HPTS_DS. We therefore need to prove that the bound also holds good for $r \geq 2$.

The PDMS_HPTS_DS algorithm always allocates tasks in the decreasing order of size; therefore, the size of the last-task considered for allocation, τ_f, must be less than or equal to the size of all the other tasks allocated to P_j (except possibly τ' which was split from task τ). The size bound of the processor P_j can now be refined:

1. When $S(\tau_f) < ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, apply (3.20) to obtain

$$SB^{case1}(DMS|x|r) = r((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \tag{3.21}$$

   Adjusting the overhead of task-splitting from (3.7), we obtain the size bound of PDMS_HPTS_DS as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + S - S(\tau'').$$

   Using (claim 1) of Lemma 3 and re-writing gives,

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - S)\tfrac{x}{1-x}.$$

   We see that β is a non-increasing function of S ($\forall\ 0 \leq x \leq 1$), therefore we want to maximize S in order to find the minimum value of β. Using (3.19):

$$S < x + S(\tau_f).$$

   Currently, we are considering the case
   $S(\tau_f) < ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, so we can minimize β as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}.$$

   Re-writing gives,

$$\beta = (r - \tfrac{x}{1-x})((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + x.$$

2. When $S(\tau_f) \geq ((\frac{2}{1+x})^{\frac{1}{r}} - 1)$, all the r tasks (other than τ') must have sizes greater than or equal to the size of the last, and the sub-task τ' has a size x, therefore:

$$SB^{case2}(DMS|x|r) = r(S(\tau_f)) + x. \tag{3.22}$$

Adjusting the overhead of task-splitting from (3.7), we get the size bound of PDMS_HPTS_DS to be:

$$\beta = r(S(\tau_f)) + S - S(\tau'')$$

Using (claim 1) of Lemma 3 and re-writing gives,

$$\beta = r(S(\tau_f)) + (1 - S)\tfrac{x}{1-x}.$$

We see that β is a non-increasing function of S ($0 \leq x \leq 1$); therefore, we want to maximize S in order to find the minimum value of β. Using (3.19) we get

$$S < x + S(\tau_f).$$

Currently, we are considering the case
$S(\tau_f) \geq ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1)$, or equivalently,
$S(\tau_f) = ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1 + \delta)$(for some $\delta \geq 0$);
Therefore, we can minimize β as:

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1 + \delta) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1 + \delta))\tfrac{x}{1-x}.$$

Re-writing this, we get

$$\begin{aligned}\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}\\ +\delta(r - \tfrac{x}{1-x})\end{aligned}$$

This is an increasing function of δ for $r \geq 2$ (since $x \leq 0.6547$); therefore, the minimum value of β occurs at $\delta = 0$. So, we obtain

$$\beta = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - x - ((\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x}.$$

Rewriting gives,

$$\beta = (r - \tfrac{x}{1-x})((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + x,$$

which is the same as the previous case. Therefore, the size-bound of PDMS_HPTS_DS ($\beta = SB(CD, PDMS\_HPTS\_DS)$) is determined by

$$\beta = (r - \frac{x}{1-x})((\frac{2}{1+x})^{\frac{1}{r}} - 1) + x. \tag{3.23}$$

For $r \geq 2$ this function β reaches a minimum of 0.6547 at $r = 2$, hence the utilization bound for $PDMS\_HPTS\_DS$ is atleast 0.6547. □

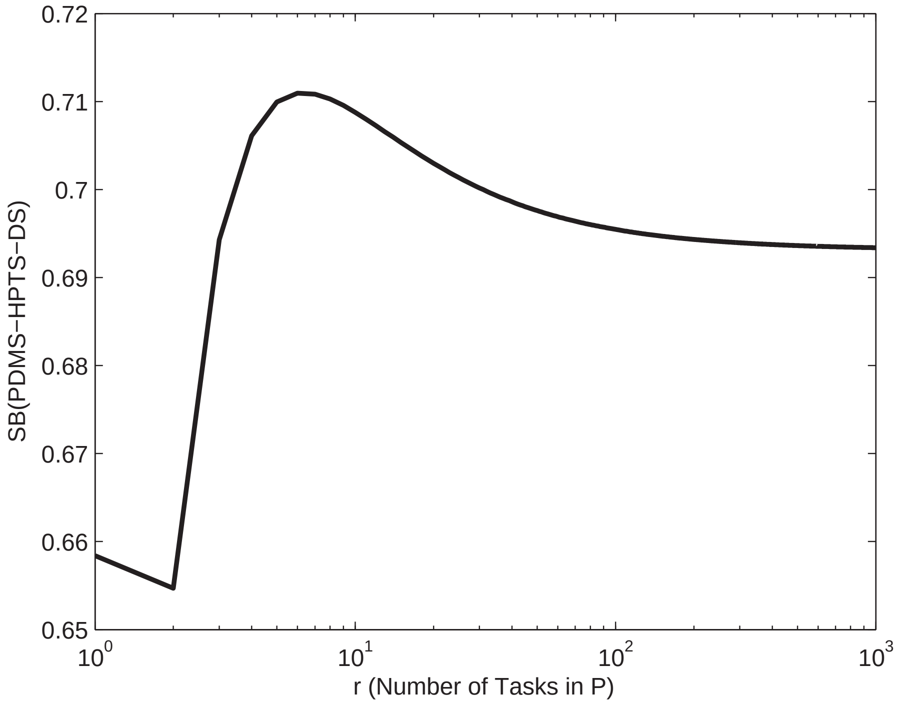
Let 'r' represent the number of tasks allocated to a processor core P. The minimum values of $SB(CD, PDMS\_HPTS\_DS)$ as a function of r are shown in the logarithmic plot Fig. 3.1(a).

The minimum value of SB(CD,PDMS_HPTS_DS), 0.6547, occurs at $r = 2$. The value of $SB(CD, PDMS\_HPTS\_DS)$ converges to 0.693 as $r \to \infty$. The size bound of PDMS_HPTS_DS is therefore 65.47%. For all tasksets with implicit deadlines, the utilization bound of PDMS_HPTS_DS is the same as the size bound, and hence at least 65.47%.

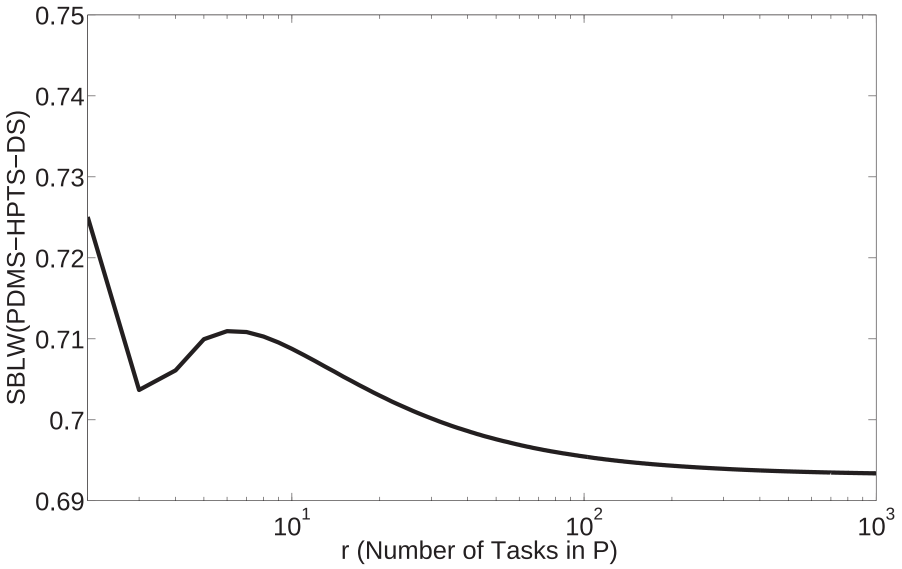
The asymptotic behavior suggests that PDMS_HPTS_DS achieves a utilization bound of 0.693 when the task-set is composed of sufficiently lightweight tasks. Space limitations prevent a presentation of the detailed analysis, but it can be easily derived that the size-bound of PDMS_HPTS_DS, for lightweight tasks with maximum utilization α, is given by:

$$\begin{gathered} SBLW(CD, PDMS\_HPTS\_DS, \alpha) \\ = r((\tfrac{2}{1+x})^{\frac{1}{r}} - 1) + (1 - MIN(\alpha, (\tfrac{2}{1+x})^{\frac{1}{r}} - 1))\tfrac{x}{1-x} \end{gathered}$$

The behavior of this function, when $\alpha = \sqrt{2} - 1 = 0.414$ is given in Fig. 3.1(b). The minimum value of $SBLW(CD, PDMS\_HPTS\_DS, \sqrt{2} - 1)$ is 0.693. The choice of $\alpha = 0.414$ implies that each processing core is forced to have more than 2 tasks, which can be guaranteed on a significant number of practical task-sets. Thus, the PDMS_HPTS_DS algorithm achieves the classical Rate-Monotonic Scheduling utilization bound of 0.6931, for sufficiently lightweight tasks (less that 41.4%).



(a) SB(CD,PDMS_HPTS_DS) as a function of r



(b) SBLW(CD,PDMS_HPTS_DS,$\sqrt{2}-1$) as a function of r

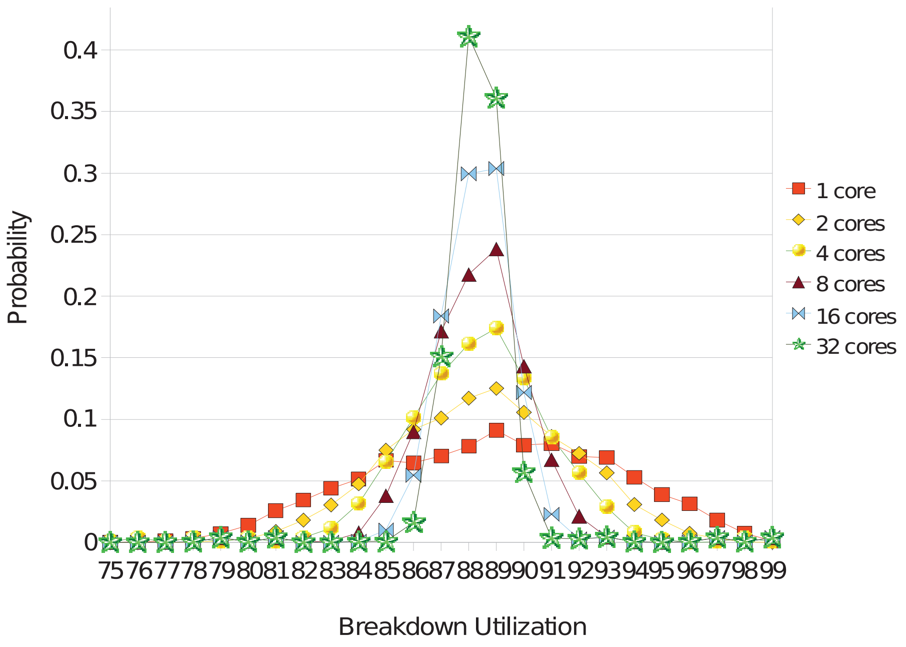


Figure 3.1: Breakdown Utilization PDF (PDMS_HPTS_DS

3.3.2 Average-Case Behavior

We have so far analyzed the worst-case performance of the PDMS_HPTS_DS algorithm and obtained utilization bounds on its adversarial task-sets. The worst-case performance occurs when all the task periods are related in a non-harmonic fashion and the task-splitting happens to divide the tasks into the worst possible pieces. These conditions represent extreme situations, and therefore the average-case performance of PDMS_HPTS_DS is expected to be far better than its utilization bound of 65%. In this section, we determine the probability density function of task-set breakdown utilizations, when the task-periods are chosen at random with a uniform distribution. This methodology is consistent with classical average-case analysis, in which the mean-performance of an algorithm is evaluated for a randomly chosen input. Earlier sections have shown the worst-case analysis. The best case is achieved when task periods are harmonic and each core is completely filled.

We studied the average-case performance of the PDMS_HPTS_DS algorithm on randomly

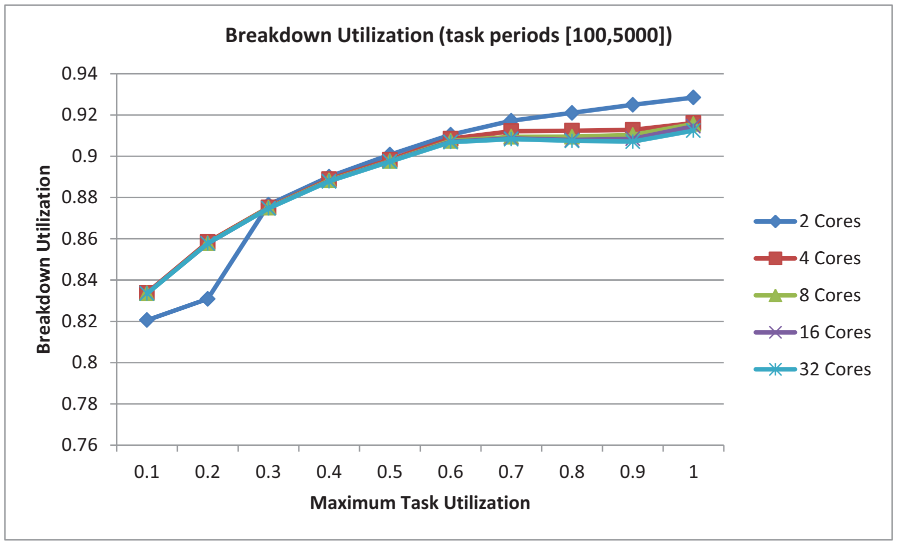


Figure 3.2: Average Breakdown Utilization with Varying Maximum Task Utilization

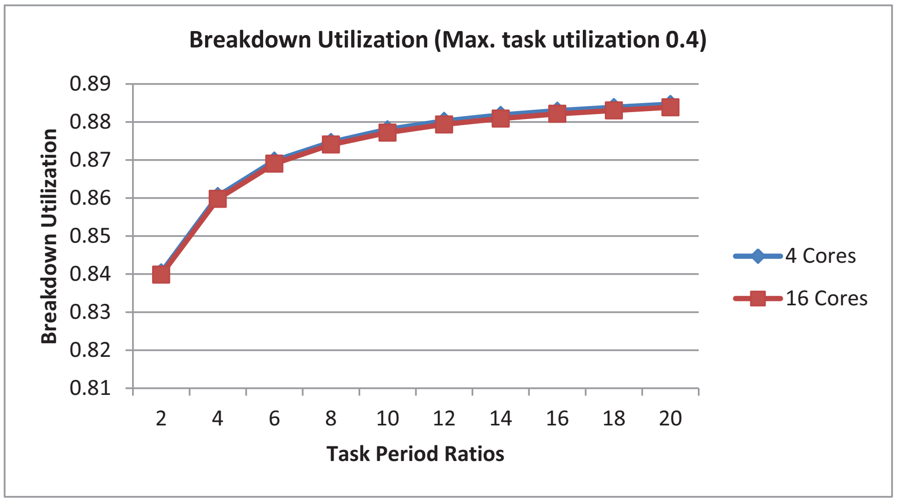


Figure 3.3: Average Breakdown Utilization with Varying Maximum Task Period Ratios (r)

generated task-sets and the break down utilization values were computed as given in [71]. Task period T for each task was chosen in an uniformly random fashion from the interval $[100, 5000]$. Computation time C for each task was chosen using a uniform distribution from the interval $[0, 0.4T]$. Tasks were generated until the total utilization exceeded $m * 100\%$ (where m is the number of cores). These computation times were then proportionally scaled down to compute the break down utilization (for more details see [71]). Although a slight decrease in the average utilization was seen with increasing number of processors, the value converges around 88%. The probability density function of the task-set breakdown utilizations is given in Fig.3.1. As we can see, with an increasing number of processors, the variance in the breakdown utilization decreases, indicating that more and more task-sets reach the average-case utilization. This behavior is due to the fact that as the number of cores increases, there is more flexibility to achieve good packings.

Figure 3.2 shows the average breakdown utilization with varying maximum task utilizations. The task sets are randomly generated with a task period that is selected from $[100, 5000]$ in a uniformly random fashion and utilization from 0 to *maximum utilization* (horizontal axis) in a uniformly random fashion. We see that when tasks are selected from a larger range of utilization values the utilization bound increases, mostly due to fewer number of tasks per processor core, which has a higher per-processor utilization bound (see [74]). In PDMS_HPTS, the last core does not incur any task splitting. The savings in splitting overheads gets amortized over the number of processor cores. This results in slightly higher utilization bound for the 2 core scenario.

We also varied the task period ratios in Figure 3.3, where the task periods where chosen from 100 to $100r$ (where r is the horizontal axis) in a uniformly random fashion. Maximum task utilization was maintained at 40%. We observe that as the task period ratios increase the average schedulable utilization increases (as observed in [71]).

3.4 Summary

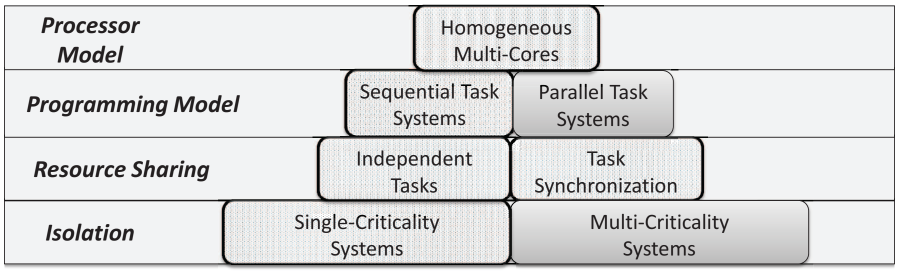
In this chapter, we have studied the building block of scheduling *independent sequential* tasks on multi-core processors. Later chapters will generalize this model to include *synchronization* and *parallel* tasks, while adopting the approach of *task splitting* to achieve higher utilization. In this chapter, the objects we are concerned with are the *tasks* themselves. We introduced the mechanism of Highest-Priority Task-Splitting (HPTS) and used it to improve the utilization bound of the class of partitioned deadline-monotonic scheduling algorithms (PDMS) from 50% to 60% on implicit-deadline task-sets. A specific instance of this class of algorithms PDMS_HPTS_DS has been shown to achieve a utilization bound of 65% for implicit-deadline task-sets. PDMS_HPTS_DS is also shown to achieve a higher utilization bound of 69% on lightweight implicit-deadline task-sets, where the individual task utilizations are restricted to a maximum of 41.4%. The average-case analysis of PDMS_HPTS_DS using randomly generated task-sets shows that it achieves a higher utilization on the average. Future work involves extensions to the basic PDMS_HPTS_DS scheme that exploit relationships between the task periods may achieve a higher utilization bound.

Chapter 4

Synchronization of Sequential Tasks

In this chapter, we relax the assumption of *independent* tasks, which was made in Chapter 3. This is an important consideration, given that task synchronization is a key problem faced in many real-world systems. Available solutions in the uniprocessor context like the Priority Ceiling Protocol (PCP) [79] have been extended to the multiprocessor scenario [103, 106]. In this chapter, we detail some of the scheduling penalties arising due to multiprocessor task synchronization, and analyze them under different execution control policies. Subsequently, we develop a synchronization-aware partitioned fixed-priority scheduler to accommodate these inefficiencies.

For the purposes of this chapter, each task τ_i still has a single computation segment $< \psi_i^1 >$ (i.e. $s_i = 1$), since we are concerned with *sequential* tasks. Each computation segment ψ_i^1 is of the form $[C_{i,1}^1, C_{i,1}'^1, C_{i,2}^1, C_{i,2}'^1, ..., C_{i,n_{i,1}-1}'^1, C_{i,n_{i,1}}^1]$ (where $n_{i,1} \geq 1$). For the purposes of this



chapter, we will simplify the notation to omit the single computation segment of $< \psi_i^1 >$ and directly represent the computation as $(C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,n_i-1}, C_{i,n_i})$ (without the superscript since there is only one computational segment in a sequential task). In this case, n_i simply denotes the number of normal executions blocks in the single sequential computational segment of task τ_i, while $(n_i - 1)$ is the number of critical section blocks. Due to the implicit-deadline nature, we represent each task τ_i as $\tau_i : ((C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,n_i-1}, C_{i,n_i}), T_i)$, where T_i is the task period. We still maintain the assumption of a single criticality domain.

In systems with task synchronization requirements, traditional synchronization-agnostic task allocation algorithms can introduce bottlenecks in the system by unnecessarily distributing tasks sharing global resources across different processors. Synchronization-agnostic scheduling can also lead to performance penalties by unnecessarily preempting tasks holding global resources. Therefore, coordination among task scheduling, allocation and synchronization is vital for maximizing the performance benefits of real-time multiprocessor systems.

The major contributions of this chapter are as follows:

1. Characterization of key synchronization penalties including (i) blocking delays on global critical sections, (ii) back-to-back execution from blocking jitter, and (iii) multiple priority inversions due to remote resource sharing between tasks allocated to different processors,
2. Development and evaluation of a synchronization-aware task-allocation scheme to accommodate these task synchronization penalties during the allocation phase,
3. Analysis of the impact of different execution control policies (ECPs) on global task synchronization, where an ECP is a mechanism to compensate for the scheduling penalties incurred by tasks due to remote blocking, and
4. Detailed empirical study of the above-mentioned execution control policies.

Our empirical results indicate that coordinated scheduling, allocation, and synchronization yields significant benefits (as much as 50% savings compared to synchronization-agnostic task allocation).

The rest of this chapter is organized as follows: Section 4.1 details the challenges associated with multiprocessor synchronization. We also review the multiprocessor priority ceiling protocol and describe the scheduling inefficiencies involved in remote task synchronization. In Section 4.2, we present our synchronization-aware task allocation algorithm and the different ECPs. Subsequently, we also provide a detailed analysis of these ECPs. Finally, a detailed evaluation of the synchronization-aware task allocation algorithm and the performance of different ECPs is provided in Section 4.3.

4.1 Multiprocessor Synchronization Challenges

In this section, we detail the various challenges associated with synchronization in multiprocessors. We first highlight the key differences between global and local task synchronization.

4.1.1 Global vs Local Synchronization

The Priority Ceiling Protocol (PCP) [79] is a real-time synchronization protocol that minimizes the time a high-priority task waits for a low priority one to release the lock on a shared resource, known as blocking time. When PCP is used by tasks deployed on different processors, this blocking time can lead to idling of the processors. For instance, consider Figure 4.1. In this figure, there are three tasks, τ_1 and τ_2 running in processor P_1 and τ_3 running in processor P_2. In addition, a resource is shared between tasks τ_2 and τ_3 using PCP. The figure depicts how τ_2 locks the resource at time 9 making τ_3 wait for the lock up to time 51 when the lock is released by τ_2. This waiting leaves processor P_2 idle because the only task deployed there, τ_3, is waiting for the lock (known as remote blocking). Furthermore, during the time τ_2 holds the lock it suffers multiple preemptions from the higher priority task τ_1. As a consequence, τ_3 misses its deadline at time 68. Such a problem is removed if, instead of sharing the resource across processors, it is shared on the same processor, i.e., tasks τ_2 and τ_3 are deployed together, say in processor P_2.

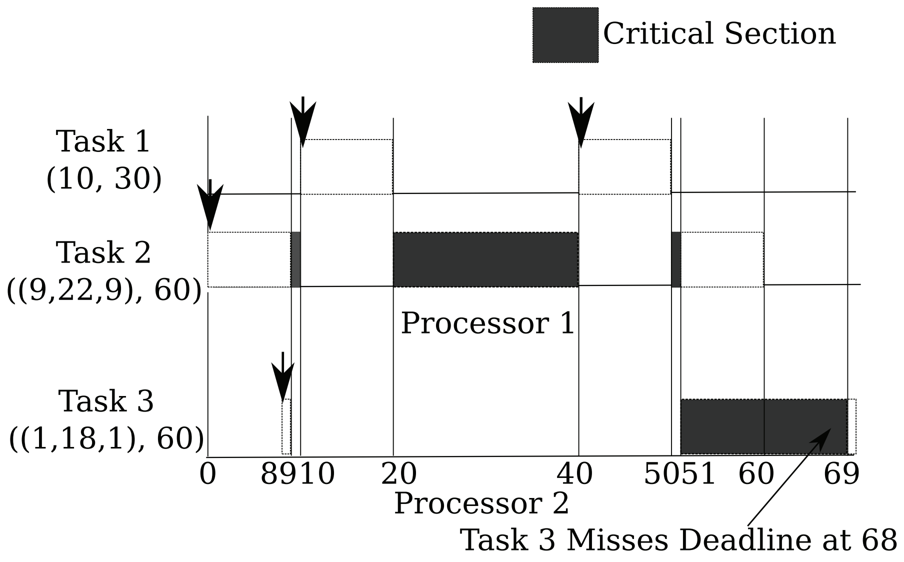


Figure 4.1: Penalty of global synchronization

The key aspect of the example in Figure 4.1 is that processor utilization is wasted during remote blocking. This is because a task that could be scheduled in the remote processor is blocked leaving the cycles reserved for it idle. This contrasts with local blocking because the task holding the lock uses the cycles the blocked task leaves idle. Furthermore, such a waste of reserved cycles can be repeated for each blocked task running on a different processor. That is, sharing a resource across n processors can waste reserved cycles in $n-1$ processors, effectively transforming this n processors into a single processor during the execution of the critical section (only one critical section can execute at a time). This highlights the significance of task allocation in determining the schedulability of a task set in a multiprocessor. In other words, the co-location of tasks that lock shared resources to the same processor prevents reserving processors cycles that are wasted in remote blocking. This motivates our synchronization-aware task-allocation algorithm.

In the worst case, however, some degree of global resource sharing may be unavoidable. As a result, techniques to mitigate its consequences are also needed.

4.1.2 Multiprocessor Priority Ceiling Protocol

We shall analyze and characterize the behavior of different execution control policies (ECPs) in Section 4.2. For the sake of self-containment, we present a brief tutorial of the multiprocessor priority ceiling protocol (MPCP) and its properties. Let us start by reviewing some definitions. A *global mutex* is a mutex shared by tasks deployed on different processing cores. The corresponding critical sections are referred to as *global critical sections* (gcs). Conversely, a *local mutex* is only shared between tasks on the same processing core, and the corresponding critical sections are *local critical sections*.

Let J' be the highest priority job that can lock a global mutex M_G. Under MPCP, when any job J acquires M_G, it will execute the gcs corresponding to M_G at a priority of $\pi_G + \pi'$, where π_G is a base priority level greater than that of any other normally executing task in the system, and π' is the priority of J'. This priority ceiling is referred to as the `remote priority ceiling` of a gcs.

MPCP was specifically developed for minimizing remote blocking and priority inversions when global resources are shared. We reproduce below a basic definition of MPCP. Readers should refer to [105] for a more detailed discussion.

MPCP Definition

1) Jobs use assigned priorities unless within critical sections.

2) The uniprocessor priority ceiling protocol [79] is used for all requests to local mutexes.

3) A job J within a global critical section (gcs) guarded by a global mutex M_G has the priority of its gcs ($\pi_G + \pi'$).

4) A job J within a gcs can preempt another job J^* within a gcs if the priority of J's gcs is greater than that of J^*'s gcs.

5) When a job J requests a global mutex M_G. M_G can be granted to J by means of an atomic transaction on shared memory, if M_G is not held by another job.

6) If a request for a global mutex M_G cannot be granted, the job J is added to a prioritized queue

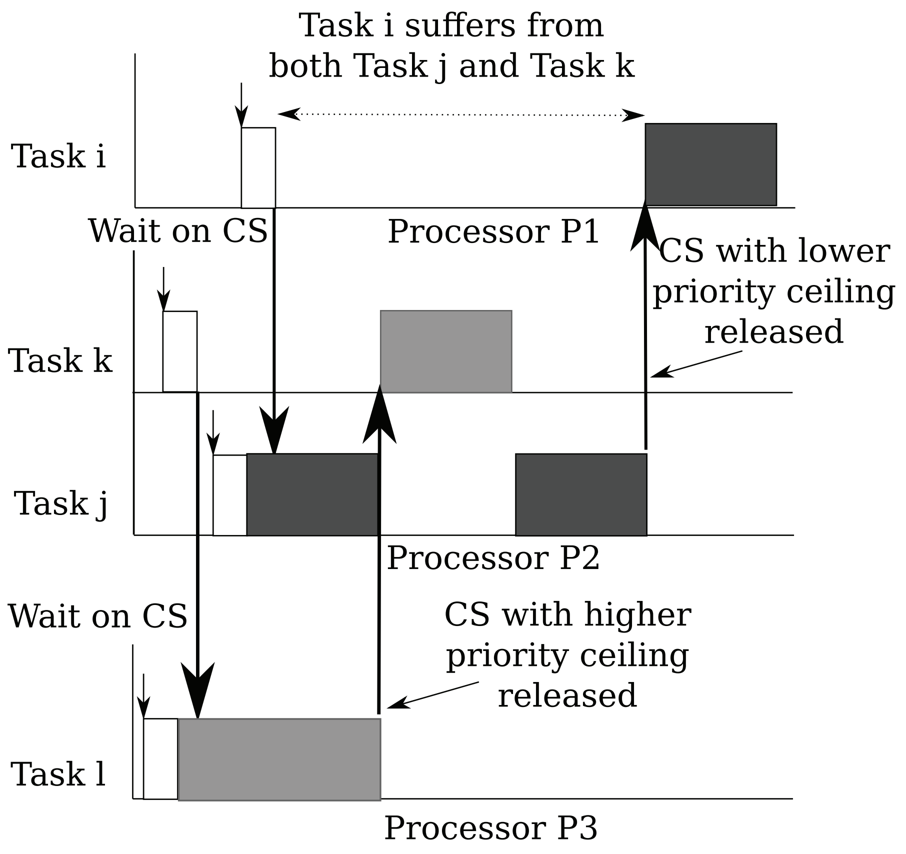

Task i suffers from both Task j and Task k
Task i
Wait on CS
Processor P1
CS with lower priority ceiling released
Task k
Task j
Processor P2
Wait on CS
CS with higher priority ceiling released
Task l
Processor P3

Figure 4.2: Transitive Interference during remote blocking

on M_G before being preempted. The priority used as the key for queue insertion is the *normal* priority assigned to J.

7) When a job J attempts to release a global mutex M_G, the highest priority job J_H waiting for M_G is signaled and becomes eligible for execution at J_H's host processor at its gcs priority. If no jobs are suspended on M_G, it is released.

Now, consider tasks that must suspend themselves when waiting for global critical section resources. Many penalties are encountered and are described next.

4.1.3 Blocking Delay on Remote Resources

MPCP executes all the gcs's at a priority level above all normal execution segments and local critical sections. Even under the purview of such a priority ceiling protocol, it is not possible to guarantee that tasks do not receive transitive interference during remote blocking as shown in Fig. 4.2. In this figure, Task τ_i in P_1 could be suspended on τ_j in P_2. Task τ_k on P_2 could be suspended on another task τ_l on another processor P_3. The priority ceiling of the mutex shared between τ_k and τ_l could be higher than the priority-ceiling of the mutex shared between τ_i and τ_j. In this scenario, the release of the critical section by τ_l would give control to the task τ_k, which preempts the task τ_j executing its critical section. The task τ_i waiting on τ_j therefore suffers from the interference due to the release of a mutex by τ_l.

Back-To-Back Execution of Suspending Tasks

A phenomenon that arises when tasks suspend themselves is that of ”back-to-back execution” [104]. Consider the example shown in Fig. 4.3. There are three tasks $\tau_1 : ((2, 2, 0), 8), \tau_2 : (4, 8), \tau_3 : ((1, 2, 2), 64)$. Task τ_1 and τ_2 are assigned to processor P_1. Task τ_3 is assigned to processor P_2.

It is easy to verify that τ_1 and τ_3 are schedulable. However, an anomalous scheduling behavior happens with respect to task τ_2. It should be schedulable if τ_1 follows a periodic release behavior, since it expects at most one preemption from a task with its same period of 8. When τ_2 is released at time instant 3, however, it faces back-to-back execution due to the remote synchronization effect of τ_1 and this leads to τ_2 suffering the interference of a second arrival of τ_1 leading to a deadline miss. This back-to-back preemption arises due to the jitter in the blocking time of task τ_1 and its self-suspending behavior.

Multiple Priority Inversions due to Suspension

The key scheduling inefficiency resulting from the remote blocking behavior of tasks is that of multiple priority inversions due to lower-priority critical sections. For example, consider the

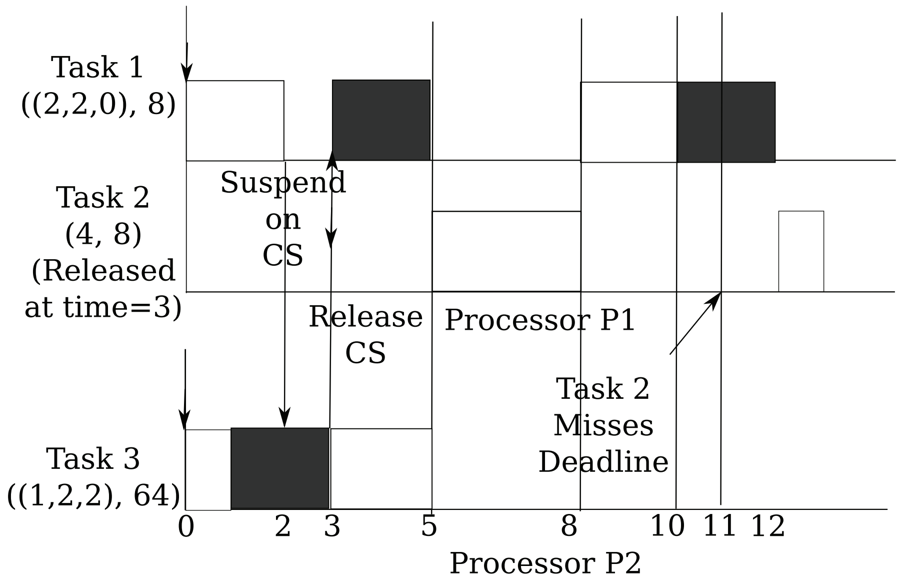


Figure 4.3: Back-To-Back Execution due to Remote Synchronization

scenario shown in Fig. 4.4. Whenever task τ_2 suspends, task τ_3 can get a chance to execute, and it can request a lock on the global critical section shared with τ_1. When τ_1 releases the global critical section, τ_3 preempts τ_2 due to its higher priority ceiling and interferes with the normal execution of τ_2 *twice*. In the worst case, *every* normal execution-segment (of duration $C_{i,k}$ $1 \leq k \leq n_i$) of a task τ_i can be preempted at most once by each of the lower-priority tasks τ_j $(j > i)$ executing their global critical sections released from remote processors.

4.2 Coordination Among Scheduling, Allocation, and Synchronization

Our goal is to develop a holistic scheme which combines a synchronization-aware task allocation strategy, with an efficient protocol for global task synchronization. In order to realize this, we first describe our synchronization-aware task allocation strategy. We then analyze different execution control policies under the multi-processor priority ceiling protocol.

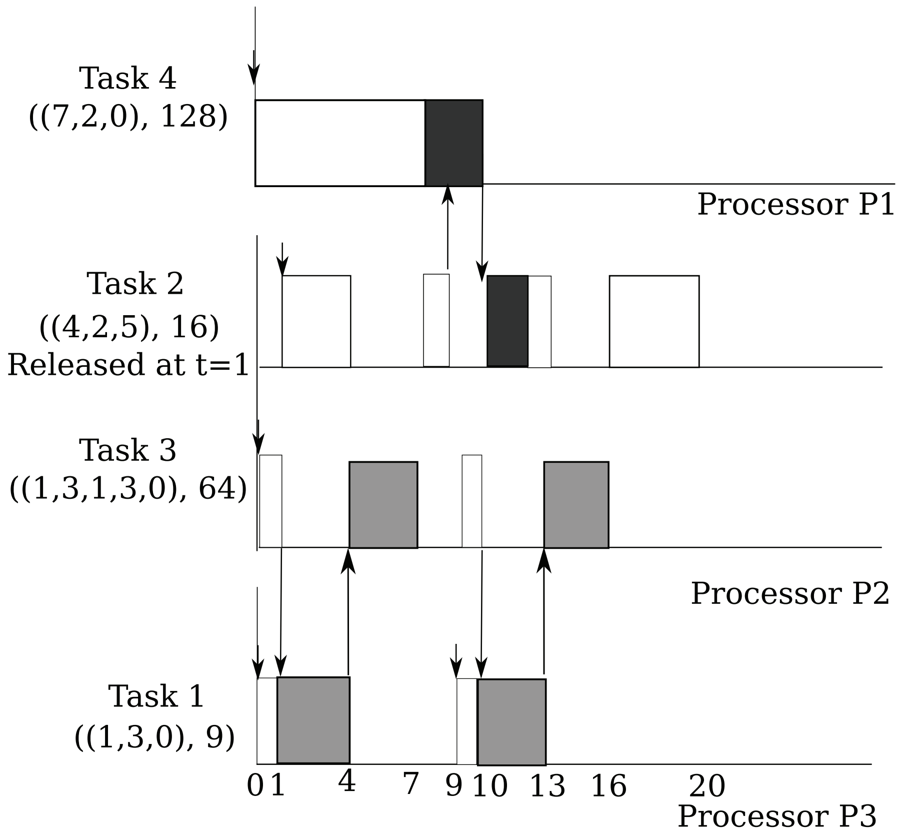


Figure 4.4: Multiple Priority Inversions Due to Suspension

4.2.1 Synchronization-Aware Task Allocation

We consider two bin-packing algorithms: the *synchronization-agnostic* and the *synchronization-aware* algorithms. The former packs objects exclusively based on size and the latter tries to pack together tasks that share mutexes. Both algorithms are modifications of the best-fit decreasing (BFD) bin-packing algorithm. The BFD algorithm orders the bins by non-increasing order of available space and the objects by non-increasing order of object size, and tries to allocate the object at the head of this sorted list into each of the bins in order.

When bin-packing algorithms are used to pack periodic tasks into processors, the utilization of each task is used as its size and one minus the total utilization deployed on a processor is used as the available space in the processors. This approach assumes that the load of the processors can reach 100%, which for rate-monotonic scheduling is only sometimes true. However, in the absence of additional information, such an approach is a good indicator of which task and which processor to try next. In our bin-packing algorithms, once we select the task to be allocated and the candidate processor for trying the allocation, we use our response-time tests to check if this allocation is possible. The fact that the allocation may not be feasible, even if the total utilization is less than 100%, signals the mismatch in assumptions between the packer algorithm and the admission test. Thus, additional inefficiencies are introduced in the packing phase.

When synchronization is used, an additional penalty can be incurred if we distribute the tasks that share a mutex among two or more processors. This is because, if we allocate these tasks to the same processor, the shared mutex becomes a local mutex and local PCP can be used. As described in the previous section, local synchronization eliminates the scheduling penalties associated with global task synchronization.

The strategy of the synchronization-aware packer is two-fold. First, tasks that share a mutex are *bundled* together. This bundling is transitive, i.e., if a task A shares a mutex with task B, and B shares a mutex with C, all three of them are bundled together. Then, each task bundle is attempted to be allocated together as a single task into a processor. We start with just enough processors to allocate the total utilization of all the tasks. Secondly, the task bundles that do not fit are put aside until all bundles and tasks that fit are allocated without adding processors. Now, only bundles that did not fit into any existing processor remain unallocated. The penalty of transforming a local mutex into a global mutex is the additional processor utilization required for schedulability. The cost of breaking a bundle is defined as the maximum of such penalties over all its mutexes. The bundles are then ordered in increasing order of cost, and the bundle with the smallest cost is selected to be broken. This bundle is broken such that it contains at least one piece as close as possible to the size of the largest available gap among the processors (in accordance with the BFD heuristic). If this allocation is not possible, a new processor is added and we try again to partition the task-set. Since the addition of new processors opens up new possibilities to allocate full bundles together, we repeat the whole strategy again starting by retrying to fit the unallocated bundles. In the absolute worst-case, each task may require its own processor, therefore, at most n processors exist in the final packing of any schedulable task-set (where n is the number of tasks).

4.2.2 Execution Control Policies

An execution control policy (ECP) is defined as a mechanism to compensate for the scheduling penalties incurred by tasks due to remote blocking. In this work, we consider the following execution control policies:

1. *Suspend:* The task is suspended during remote blocking, enabling lower priority tasks to execute.

2. *Spin:* The task continues to spin on the remote critical section, preventing lower priority tasks from executing.

Other execution control policies such as period enforcement [104] can also be applied for minimizing the scheduling penalty arising from synchronization, but these extensions are beyond the scope of this dissertation. We now describe different execution control policies and their schedulability implications.

MPCP:Suspend

The *MPCP:Suspend* execution control policy forces a task to suspend when it waits for a gcs entry request to be satisfied. In this version, tasks blocking on remote resources release the processor for other tasks executing in the system. It suffers from all the scheduling penalties described in the previous section. We now quantify each of the scheduling inefficiencies described in the earlier sections.

1) Remote Blocking due to Global Critical Sections: This is captured by a separate term $B^r_{i,j}$ for the j^{th} critical section acquired by the task τ_i (the r in $B^r_{i,j}$ denotes *remote* blocking as opposed to *local* blocking). We will compute this term later.

2) Back-To-Back Execution due to Suspending Tasks: In addition to the preemptions considered by conventional Rate-Monotonic Scheduling, in the worst case, back-to-back execution can result in additional interference from each higher priority task (τ_h).

This additional interference is upper bounded by the maximum interference from each higher priority task (τ_h) over a duration equal to its maximum duration of remote blocking ($B^r_h = \sum_{q=1}^{n_h-1} B^r_{h,q}$) (see [85] for details on using such suspension-based blocking terms in response-time tests).

This upper bound on the interference is captured in our analysis (see eqn.(4.1)) using the second term.

3) Multiple Priority Inversions due to Global Critical Sections: The global critical sections of each lower priority task can affect the normal execution segment of a higher priority task. This is captured by the per-segment interference given by $\sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < n_l} C'_{l,k}$.

Let us denote the total response-time of task τ_i without interference on its local processor as $C^*_i = C_i + B^r_i$. In such a scenario without interference, task τ_i is never preempted on the local processor requiring C_i units of computation time, and it can be remote blocked for at most B^r_i

units of time during lock acquisition for global critical sections, resulting in a total response-time of $C_i^* = C_i + B_i^r$.

The worst-case response time of task τ_i is bound by the convergence W_i of:

$$W_i^{n+1} = C_i^* + \sum_{h<i \,\&\, \tau_h \in P(\tau_i)} \lceil \frac{W_i^n + B_h^r}{T_h} \rceil C_h$$

$$+ n_i \sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < n_l} C'_{l,k} \quad (4.1)$$

where, $W_i^0 = C_i^*$

Let the global priority ceiling of the critical section $C'_{i,k}$ be denoted by $gc_{i,k}$.

The worst-case response time of the execution time of critical segment $C'_{i,k}$ is bounded by:

$$W'_{i,k} = C'_{i,k} + \sum_{\tau_u \in P(\tau_i)} \max_{1 \le v \le n_u \,\&\, gc_{u,v} > gc_{i,k}} C'_{u,v} \quad (4.2)$$

The reasoning here is that the global critical sections have an absolute priority greater than all the normal execution segments. Therefore, we only need to consider one outstanding global critical section request per task.

$B_{i,j}^r$ represents the remote blocking term for task τ_i in acquiring the global critical section $C'_{i,j}$. This is bounded by the convergence,

$$B_{i,j}^{r,n+1} = \max_{l>i \,\&\, (\tau'_{l,u}) \in R(\tau_i, j)} (W'_{l,u}) \quad (4.3)$$

$$+ \sum_{h<i \,\&\, (\tau'_{h,v}) \in R(\tau_i, j)} (\lceil \frac{B_{i,j}^{r,n}}{T_h} \rceil + 1)(W'_{h,v})$$

where, $B_{i,j}^{r,0} = \max_{l>i \,\&\, (\tau'_{l,u}) \in R(\tau_i, j)} (W'_{l,u})$

The key benefit of *MPCP:Suspend* is that the idle time during suspension is available for execution by other tasks. The disadvantages however are the penalties of back-to-back executions and multiple priority inversions per task.

An alternative approach is to prevent back-to-back execution and multiple priority inversions by not relinquishing the processor and spinning until the critical section is obtained [25], similar to *MPCP:Spin* defined next.

MPCP:Spin

In the *MPCP:Spin* protocol, tasks spin (i.e. execute a tight loop) while waiting for a gcs to be released. This avoids any future interference from global critical section requests from lower-priority tasks, which may be otherwise issued during task suspension. In practice, this could be implemented as *virtual* spinning, where other tasks are allowed to execute unless they try to access global critical sections. In that case, they would be suspended. As a result, the number of priority inversions per task is restricted to one per lower priority task. The back-to-back execution phenomenon is also avoided since the tasks do not suspend in the middle. The time spent waiting for the lock becomes part of the task execution time, therefore, the task never voluntarily suspends during its execution. This improves average-case performance but cannot guarantee worst-case improvements.

The analysis of the *MPCP:Spin* protocol is quite straight-forward. An upper bound on the computation time of task τ_i is given by: $C_i^* = C_i + B_i^r$

The worst-case computation time of task τ_i is bounded by the convergence W_i of:

$$\begin{aligned} W_i^{n+1} = C_i^* &+ \sum_{h<i\,\&\,\tau_h \in P(\tau_i)} \lceil \frac{W_i^n}{T_h} \rceil C_h^* \\ &+ \sum_{l>i\,\&\,\tau_l \in P(\tau_i)} \max_{1\leq k<n_l} C'_{l,k} \end{aligned} \qquad (4.4)$$

where, $W_i^0 = C_i^*$.

The blocking terms $B_{i,j}^r$ are the same as those given in Equation (4.3).

As can be seen, spinning reduces the preemptions from global critical sections of lower priority tasks since $\sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \leq k < n_l} C'_{l,k}$ is used in Equation (4.4) instead of $n_i \sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \leq k < n_l} C'_{l,k}$ used in Equation (4.1) for suspension. However, spinning results in additional preemption from higher priority tasks as captured by using the C^*_h term in Equation (4.4) instead of the C_h term used in Equation (4.1) for suspension. As a part of our empirical evaluation, we compare these two different execution control policies, and study the realm of applicability of each policy.

4.3 Evaluation

In this section, we present an experimental evaluation of the synchronization schemes and their integration with our synchronization-aware bin-packing algorithm. The metric used to compare the effectiveness of each algorithm is the number of bins needed to allocate a given taskset. The fewer the number of bins needed, the better is the performance. In order to study the performance of synchronization algorithms in isolation, we use the synchronization-agnostic packing algorithm. The benefits of using a synchronization-aware packing algorithm for each of these schemes is quantified later.

4.3.1 Experimental Setup

All our experiments evaluate how many processors of equal capacity (100% utilization) an algorithm uses to schedule a task set. We compare the number of processors needed among all the algorithms against the optimal packing algorithm. Given that optimal bin-packing is an intractable problem, we start with a fully-packed configuration instead (from a bin-packing standpoint disregarding scheduling inefficiencies). We do this processor by processor by dividing the 100% utilization into a defined number of tasks that would fit this processor perfectly. Each of these tasks is assigned a random utilization that all add up to 100%. Then, their periods are chosen randomly between 10ms and 100ms. Next, their execution time is calculated to match their utilization. Now, given a selected number of critical sections per task, the execution is divided into two types of segments: normal execution and critical section. These segments are arranged starting with a segment of normal execution followed by one of critical section and then another of normal execution. This arrangement continues until the task has the required number of critical sections. Each critical section is associated with a mutex that is locked by some chosen number of other tasks.

4.3.2 Comparison of Synchronization Schemes

We explore three main factors that affect the different ECPs: (i) the size of the critical sections, (ii) the number of tasks per processor, and (iii) the number of lockers per mutex.

In order to obtain a conceptual comparison of the different synchronization schemes, we consider their behavior under zero overheads. Overheads can change, and likely decline over time, and are platform-dependent. Later on, we explicitly specify and evaluate the impact of different preemption costs on these different schemes.

The most important factor that affects the different ECPs is the size of a critical section. Figure 4.5 depicts the results of our experiments with increasing critical section sizes. We create task sets of 8 fully-packed processors with 5 tasks per processor, 2 critical sections per task, and 2 lockers per mutex. Then, we pick critical section lengths from 5μs to 1280μs.

Initially, both *MPCP:Spin* and *MPCP:Suspend* exhibit similar performance. At longer critical section lengths however, *MPCP:Spin* requires many more processors compared to *MPCP:Suspend*. This is mainly due to the processor time lost during spinning on a mutex. In the case of *MPCP:Suspend*, this time is effectively used by the other tasks executing on the same processor. As a general trend, however, both *MPCP:Spin* and *MPCP:Suspend* require more processors with longer critical section lengths. This is mainly due to increasing remote blocking terms with increasing global critical section lengths. In the case of *MPCP:Suspend*, this overhead manifests as longer priority inversions from lower-priority global critical section executions. *MPCP:Spin* experiences a similar overhead due to the increased usage of CPU time during spinning.

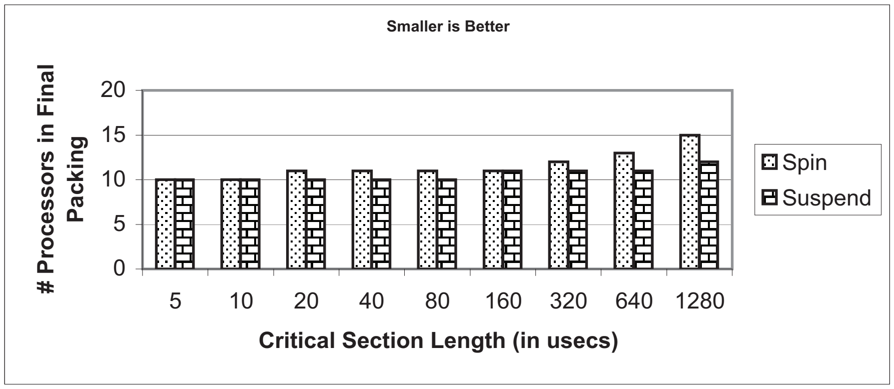


Figure 4.5: Efficiency of Algorithms as Length of Critical Section Increases (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, average of 5 tasks per processor in initial workload)

Next, we increased the number of tasks per processor, since this has the potential of increasing the number of preemptions for a task and also the number of priority inversions.

Figure 4.6 depicts the number of processors needed to pack a workload of eight fully-packed processors with two critical sections per task and two lockers per mutex. Each critical section has a duration of 500μs. We chose this value to observe meaningful differences between the two ECPs, based on the results of the earlier experiments with varying critical section lengths. Critical sections of similar durations have been observed in the past [69, 70], and hence considering such lengths is both relevant and useful. For smaller critical section lengths, the previous experiments already indicate that the performance difference is negligible, and this was verified.

In Figure 4.6, we can observe that, as the number of tasks per processor increases, the spin-based synchronization scheme requires more processors compared to the suspension-based scheme. This is because having more tasks during spinning to wait for global mutexes increases the loss of processor utilization. The suspension scheme, however, effectively utilizes this duration to execute other eligible tasks hosted on the same processor. In general, both the schemes require more processors with an increasing number of tasks. With *MPCP:Spin*, this is due to more tasks using CPU time for spinning, whereas with *MPCP:Suspend*, this is due to more priority inversions from lower priority tasks locking global mutexes.

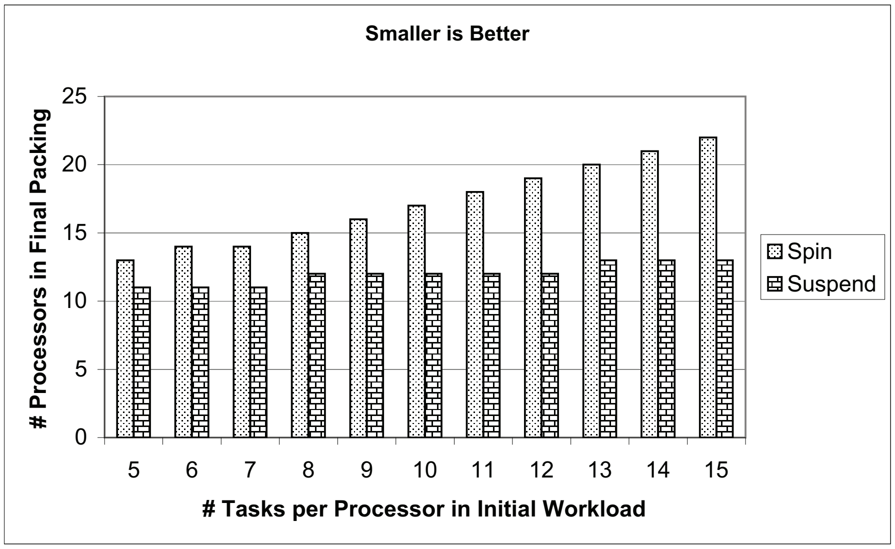


Figure 4.6: Efficiency of Algorithms as Tasks per Processor Increases (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

The third factor that affects different ECPs is the number of lockers per mutex. Figure 4.7 depicts the results of our experiments with a growing number of lockers. For this experiment, we created task sets of eight fully-packed processors with five tasks per processor, two critical sections per task, and a critical section of 100μs. In this figure, we can see that *MPCP:Spin* is adversely affected, when the number of lockers increases. This is due to the fact that the waiting time can be enlarged with the number of tasks that can potentially lock the mutex. Given that this spinning task does not relinquish the processor, it reduces the total useful utilization. Because *MPCP:Suspend* does not spin-wait for the mutexes, it does not suffer as much. The increase in blocking terms also results in a slight increase for suspension-based schemes. Using less pessimistic admission control tests could potentially improve this scenario for suspension.

Finally, we also explored the trend of increasing the presented workload of fully-packed processors. Figure 4.8 depicts this experiment. In this figure, we can see that *MPCP:Spin* and *MPCP:Suspend* exhibit similar behavior with increasing workloads. The major observations are: (1) *MPCP:Suspend* requires fewer processors than *MPCP:Spin*, and (ii) the ratio of the number of processors required by the two schemes remains fairly constant. This behavior is due to the

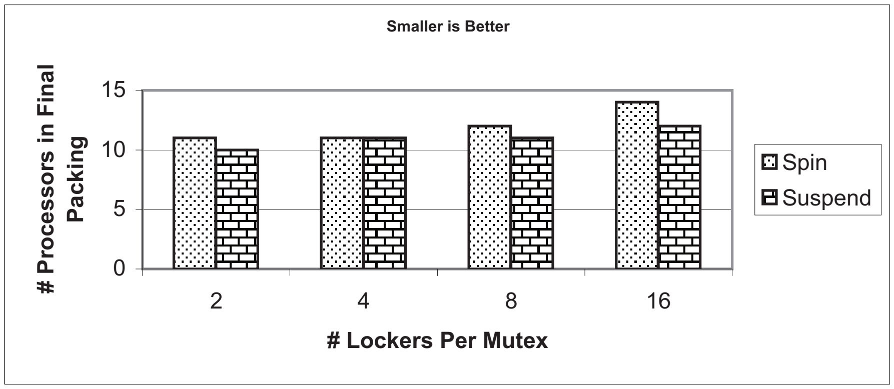


Figure 4.7: Efficiency of Algorithms as Lockers per Mutex Increases (workload: 8 fully-packed processors, 2 critical sections per task, critical section length 100μs, average of 5 tasks per processor in initial workload)

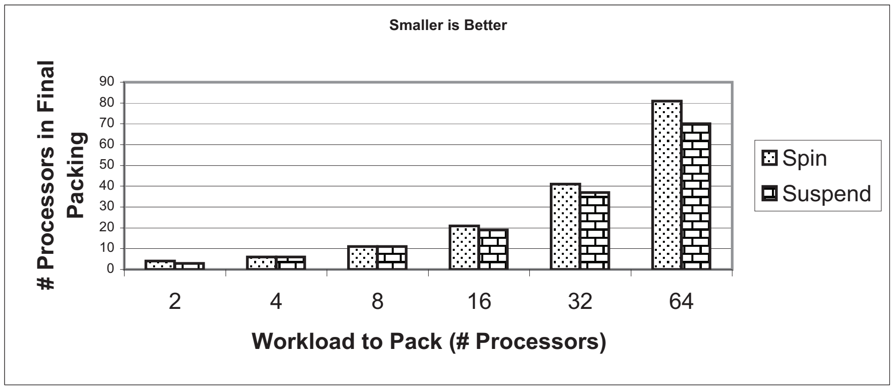


Figure 4.8: Efficiency of Algorithms as Workload Increases (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

fact that increasing the workload only affects the task allocation algorithm as opposed to the synchronization schemes themselves (as long as other parameters remain constant).

In summary, under zero implementation overheads; suspension-based MPCP performs better than spin-based MPCP. However, as we show later in Subsection 7.5.10, implementation overheads also play a significant role in the performance of synchronization protocols. For smaller critical section durations and larger implementation overheads, spin-based MPCP can be expected to perform better because it should have fewer context-switches.

4.3.3 Synchronization-Aware Task Allocation

Our synchronization-aware packing algorithm bundles together synchronizing tasks and tries to deploy them. This strategy reduces the number of global mutexes and resulting remote blocking. However, the bundling heuristic artificially creates larger objects to pack, and this can lead to less efficient packing than the BFD heuristic on the original taskset. As a result, our algorithm does not always pay off. However, when remote blocking penalties play a major role, then our synchronization-aware packer yields significant benefits.

Figures 4.9 and 4.10 depict the behavior with increasing number of tasks per processor. In these figures, we pack a workload of eight fully-packed processors, critical section length of 500μs, 2 critical sections per task, and 2 lockers per critical section. As seen in earlier experiments, both schemes exhibited similar performance characteristics at lower critical section lengths, and hence a critical section length of 500μs is considered. On the average, synchronization-aware bin packing saves about 6 processors with *MPCP:Spin*, whereas there is a reduction of about 2 processors with the suspension-based technique.The reason for the bigger savings under spinning is that there is more penalty associated with a global critical section than in a suspension-based scheme (for the critical section length under consideration). Hence, synchronization-aware packing to eliminate such penalties saves more processors in spin-based schemes. As the number of tasks increases, synchronization-aware packing results in up to 50% reduction in the required number of bins (under *MPCP:Spin* beyond 15 tasks per processor). We also investigated the impact of increasing the workload. We again observed much less savings with the suspension-based technique compared to spinning. This is due to the fact that there is more room for optimization under *MPCP:Spin*, compared to *MPCP:Suspend*. The flexibility gained by increasing the workload therefore tends to have an impact on *MPCP:Spin*, whereas there is less benefit for *MPCP:Suspend*.

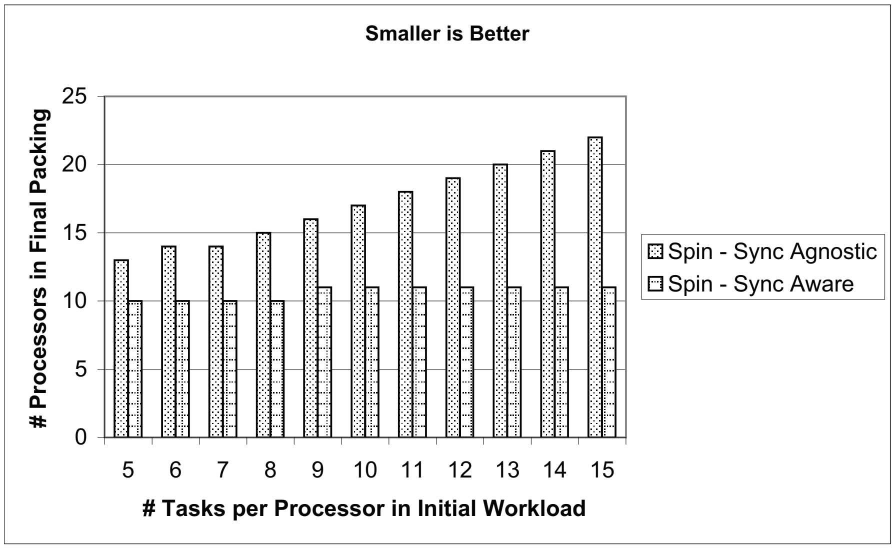


Figure 4.9: Efficiency of Sync-Aware Packing with Number of Tasks Per Processor in Initial Workload (Spin) (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

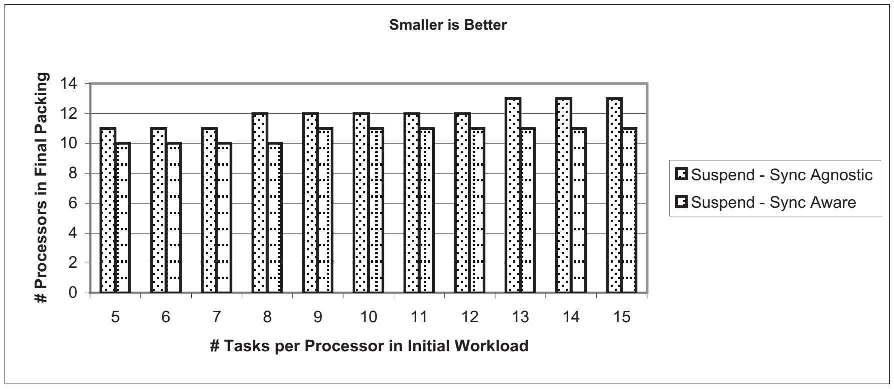


Figure 4.10: Efficiency of Sync-Aware Packing with Number of Tasks Per Processor in Initial Workload (Suspend) (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

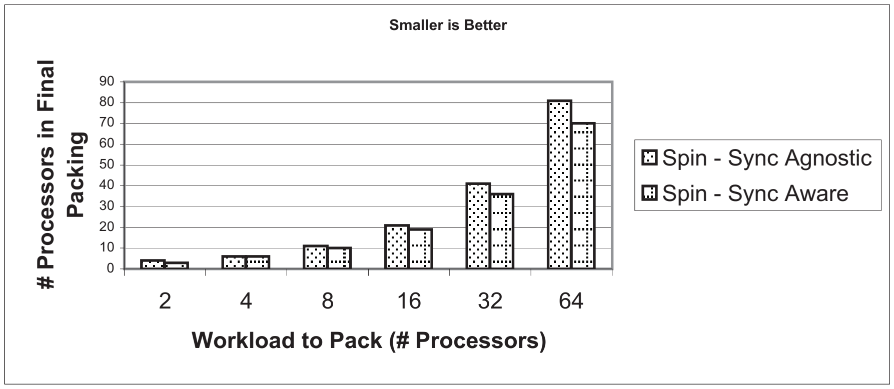


Figure 4.11: Efficiency of Sync-Aware Packing as Workload Increases (Spin) (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

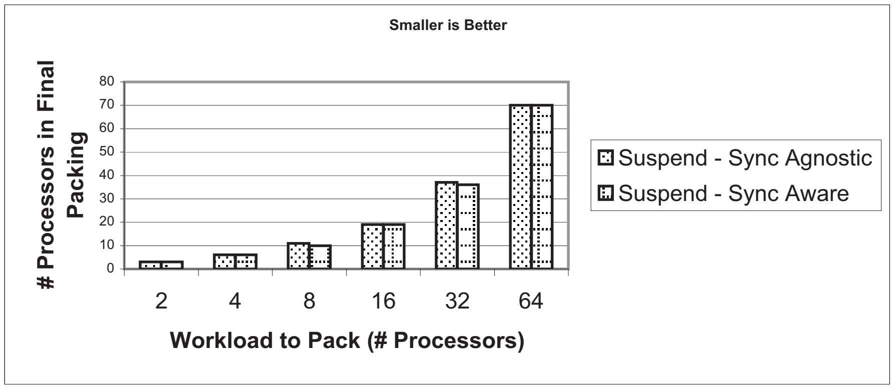


Figure 4.12: Efficiency of Sync-Aware Packing as Workload Increases (Suspend) (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

4.3.4 Synchronization-Aware Task Allocation and Splitting

The approach thus far has been to perform synchronization-aware task allocation and scheduling for avoiding the penalty of inter-core task synchronization. As described earlier, the bin-packing bounds still apply when purely partitioned approaches are considered. Even though object splitting enables us to split *composite tasks* into its constituent subsets, the fragmentation penalty from partitioning could still remain. In systems, where the task set is still not schedulable after the synchronization-aware task allocation, the next step would be to attempt task splitting. Our primary goal in this chapter has been to compare the benefits of using synchronization information during the allocation phase, hence we have not provided a detailed empirical evaluation of this extension of using task splitting to recover additional utilization.

From the bin-packing perspective, we will first allocate the composite tasks that are schedulable under standard synchronization-aware allocation. We then allocate the remaining tasks using the task splitting approach since the system is otherwise unschedulable under the traditional bin-packing approach. In order to attempt this task splitting, we should note that the remaining tasks might not necessarily have the highest priority on the allocated processor. The approach to take here is to assign the split task a deadline equal to the deadline of the currently existing highest-priority task on the host processor.

For example, consider a task τ with a computational requirement of 3 and deadline of 9 that needs to be split into τ' and τ''. In this scenario, τ' is created such that some processor P_i is maximally utilized when τ' is added. Let the highest-priority task on P_i have a deadline of 4. Let us say that assigning a deadline of 4 to τ' results in a maximum possible computational time of 1 for τ' on P_i. The remaining task τ'' will now have a computational time of 2 and a deadline of 8, since τ' can be assigned the highest priority on P_i. By performing such a splitting, the remaining object τ'' has a size of 25% instead of the original object τ with a size of 33.33%. Task splitting can thus be used an additional tool after the synchronization-aware task allocation is completed, for scheduling the remaining tasks that are otherwise unschedulable. Further evaluating these benefits in randomly generated task sets and quantifying the performance benefit is part of our key future work. It should be noted that for the split tasks the critical sections will automatically become *global* critical sections since they could be accessed from any of the allocated processors for the task. Also, the migration and preemption cost need to be included as a part of the critical section execution time, since tasks might migrate while holding the corresponding mutex.

4.4 Summary

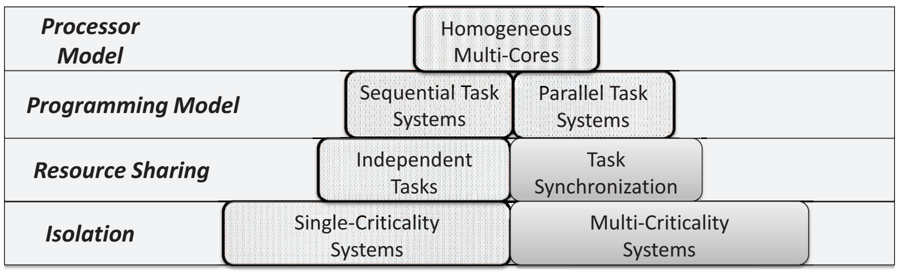
In this chapter, we studied the problem of scheduling *synchronizing sequential* tasks on multi-core processors. We developed a coordinated approach for dealing with task synchronization on chip-multiprocessors. We identified the major scheduling inefficiencies associated with remote task synchronization in fixed-priority multiprocessor scheduling. In order to minimize these scheduling inefficiencies, we analyzed and evaluated different execution control policies. Subsequently, we developed a synchronization-aware task allocation algorithm to explicitly consider these penalties. In line with the bin-packing problem, we consider *sets of synchronizing tasks* as a single *object* and try to allocate such objects on processor cores. When we cannot allocate all the objects to processor cores, we adopt the *object splitting* approach and split the object with the minimum remote synchronization penalty. Experimental evaluation suggests that significant benefits can be achieved using coordinated approaches to task scheduling, allocation and synchronization (in some cases up to 50% reduction in the required number of processing cores). Choosing the appropriate execution control policy (*MPCP:Suspend* or *MPCP:Spin*) based on the system characteristics reduces the penalty due to remote blocking. Synchronization-aware task allocation further reduces the penalty from global resource sharing.

Chapter 5

Parallel Real-Time Tasks

We have thus far focused only on the largely familiar *sequential* task model, where tasks only have a single thread of execution. In a real-time system adopting multi-core processors, the use of a sequential programming model is still a significant handicap to efficiently using the available parallelism. Many industry vendors in the general-purpose computing domain are already considering parallel programming models such as Message-Passing Interface (MPI), OpenMP, and Thread-Building Blocks (TBBs) to better utilize multi-core processors. In this chapter, we generalize our model to include the scheduling of parallel real-time tasks and provide early solutions in this emerging topic. We restrict our attention to independent tasks for the purposes of this dissertation, with the only inter-dependence being the precedence relationships between the sequential and parallel execution segments of the same task.

For *independent* tasks, each task τ_i has multiple computation segments $< \psi_i^1, \psi_i^2, ..., \psi_i^{s_i} >$



(where $s_i \geq 1$). However, each computation segment ψ_i^1 is of the form $[C_{i,1}^1]$ (i.e. $n_{i,j} = 1$) due to the assumption of *independent* tasks. The computational segments alternate between sequential and parallel sections (see Figure 1.2). Therefore, for clarity of notation, we will represent the computation of the j^{th} sequential segment as C_i^j (where j is an odd number) and the computation of the k^{th} parallel segment as P_i^k (where k is an even number). Note that we omit the second subscript due to the assumption of independent tasks, which results in a single contiguous execution block within each computational segment. The number of parallel threads forked by each parallel segment P_i^k is represented as m_i. Each task τ_i in the system can therefore be represented as $\tau_i : ((C_i^1, P_i^2, C_i^3, P_i^4, ..., P_i^{s_i-1}, C_i^{s_i}), m_i, T_i)$, where T_i is the task period.

The main contributions of this chapter are:

1. the best-case and worst-case basic fork-join task sets from a feasibility perspective,
2. the task *stretch* transform to reduce the scheduling penalty of basic fork-join structures, and
3. a resource augmentation bound of 3.42 for a task partitioning algorithm with deadline-monotonic scheduling.

The rest of this chapter is organized as follows. For parallel real-time fork-join task sets, we show the theoretical best-case and worst-case task characteristics in Section 5.2. Based on our observations of these key task structures, we develop a task transform in Section 5.3. Finally, we describe a partitioned preemptive fixed-priority scheduling approach for scheduling *fork-join* task sets in Section 5.4 and provide key resource augmentation results for the same.

5.1 Properties

For convenience, let us define

$$P_i = \sum_{s=1}^{\frac{s_i-1}{2}} (P_i^{2s})$$

as the minimum parallel segment execution time.

The ratio of the *maximum execution length* to *minimum execution length* is defined as the *parallelism speedup factor* Υ_i possible for task τ_i under maximal parallelism

$$\Upsilon_i = \frac{C_i}{\eta_i}$$

Based on these definitions, the following properties hold:

- For a task τ_i that cannot be parallelized, $\Upsilon_i = 1$.
- For a task that can be *fully* parallelized on m_i cores i.e. $s_i = 3$, $C_i^1 = C_i^3 = 0$, and $P_i^2 = \frac{C_i}{m_i}$, $\Upsilon_i = m_i$.

Proposition 1. *For a given processor speed, if the minimum execution length η_i of any task τ_i is greater than its period (implicit deadline) T_i, then τ_i is not schedulable on any number of processors with the same speed.*

Proof. The proof trivially follows from the definition of *minimum execution length*. η_i is the response time of τ_i when it is assigned m_i processor cores *exclusively*. Adding additional processor cores does not result in any additional speed up for τ_i since it never spawns more than m_i parallel execution threads in any of its parallel segments. Therefore, if jobs of τ_i cannot meet their implicit deadline on m_i processors, then they cannot meet their deadlines with any number of additional processor cores. □

A task set τ in this task model is represented as $\tau \colon \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i follows the *fork-join* structure. Without loss of generality, we assume that these tasks are sorted in non-decreasing order of task periods. The implicit-deadline nature of these tasks also means that these tasks are also sorted in non-decreasing order of relative deadlines.

Using the *fork-join* parallel real-time task model, we next characterize the best-case and worst-case task set structures for multiprocessor systems.

5.2 Worst-Case and Best-Case Task sets in the Fork-Join Model

Consider a processor with m processor cores. From a feasibility analysis perspective, we are interested in (i) the task set with *maximum* processor utilization that is *feasible* on the given processor with m cores, and (ii) the task set with *minimum* processor utilization that is *infeasible* on the given processor with m cores. In this section, we develop both task sets, and identify the characteristics for developing scheduling algorithms for fork-join task sets.

5.2.1 Best-Case Fork-Join Task Structure

The theoretical best-case structure for fork-join tasks to be scheduled on m processor cores happens when each task τ_i in the task set τ has the structure of $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$. The utilization of each task τ_i is $\frac{C_i}{T_i}$. We will now proceed to show that if any task set τ comprises solely of tasks with type $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$, then τ is feasible as long as the total utilization of τ does not exceed m.

Proposition 2. *When each task τ_i in a fork-join task set τ has the structure $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$ with an implicit deadline of T_i, then the task set τ is feasible on m processor cores as long as the cumulative utilization does not exceed m.*

Proof. The proof follows by showing that there exists a scheduling algorithm which guarantees that jobs of each task τ_i in the given fork-join task set τ will meet their deadlines.

Under the task structure $\tau_i : ((0, C_i/m, 0), m, T_i)$, each fork-join task is composed of exactly m parallel threads that execute for C_i/m time units each. Consider a Global EDF schedule of τ, where at any time instant t, all m processor cores will execute parallel threads from the same task. Given that each task is composed of exactly m parallel threads of equal execution requirements that are released simultaneously, the schedules on all the m processor cores are identical. Therefore, for the given task set to be feasible under Global EDF, it is sufficient that the threads scheduled on each individual processor core schedulable under EDF. From the schedulability condition proposed in [74], it follows that τ is schedulable as long as each individual processor core's utilization does not exceed 1 or the total utilization of τ does not exceed m. $\square$

Corollary 1. *When each task τ_i in a fork-join task set τ has the structure $\tau_i : ((0, \frac{C_i}{m}, 0), m, T_i)$ with an implicit deadline of T_i, then the task set τ is schedulable under Global RMS on m processor cores as long as the cumulative utilization does not exceed $m \ln 2$.*

Proof. Using the argument from Proposition 2 that each task in τ is composed of exactly m parallel threads of equal execution requirements that are released simultaneously, the schedules on all the m processor cores are identical. It is therefore sufficient that the threads on each individual processor core are schedulable under uniprocessor RMS. From the schedulability condition in [74], it follows that τ is schedulable as long as each individual processor core's utilization does not exceed $\ln 2$ or the total utilization of τ does not exceed $m \ln 2$. $\square$

It should be noted here that Proposition 2 is closely related to the result proved in [56], which shows that EDF with all jobs parallelized on all processors is optimal. Proposition 2 derives this result in the context of the fork-join task model.

We now show a worst-case task set that has the minimum total utilization among all infeasible fork-join task sets.

5.2.2 Theoretical Worst-Case Fork-Join Taskset

The theoretical worst case for fork-join task sets from a schedulability perspective is shown in Figure 5.1. This scenario comprises an infeasible fork-join task set with the least possible cumulative utilization among all infeasible fork-join task sets. In this scenario, there are two tasks τ_i and τ_j. Task τ_i has the structure $\tau_i : ((0, 1, F), m, 1 + F)$, and τ_j has the structure $\tau_j : ((\epsilon), 1, 1)$. For $F >> m$ and arbitrarily small $\epsilon > 0$, the utilization U of this task set is:

$$U = \frac{m + F}{1 + F} + \frac{\epsilon}{1} = 1 + \epsilon + \frac{m - 1}{1 + F} \approx 1 \tag{5.1}$$

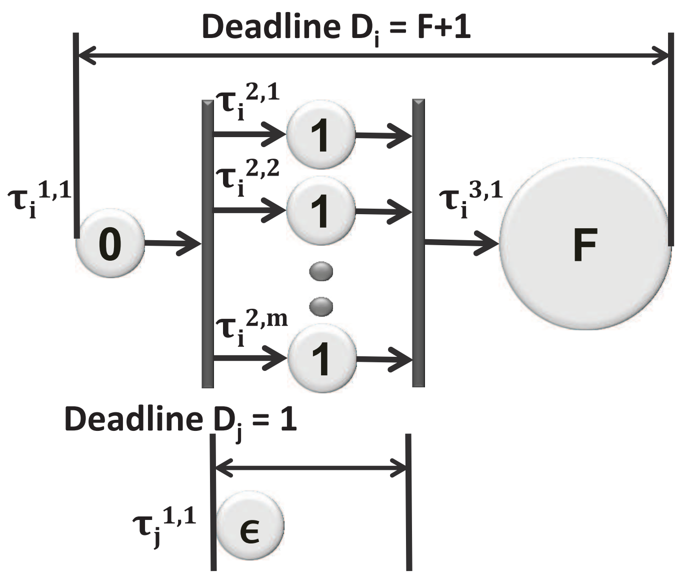


Figure 5.1: Worst-Case Fork-Join Task Set.

The utilization U is slightly greater than 1.

Proposition 3. *The fork-join task set comprising of two tasks* $\tau_i : ((0, 1, F), m, 1 + F)$ *and* $\tau_j : ((\epsilon), 1, 1)$ *is unschedulable on a system with no greater than* m *processing cores.*

Proof. The infeasibility follows from the critical instant when both τ_i and τ_j are released together at time 0. At this instant, τ_i will have m parallel threads with unit execution requirements that are ready to be scheduled, while τ_j will have 1 thread with ϵ execution requirement that is ready. There is at most a total of m available processor cores, therefore over the time interval $[0, 1]$ no more than m time units of execution can be completed causing either τ_j to miss its implicit deadline of 1 or one of the parallel threads of τ_i to face a preemption of ϵ. If any parallel thread of τ_i faces a preemption of ϵ from τ_j, then it would cause τ_i to miss its deadline since the *minimum execution length* of τ_i is equal to its period (implicit deadline) of $(1 + F)$ and an additional preemption of ϵ would cause it to miss its deadline by ϵ. Therefore, the task set comprising of tasks τ_i and τ_j is unschedulable. □

Lemma 5. *For* $F >> m$ *and arbitrarily small* $\epsilon > 0$*, the fork-join task set comprising of two* *tasks* $\tau_i : ((0, 1, F), m, 1 + F)$ *and* $\tau_j : ((\epsilon), 1, 1)$ *is an infeasible task set with the least possible utilization among all infeasible task sets.*

Proof. The proof is obtained by contradiction. For large $F >> m$ and small $\epsilon > 0$, the utilization of the task set with tasks $\tau_i : ((0, 1, F), m, 1 + F)$ and $\tau_j : ((\epsilon), 1, 1)$ is arbitrarily close to 1 as shown in Equation (5.1). From Proposition 3, it also follows that this task set with τ_i and τ_j is unschedulable on m processor cores. Suppose there exists another task set τ that has a lower utilization $U_\tau \leq 1$, and τ is infeasible on m processors. Consider a transform of task set τ to τ', where each task τ_i in τ is considered as a conventional sequential task $\tau'_i : ((C_i), 1, T_i)$ in task set τ'. Taskset τ' is schedulable under the Earliest-Deadline First (EDF) scheduling algorithm [74] on a uniprocessor. Therefore, the original task set τ is feasible by scheduling the fork-join tasks as sequential tasks on one processor core using EDF. This results in a contradiction that the fork-join task set τ is infeasible. $\square$

Analyzing the worst-case fork-join task set shown in Figure 5.1, note that the minimum execution length $C_i = (1 + F)$ of τ_i is equal to its period of $T_i = (1 + F)$. This translates to allowing no slack for any execution segment of τ_i. The parallel execution segments of τ_i therefore have an execution requirement of 1, and an inherited deadline of 1 since it would cause τ_i to miss its deadline otherwise. Therefore, these parallel execution segments can be considered as *subtasks* with an execution requirement of 1, period of $(1+F)$, and a constrained deadline of 1. Therefore, no additional task with a period (and deadline) less than or equal to 1 is schedulable on the m cores since the execution of τ_i has zero available slack. In this scenario, a single processor has a total utilization of 1, while the remaining $(m - 1)$ processors have a utilization of $\frac{1}{1+F}$ each. For arbitrarily large F, this task set demonstrates that the m processor utilization bound of fork-join task sets reaches only the single processor utilization bound of 1 for EDF.

The following are our key observations from the theoretical best-case and worst-case task sets:

(1) Uniformly parallelized tasks with a *parallelism speedup factor* Υ of m provide the most

benefit from a schedulable utilization perspective (from Proposition 2). Transforming the given *fork-join* tasks to resemble the best-case task structure might prove useful to improve schedulable utilization.

(2) There exist task sets with a total utilization slightly greater than and arbitrarily close to 1 that are unschedulable on a system with m processor cores (from Lemma 5). Therefore, conventional *utilization bounds* such as those employed in [74] may not be useful in characterizing the performance of scheduling algorithms for *fork-join* task sets. Resource augmentation bounds such as those presented in [50] seem to be promising candidates for the performance analysis of scheduling algorithms in the context of *implicit-deadline fork-join* task sets. Based on these observations, we now define a task transform that reduces the problem of scheduling *fork-join* task sets on multiprocessors to the problem of scheduling *constrained-deadline* task sets on multiprocessors. We then develop key resource augmentation bounds for analyzing the schedulability of such task sets.

5.3 Task Transformation

Fork-join task sets on multiprocessor systems can have schedulable utilization bounds slightly greater than and arbitrarily close to uniprocessor schedulable utilization bounds. From the perspective of schedulability, it is therefore desirable to avoid such task structures as much as possible. In this section, we propose a fork-join task *stretch* transform that avoids fork-join structures when possible. Based on this task transform, we will develop a partitioned preemptive fixed-priority scheduling algorithm for fork-join task sets.

We first illustrate the task *stretch* transform with an example fork-join task set having two tasks $\tau_1 : ((2, 6, 2), 4, 15)$ and $\tau_2 : ((15), 1, 20)$ to be scheduled on 4 processors. Under global scheduling (see Figure 5.3), τ_2 misses its deadline since threads of τ_1 have a higher priority (under both EDF and DM). Under partitioned scheduling based on First-Fit Decreasing (FFD), the deadline miss for τ_2 may be avoided by allocating it exclusively to a processor. However, such

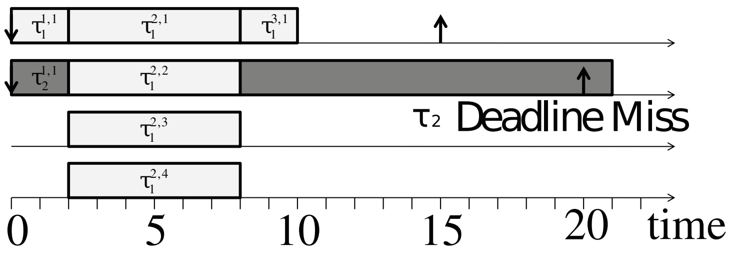


Figure 5.2: Unschedulable under traditional global Scheduling. Scheduling a task set τ_1 : $((2, 6, 2), 4, 15)$ and τ_2 : $((15), 1, 20)$ on 4 cores.

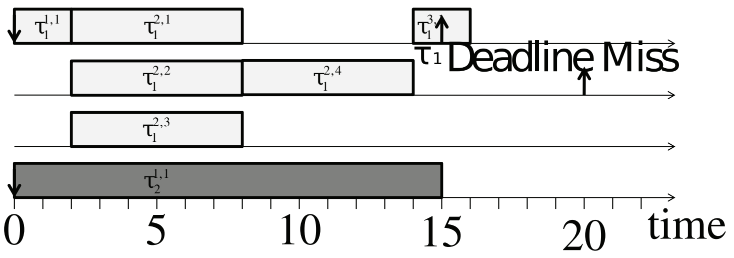


Figure 5.3: Unschedulable under partitioned Scheduling (with FFD). Scheduling a task set τ_1 : $((2, 6, 2), 4, 15)$ and τ_2 : $((15), 1, 20)$ on 4 cores.

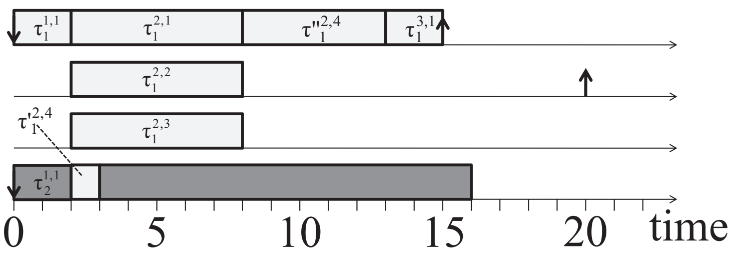
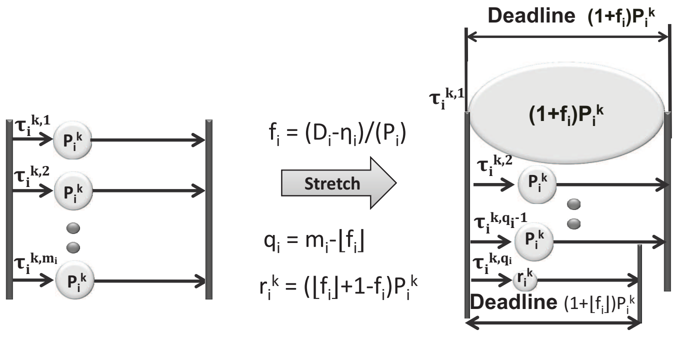


Figure 5.4: Schedulable under *stretch* Transform. Scheduling a task set τ_1 : $((2, 6, 2), 4, 15)$ and τ_2 : $((15), 1, 20)$ on 4 cores.



$\tau_i^{k,1}$
P_i^k
$\tau_i^{k,2}$
τ_i^{k,m_i}
$f_i = (D_i-\eta_i)/(P_i)$
Stretch
$q_i = m_i-\lfloor f_i \rfloor$
$r_i^k = (\lfloor f_i \rfloor+1-f_i)P_i^k$
Deadline $(1+f_i)P_i^k$
$(1+f_i)P_i^k$
τ_i^{k,q_i-1}
τ_i^{k,q_i}
r_i^k
Deadline $(1+\lfloor f_i \rfloor)P_i^k$

Figure 5.5: Task *Stretch* Transformation.

an allocation causes τ_1 to miss its deadline instead, as shown in Figure 5.3. Now consider an alternative allocation, where the master thread $(\tau_1^{1,1}, \tau_1^{2,1}, \tau_1^{3,1})$ of task τ_1 is allocated to processor 1, requiring 10 time units of execution every 15 time units. Therefore, τ_1 will have an available slack of 5 time units on processor 1. The parallel segments $\tau_1^{2,2}$ and $\tau_1^{2,3}$ are allocated to processors 2 and 3 respectively. In order to consume the 5 time units of available slack on processor 1, we also assign 1 time unit of parallel segment $\tau_1^{2,4}$ to processor 4, while the remaining 5 time units are sequentially executed after $\tau_1^{2,1}$ on processor 1. This *stretching* of task τ_1 ensures that task τ_2 can be accommodated on processor 4. Thus, the two tasks can be scheduled without missing any deadlines (see Figure 5.3).

In the task stretch transformation (Figure 5.6, each newly created constrained deadline parallel thread is represented as $\tau : (C, D)$, with C as the worst-case execution time, and D representing the deadline. When appending to an existing thread, we use $\tau : (C)$ to represent the execution time of the subtask appended.

We now formally develop the task *stretch* transform for scheduling basic fork-join task sets. Let us consider a fork-join task $\tau_i : ((C_i^1, P_i^2, C_i^3, P_i^4, ..., P_i^{s_i-1}, C_i^{s_i}), m_i, T_i)$. Task τ_i can be alternatively represented as the sequence of subtasks $\tau_i : (\tau_i^{1,1}, (\tau_i^{2,1}, ..., \tau_i^{2,m_i}), \tau_i^{3,1}, (\tau_i^{4,1}, ..., \tau_i^{4,m_i}), ..., \tau_i^{s_i,1})$. The *master string* of a non-stretched task τ_i is defined as the *thread sequence* $(\tau_i^{1,1} \rightarrow \tau_i^{2,1} \rightarrow ..., \tau_i^{s,1} \rightarrow ... \rightarrow \tau_i^{s_i,1})$ comprising of exactly one thread $\tau_i^{s,1}$ for each computation segment s (either parallel or sequential). The goal of our task *stretch* transform illustrated in Figure 5.6 is

```
Algorithm: StretchTask
input: $\tau_i : (\tau_i^{1,1}, (\tau_i^{2,1}, ..., \tau_i^{2,m_i}), ..., \tau_i^{s_i,1})$
output:$(\tau_i^{master}, \{\tau_i^{cd}\})$
$\tau_i^{master} \leftarrow ()$
$\{\tau_i^{cd}\} \leftarrow \{\}$
if $C_i \leq T_i$ then
    ▷ Make the task sequential
    for $s \leftarrow 1$ to $\frac{s_i-1}{2}$ do
        $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s-1,1} : (C_i^{2s-1})$
        for $k \leftarrow 1$ to $m_i$ do
            $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s,k} : (P_i^{2s})$
        end for
    end for
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
else
    ▷ Stretch the task to its deadline
    $f_i \leftarrow \frac{D_i-\eta_i}{\sum_{s=1}^{\frac{s_i-1}{2}} P_i^{2s}}$
    $q_i \leftarrow (m_i - \lfloor f_i \rfloor)$
    for $s \leftarrow 1$ to $\frac{s_i-1}{2}$ do
        $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s-1,1} : (C_i^{2s-1})$
        for $k \leftarrow 1$ to $m_i$ do
            if $k = 1$ or $k > q_i$ then
                $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2s,k} : (P_i^{2s})$ ▷ Part of the master string
            else
                if $k < q_i$ then
                    $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{2s,k} : (P_i^{2s}, (1+f_i)P_i^k)$ ▷ Create a new parallel thread
                else
                    ▷ Split among a parallel thread and the master string
                    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i'^{2s,k} : ((f_i - \lfloor f_i \rfloor)P_i^k)$
                    ▷ Create a new parallel thread
                    $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i''^{2s,k} : ((\lfloor f_i \rfloor + 1 - f_i)P_i^k, (1 + \lfloor f_i \rfloor)P_i^k)$
                end if
            end if
        end for
    end for
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
end if
return $(\tau_i^{master}, \{\tau_i^{cd}\})$
```

Figure 5.6: Task *Stretch* Transformation.

to increase the *execution length* of the *master string* of each task τ_i to equal the task period T_i. We define the operation $\oplus$ of *coalescing* a thread with the *master string* as inserting $\tau_i^{k,j}$ into the *master string* while respecting thread precedence constraints and offsets.

For applying the *stretch* transform to a task τ_i with period T_i and maximum execution time C_i, we need to consider the following two cases for the value of C_i for task τ_i:

• $C_i \leq T_i$. Under the task *stretch* transform, τ_i can be treated as a *non-parallel* task with execution requirement C_i, task period T_i, and an implicit deadline D_i of T_i. From an implementation perspective, this simply requires all the subtasks of τ_i to be statically assigned to the same processor core. The actual program need not be modified for the *stretch* transform, as it can be easily accomplished at the OS scheduler level when assuming unique priorities or by assigning `OMP_NUM_THREADS` to 1. We can therefore ignore such tasks in this section.

• $C_i > T_i$. In this case, the *stretch* transform cannot avoid the fork-join structures completely. However, it can maximize the slack available to the parallelized segments by assigning the *master string* to its own processor core and eliminating any interference to the *master string*. From an implementation perspective, these tasks can also be transformed at the OS scheduler level without requiring modifications to the original task as we will show later.

The *stretch* transform is given in Figure 5.5. Consider the original task τ_i scheduled on m processor cores with an implicit deadline of $D_i = T_i$. The positive slack L_i available to task τ_i when it is scheduled exclusively without interference from other tasks is given by:

$$L_i = (D_i - \eta_i) \tag{5.2}$$

For the example task τ_1 given in Figure 5.3, $D_1 = 15$ and $\eta_1 = 10$ yielding a slack of $L_1 = 5$.

For notational convenience, we define

$$f_i = \frac{L_i}{\sum_{s=1}^{\frac{s_i-1}{2}} P_i^{2s}} = \frac{L_i}{P_i} \tag{5.3}$$

In the context of example task τ_1 in Figure 5.3, $L_1 = 5$ and $P_1 = 6$ resulting in $f_1 = \frac{5}{6}$.

As $C_i > T_i$, the *master string* can always be stretched to have an execution length of $D_i = T_i$. Specifically, stretching can be performed by having two or more parallel segments (or parts of them) being executed in sequence to take up the available slack. The remainder of any partial execution segment executed in this sequence will be executed in another core. The slack can therefore be distributed proportionally among the set of parallel segments to create constrained deadlines for scheduling purposes. Each parallel segment $2s$, $\forall 1 \leq s \leq (s_i - 1)/2$, is assigned a deadline d_i^{2s} as follows:

$$d_i^{2s} = P_i^{2s}(f_i + 1)\ \forall 1 \leq s \leq \tfrac{s_i - 1}{2}$$

In the case of τ_1 in Figure 5.3, d_1^2 would be equal to $6(\frac{5}{6} + 1) = 11$.

In order to reduce the utilization loss arising from task partitioning, the *master string* of task τ_i is assigned its own processor. The slack available in each of the parallel segments for the *master string* is given by:

$$L_i^{2s} = (d_i^{2s} - P_i^{2s})$$

For example task τ_1, $L_1^2 = 5$ is the slack.

As the master string is assigned its own processor core, we can allocate this slack L_i^{2s} to threads from the same parallel segment as much as possible and coalescing them with the master string $\tau_i^{2s,1}$. After this allocation, each parallel segment will comprise of:

- $\tau_i^{2s,1}$ with a computation requirement of d_i^{2s} and constrained deadline of d_i^{2s},
- $(m_i - \lfloor f_i \rfloor - 2)$ parallel threads with a computation requirement of P_k^{2s} and a constrained deadline of d_i^{2s}
- one remaining thread with a computation requirement of $r_k^{2s} = (\lfloor f_i \rfloor + 1 - f_i)P_i^{2s}$ and a constrained deadline of $(1 + \lfloor fi \rfloor)P_i^{2s}$. The remaining computation will be executed on the processor allocated to the master string.

The total number of parallel threads in each parallel segment is given by:

$$q_i = (m_i - \lfloor f_i \rfloor)$$

where, all q_i threads have the same relative deadline, $(q_i - 2)$ threads have equal execution requirements, one thread (part of the master string) can possibly have a larger execution requirement than the others, and one thread can possibly have a smaller execution requirement than the others.

In the context of τ_1 in Figure 5.3 $q_i = 4$. τ_1 has equal execution requirements on processors 2 and 3, while there is more execution requirement from τ_1 on processor 1, and less execution requirement from τ_1 on processor 4.

The *master string* is schedulable by itself exclusively on a processor since it has an execution requirement of exactly $D_i = T_i$ due to the *stretch* operation described above.

5.3.1 Advantages of Task Stretch

The key advantages of task stretch are as follows: **(a)** The master string has an execution requirement exactly equal to the task period (and implicit deadline). This leads to an efficient task allocation under partitioned multiprocessor scheduling algorithms by avoiding any fragmentation of the available processor utilization. **(b)** The parallel threads can be statically assigned a release offset, avoiding any release jitter that may arise otherwise if the master string were to be co-scheduled with other higher-priority threads. The static offset ϕ_i^{2s} for threads $\tau_i^{2s,r}$ $\forall 1 \leq s \leq (s_i - 1)/2$ and $\forall 1 \leq r \leq q_i$ in parallel segment $2s$ is given by:

$$\phi_i^{2s} = \sum_{k=0}^{k=(s-1)} C_i^{2k+1} + \sum_{k=1}^{k=(s-1)} d_i^{2k}$$

(c) Each parallel thread $\tau_i^{2s,r}$ $\forall 1 \leq s \leq (s_i - 1)/2$ can be considered to be a uniprocessor constrained-deadline task with a release offset of ϕ_i^{2s}, deadline of d_i^{2s}, and a period of T_i. This enables us to leverage known results for scheduling constrained deadline tasks on multiprocessors. We will use this approach for the scheduling algorithm that we propose for *fork-join* task sets. **(d)** The density of a subtask is defined as the ratio of its computation requirement to its relative deadline. All parallel threads other than those belonging to the master string itself, have a maximum density δ_i^{max} of:

$$\delta_i^{max} = \max_{s=1}^{\frac{s_i-1}{2}}\{\frac{P_i^{2s}}{(1+f_i)P_i^{2s}}, \frac{(\lfloor f_i \rfloor + 1 - f_i)P_i^{2s}}{(1+\lfloor f_i \rfloor)P_i^{2s}}\}$$

$$= \max_{s=1}^{\frac{s_i-1}{2}}\{\frac{P_i^{2s}}{(1+f_i)P_i^{2s}}, \frac{P_i^{2s} - (f_i - \lfloor f_i \rfloor)P_i^{2s}}{(1+f_i)P_i^{2s} - (f_i - \lfloor f_i \rfloor)P_i^{2s}}\}$$

$$\forall f_i \geq 0 \Longrightarrow \delta_i^{max} = \frac{1}{1+f_i} \tag{5.4}$$

This property is useful in developing resource augmentation bounds for partitioned preemptive fixed-priority scheduling for fork-join task sets. It is important to observe here that the task stretch transform is a means for achieving guaranteed schedulable utilization within a resource augmentation bound, and does not always result in the best possible schedulable utilization, hence we would need an in-depth analysis for our model.

5.4 Fixed-Priority Partitioned Fork-Join Scheduling

Multiprocessor scheduling algorithms are traditionally classified as (i) partitioned approaches, where tasks are not allowed to migrate across processor cores, and (ii) global scheduling, where tasks are allowed to migrate across processor cores. In this work, we focus on *subtask*-level partitioned scheduling, where each *subtask* $\tau_i^{s,k}$ obtained from the task *stretch* transform of task τ_i is statically assigned to a processor core. We also restrict our attention to preemptive fixed-priority scheduling, specifically *deadline-monotonic* scheduling (DMS) [11].

As mentioned in the earlier section, our approach is to *stretch* the tasks whenever possible to avoid fork-join (FJ) structures that could potentially lead to unschedulability at low utilization levels. The constrained deadline subtasks generated by the *stretch* transform (if any) can be scheduled using a standard partitioned constrained-deadline task scheduling algorithm such as FBB-FFD [48]. Our partitioned DMS scheduling algorithm for fork-join task sets is provided in Figure 5.7. As can be seen, the master strings are allocated to their own individual cores, while the remaining constrained-deadline tasks are scheduled using *FBB-FFD*.

The FBB-FFD (Fisher Baruah Baker - First-Fit Decreasing) algorithm uses a variant of the

```
Algorithm: *Partitioned-FJ-DMS*
input: $\tau : \{\tau_1, \tau_2, ..., \tau n\}$
output: Processor Core Assignment
$\tau^{fdd} \leftarrow \{\}$
for $i \leftarrow 1$ to $n$ do
    $(\tau_i^{master}, \{\tau_i^{cd}\}) \leftarrow StretchTask(\tau_i)$
    if $\{\tau^{cd}\} = \{\}$ then
        ▷ Task has $C_i \leq T_i$
        ▷ $\tau_i$ is not fully stretched
        $\tau^{fdd} \leftarrow \tau^{fdd} \cup \tau_i^{master}$
    else
        Assign a processor exclusively for $\tau_i^{master}$
        $\tau^{fdd} \leftarrow \tau^{fdd} \cup \{\tau_i^{cd}\}$
    end if
end for
Assign processors for $\tau^{fdd}$ using FBB-FFD ([48])
return
```

Figure 5.7: A Partitioned DMS Algorithm for Fork-Join task sets.

First-Fit Decreasing bin-packing heuristic, wherein tasks are considered for allocation in the decreasing order of deadline-monotonic priorities, and each task is allocated to the first available core that satisfies a sufficient schedulability condition proposed in [48].

We will now provide key resource augmentation results for our fixed-priority scheduling algorithm in the context of FJ task sets leveraging the analysis of FBB-FFD from [48].

5.4.1 Analysis

The resource augmentation bound for the m-processor partitioned deadline-monotonic scheduling algorithm provided in Figure 5.7 is a *processor speedup factor* of 3.42. This implies that if a task set τ is feasible on m identical unit speed processors, then the *Partitioned-FJ-DMS* algorithm is guaranteed to successfully partition and schedule this task set on a platform comprising of m processors that are each 3.42 times as fast as the original. In order to derive this result, we use the notion of a *Demand Bound Function* (DBF) [21] for each task τ_i, which represents the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t. For a task τ_i with a total computation requirement of C_i, period of T_i, and a deadline of D_i ($\leq T_i$) given by:

$$DBF(\tau_i, t) = \max\left(0, (\lfloor\frac{t - D_i}{T_i}\rfloor + 1)C_i\right) \tag{5.5}$$

The density of constrained-deadline task τ_i is $\delta_i = \frac{C_i}{D_i}$.

The cumulative demand bound function δ_{sum} is defined as:

$$\delta_{sum} = \max_{t>0}\left(\frac{\sum_{i=1}^{n} DBF(\tau_i, t)}{t}\right) \tag{5.6}$$

The total utilization u_{sum} is given by $u_{sum} = \sum_{i=1}^{n}\frac{C_i}{T_i}$.

Lemma 6. *For any given implicit-deadline task τ_i with the maximum execution length less than or equal to the corresponding task period, the* stretch *transform does not affect the Demand Bound Function, i.e. if the stretched task is represented as $\tau_i^{stretched}$*

$$\forall \tau_i \; s.t. \; C_i \leq T_i, \forall t, DBF(\tau_i, t) = DBF(\tau_i^{stretched}, t)$$

Proof. The demand bound function for τ_i is given by Equation (5.5). Using the implicit-deadline nature of τ_i:

$$DBF(\tau_i, t) = \max(0, (\lfloor\frac{t}{T_i}\rfloor)C_i)$$

We consider τ_i with maximum execution length less than or equal to the corresponding task period, and hence τ_i will get fully *stretched*. Observe that the total execution requirement, period, and deadline of the stretched task $\tau_i^{stretched}$ are the same as that of the original task τ_i. Therefore, the demand bound function remains unchanged i.e. $DBF(\tau_i^{stretched}, t) = DBF(\tau_i, t)$. □

Corollary 2. *For any given implicit-deadline task τ_i with a maximum execution length less than or equal to the corresponding task period, the resulting* stretched *task $\tau_i^{stretched}$ from applying the*

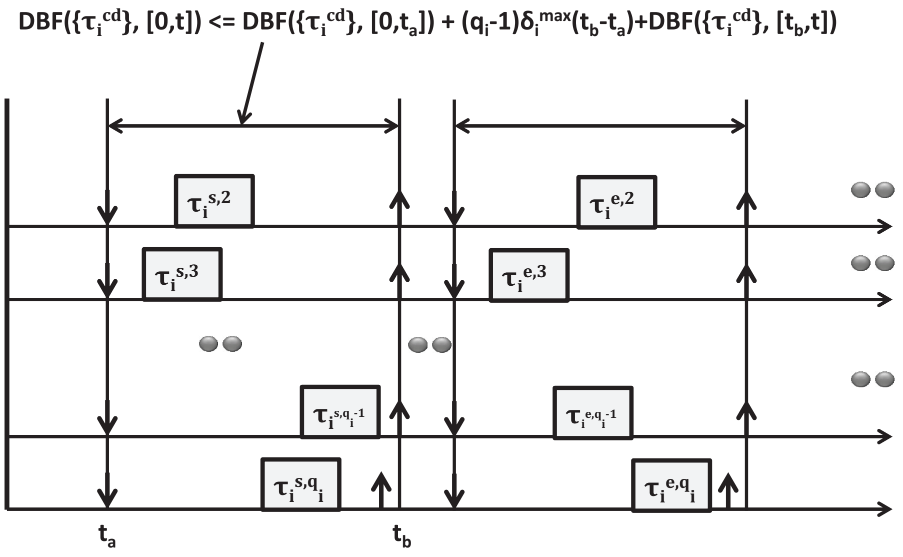

DBF({τ_i^cd}, [0,t]) <= DBF({τ_i^cd}, [0,t_a]) + (q_i-1)δ_i^max(t_b-t_a)+DBF({τ_i^cd}, [t_b,t])
τ_i^{s,2}
τ_i^{e,2}
τ_i^{s,3}
τ_i^{e,3}
τ_i^{s,q_i-1}
τ_i^{e,q_i-1}
τ_i^{s,q_i}
τ_i^{e,q_i}
t_a
t_b

Figure 5.8: Demand bound function of $\tau_i^{stretched}$.

stretch *transform has a Demand Bound Function* $DBF(\tau_i^{stretched}, t)$ *that is bounded from above by* $\frac{C_i t}{T_i - \eta_i}$*, when* $0 \leq \eta_i \leq T_i$*, i.e.*

$$\forall t, DBF(\tau_i^{stretched}, t) \leq \frac{C_i t}{T_i - \eta_i}$$

Proof. The proof follows from Lemma 6. The Demand Bound Function of $\tau_i^{stretched}$ is given by:

$$\begin{aligned} DBF(\tau_i^{stretched}, t) = DBF(\tau_i, t) &= max(0, \lfloor \tfrac{t}{T_i} \rfloor C_i) \\ &\leq \tfrac{C_i t}{T_i} \\ &\leq \tfrac{C_i t}{T_i - \eta_i} \text{ (since } 0 \leq \eta_i \leq T_i) \end{aligned}$$

□

Lemma 7. *For any given implicit-deadline task* τ_i *with a maximum execution length greater than the corresponding task period, the resulting* stretched *task* $\tau_i^{stretched}$ *from applying the* stretch *transform has a Demand Bound Function* $DBF(\tau_i^{stretched}, t)$ *bounded from above by* $\frac{C_i t}{T_i - \eta_i}$*, i.e.*

$$\forall t, DBF(\tau_i^{stretched}, t) \leq \frac{C_i t}{T_i - \eta_i}$$

Proof. After the *stretch* transform (from Figure 5.6) is applied on a task τ_i with maximum execution length greater than the corresponding task period, the task $\tau_i^{stretched}$ can be represented as two components: (i) master string τ_i^{master}, and (ii) set of constrained-deadline sub tasks $\{\tau_i^{cd}\}$.

The demand bound function of $\tau_i^{stretched}$ over an interval t can be represented as:

$$DBF(\tau_i^{stretched}, t) \leq DBF(\tau_i^{master}, t) + DBF(\{\tau_i^{cd}\}, t) \tag{5.7}$$

The *stretch* transform ensures that the master string τ_i^{master} has an execution requirement of T_i, which equals the original task period T_i, therefore,

$$DBF(\tau_i^{stretched}, t) \leq t \tag{5.8}$$

The constrained-deadline subtasks in set $\{\tau_i^{cd}\}$ are offset (as shown in Figure 5.8) to ensure that subtasks belonging to different parallel execution segments of τ_i are never simultaneously active. If the release offset of subtasks corresponding to the execution segment s of τ_i is represented as ϕ_i^s, and its deadline is D_i^s, then

$$\phi_i^s + D_i^s \leq \phi_i^s$$

From the DBF perspective, this property of subtasks in $\{\tau_i^{cd}\}$ ensures that the demand from $\{\tau_i^{cd}\}$ over any interval of length t does not exceed $\delta_i^{max}(q_i - 1)t$ (see Figure 5.8).

$$DBF(\{\tau_i^{cd}\}, t) \leq \delta_i^{max}(q_i - 1)t \leq \frac{1}{1 + f_i}(q_i - 1)t$$

(using Equation (5.4)) using Equations (5.3) & (5.2) with $D_i = T_i$ (using $P_i \geq 0$)

$$DBF(\{\tau_i^{cd}\}, t) \leq \frac{(q_i - 1)P_i}{P_i + T_i - \eta_i}t \leq \frac{(q_i - 1)P_i}{T_i - \eta_i}t \tag{5.9}$$

Using Equations (5.8) and (5.9) in (5.7), we get

$$\begin{aligned}
DBF(\tau_i^{stretched}, t) &\leq t + \frac{(q_i - 1)P_i}{T_i - \eta_i}t \\
&\leq \frac{T_i - \eta_i + (q_i - 1)P_i}{T_i - \eta_i}t \\
&\leq \frac{f_i P_i + (m_i - \lfloor f_i \rfloor - 1)P_i}{T_i - \eta_i}t \\
&\leq \frac{m_i P_i}{T_i - \eta_i}t \\
&\leq \frac{C_i}{T_i - \eta_i}t
\end{aligned}$$

□

For proving the resource augmentation bound for *Partitioned-FJ-DMS*, we will use the following result from Theorem 2 of [48],

Theorem ([48]): *Any constrained sporadic task system τ is successfully scheduled by FBB-FFD on m unit-capacity processors for*

$$m \geq \frac{\delta_{sum} + u_{sum} - \delta_{max}}{1 - \delta_{max}} \tag{5.10}$$

Using this, we can now provide the resource augmentation bound for Partitioned-FJ-DMS.

Theorem 4. *If any fork-join task set τ is feasible on m identical unit speed processors, then Partitioned-FJ-DMS is guaranteed to successfully allocate this task set on m identical processors that are each 3.42 times as fast as the original.*

Proof. The fork-join task set τ is feasible on m identical unit speed processors, which implies

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \tag{5.11}$$

Otherwise, τ is not feasible since over any time interval of length t, only mt units of computation cycles are available on an unit speed processor, while τ demands an utilization greater than m.

Consider the *minimum execution length* η_i for any task τ_i. It must be the case that

$$\forall 1 \leq i \leq n \;\; \eta_i \leq T_i \tag{5.12}$$

Otherwise, τ_i would be unschedulable on unit-speed processors from Proposition 1.

On a processor that is ν times faster, the *minimum execution length* η_i^ν is given by

$$\forall 1 \leq i \leq n \;\; \eta_i^\nu = \frac{\eta_i}{\nu} \leq \frac{T_i}{\nu} \tag{5.13}$$

There are two possible cases for each task τ_i in *Partitioned-FJ-DMS*,

Case 1. Task τ_i can be stretched into an implicit-deadline task on ν speed processors i.e. on a ν speed processor, the maximum execution length ($C_i^\nu = \frac{C_i}{\nu}$) of τ_i is less than or equal to its period T_i. In this scenario, $\tau_i^{stretched}$ is treated as a standard implicit-deadline task. Therefore, using Corollary 2, we get: $DBF(\tau_i^{stretched}, t) \leq \frac{C_i^\nu t}{(T_i - \eta_i^\nu)}$

Case 2. Task τ_i cannot be stretched into an implicit-deadline task on ν speed processors i.e. on a ν speed processor the maximum execution length ($C_i^\nu = \frac{C_i}{\nu}$) of τ_i is greater than the task period T_i. In this scenario, using Lemma 7, we see that $DBF(\tau_i^{stretched}, t) \leq \frac{C_i^\nu t}{(T_i - \eta_i^\nu)}$

Using the above cases for all the tasks in the task set,

$$\delta_{sum}^v = \max_{t>0}\left(\frac{\sum_{i=1}^{n} DBF(\tau_i^{stretched}, t)}{t}\right) \leq \sum_{i=1}^{n} \frac{C_i^\nu}{(T_i - \eta_i^\nu)} \tag{5.14}$$

From Equation (5.13),

$$\forall 1 \leq i \leq n \;\; \eta_i^\nu \leq \tfrac{T_i}{\nu} \Longrightarrow (T_i - \eta_i^\nu) \geq T_i(1 - \tfrac{1}{\nu})$$

Using this in Equation (5.14),

$$\delta_{sum}^\nu \leq \sum_{i=1}^{n} \frac{C_i^\nu}{T_i(1 - \frac{1}{\nu})} \Longrightarrow \delta_{sum}^\nu \leq \tfrac{1}{\nu-1} \sum_{i=1}^{n} \frac{C_i}{T_i} \Longrightarrow \delta_{sum}^\nu \leq \tfrac{1}{\nu-1} u_{sum}$$

Also, on ν speed processors,

the total utilization $u^{\nu}_{sum} = \frac{u_{sum}}{\nu}$ and $\delta^{\nu}_{max} = \frac{\delta_{max}}{\nu}$.

Using Equation (5.10), the task set is schedulable if

$$m \geq \frac{\delta^{\nu}_{sum}+u^{\nu}_{sum}-\delta^{\nu}_{max}}{1-\delta^{\nu}_{max}}$$

Therefore, the task set is schedulable if

$$\implies m \geq \frac{\frac{u_{sum}}{\nu-1}+\frac{u_{sum}}{\nu}-\frac{\delta_{max}}{\nu}}{1-\frac{\delta_{max}}{\nu}}$$

From Equation (5.11), the task set is schedulable if

$$m \geq \frac{\frac{m}{\nu-1}+\frac{m}{\nu}-\frac{\delta_{max}}{\nu}}{1-\frac{\delta_{max}}{\nu}}$$

This is an increasing function of δ_{max} for $m \geq \frac{\nu}{2}$

The density of any parallel thread with constrained deadlines is bounded from above in Equation (5.4) as

$$\forall f_i \geq 0 \implies \delta^{max}_i = \frac{1}{1+f_i} \text{ and } f_i = \frac{T_i-\eta_i}{P_i}$$

$\implies \frac{1}{1+f_i} = \frac{P_i}{T_i-\eta_i+P_i} \leq 1$ (since $\eta_i \leq T_i$ from Inequality (5.12)).

Therefore, when $m \geq \frac{\nu}{2}$, the schedulability is ensured if

$$m \geq \frac{\frac{m}{\nu-1}+\frac{m}{\nu}-\frac{1}{\nu}}{1-\frac{1}{\nu}}$$

$$m(1-\frac{1}{\nu}) \geq \frac{m}{\nu-1}+\frac{m}{\nu}-\frac{1}{\nu}$$
$$\nu-\frac{1}{\nu-1} \geq 3-\frac{1}{m} \Longleftarrow \nu \geq (2+\sqrt{2})$$

This holds good for all $m \geq \frac{\nu}{2}$, using $\nu \geq (2+\sqrt{2}) \approx 3.42$ for all $m \geq 2$ processors.

For the uniprocessor case, the feasibility of task set τ implies that all the fork-join tasks can be stretched into implicit-deadline task sets. In this scenario, the task set is schedulable on the processor with speed $\nu \geq 3.42$ under RMS [74], since $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \leq (3.42)ln2$

Hence, any feasible fork-join task set τ on m unit-speed processors is guaranteed to be scheduled by Partitioned-FJ-DMS on m processors, each with speed 3.42. $\square$

5.5 Summary

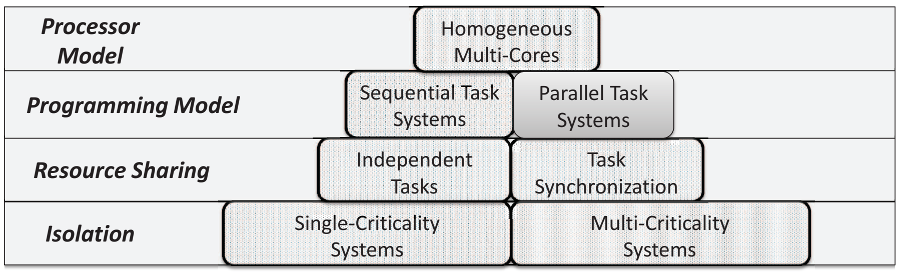
Sequential programming paradigms are ineffective in harnessing the processing capability of evolving massive multi-core systems. Established parallel programming paradigms such as *OpenMP* are promising candidates for extracting better performance from multi-core processors. Parallelism in *OpenMP* is achieved through basic *fork-join* task structures. In this chapter, we have introduced the problem of scheduling *implicit-deadline periodic fork-join* task sets on multiprocessor systems. We illustrated that the worst-case schedulable utilization for *fork-join* task sets on multiprocessors can be slightly greater than and arbitrarily close to 100% (single processor) even when a large number of cores is available. Based on our observations from the worst-case task set characteristics, we define a task *stretch* transform that can be performed by the OS scheduler to avoid fork-join structures as much as possible. The *stretch* transform uses up a single processor core for scheduling the master thread and as many parallel segments as possible. The remaining parallel segments can be scheduled as standard constrained-deadline tasks, where the deadlines themselves are derived from the master thread allocation. The *object splitting* approach is used to *completely* fill up the processor core hosting the master thread by splitting *at most one* parallel segment. We also showed that the stretched task sets can be scheduled using partitioned preemptive fixed-priority multiprocessor scheduling algorithms, and provided the associated resource augmentation bound of 3.42. The task stretch transform is easily implementable on standard real-time operating systems.

Chapter 6

Mixed-Criticality Systems

The assumption so far in this dissertation has been that all the tasks in the system belong to a single criticality domain i.e. each task is equally important as any other task in the system. Although this is a valid assumption for many real-time systems, emerging trends indicate that future systems could host tasks from different criticality domains on the same multi-core processor. This would be an effective way to reduce the number of processors and better utilize the additional computation power of the available processor cores. Additionally, such consolidation of criticality domains using multi-core processors also leads to possibilities for criticality-aware load shedding during overload scenarios. In order to employ multi-core processors in such mixed-criticality systems, we need new techniques for scheduling under overload scenarios and partitioning tasks corresponding to different criticality domains.

In this chapter, we study the problem of provisioning such mixed-criticality systems for han-



dling overload conditions by (i) ensuring that low criticality tasks do not interfere with high criticality tasks from meeting timing deadlines, and (ii) allowing high criticality tasks to steal resources from low criticality tasks to meet deadlines. We use a radar-surveillance system to illustrate the benefits of such overload behavior.

For the purposes of this chapter, we will first start with the assumption of *independent* and *sequential* tasks. We will later introduce the issue of synchronization and address it using priority and criticality inheritance/ ceiling protocols. The key extension with respect to the notation in this chapter is that each task independent and sequential task (compared to Chapter 3) is assumed to have 2 mutually exclusive operating states (i) normal - with an execution budget of C_i, and (ii) overload - with an execution budget of C_i^o. Also, each task τ_i could have a different criticality value κ_i. We still restrict ourselves to tasks with the same deadline D_i as periods T_i.

The rest of this chapter is organized as follows. First, Section 6.1 introduces a motivating radar-surveillance system and describes the mixed-criticality scheduling requirements in that context. In Section 6.2, we formally capture the mixed-criticality scheduling guarantee in terms of our *ductility* metric. Section 6.3 describes the zero-slack rate-monotonic scheduling algorithm for use in uniprocessor mixed-criticality systems. Allocating tasks to processors using the Compress-on-Overload Packing (COP) algorithm is described in Section 6.5. Detailed evaluation and applicability to radar surveillance systems are presented in Section 6.6.

6.1 Radar Surveillance

Electronically Scanned Array (ESA) radar-surveillance systems are examples of large-scale cyber-physical systems, where physical targets in the air space are sensed by actuating the different radar arrays and listening for any corresponding echoes. Timing constraints are key to ensuring that the underlying control algorithms can accurately maintain tracks. Variability in terms of the number of objects within sensing range results in workload fluctuations at the radar. The target profile including identity (*hostile* or *friendly*) and maneuverability determines the *criticality* of each individual track being sensed by the system. The proximity of the target determines the corresponding sampling rate for more accurate tracking.

Tracking algorithms employed in standard radar systems are based on classical techniques like Kalman filtering and $\alpha\beta\gamma$ filtering [66]. These algorithms estimate the speed and direction of targets by combining the radar observation of each target with its previous track data. Estimates are adjusted by periodically aiming the radar at the next estimated position of each target. If the tracking algorithm misses its deadline, then the error in the estimated target position could be large enough causing the corresponding track to be dropped. Meeting tracking task deadlines thus ensures that targets are tracked continually. Timing constraints, therefore, play an important role in radar surveillance systems. Depending on whether the target is a *far-range target* or *near-range target*, the tracking period could be either longer or shorter respectively.

The number of targets to be tracked determines the total workload offered to the tracking algorithm. From the system perspective, it is more important to track *hostile* objects in fine-grained fashion rather than tracking *friendly* objects. This represents a *mixed-criticality* constraint in which certain tasks are more critical than others. Under overload scenarios (i.e. when there are more targets than under normal operating conditions), it is desirable that (i) tracking friendly objects does not interfere with tracking hostile objects, and (ii) hostile object tracking grab resources from friendly object tracking. This property results in what we call as an *overload guarantee* for mixed-criticality scheduling.

Although the number of potential targets could be unbounded, the system requirements typically dictate a minimum number of hostile objects that the radar needs to guarantee to be able to track. This requirement can be translated back into an overload budget that can be assigned for hostile tracks. Under average operating conditions there would still be enough utilization available for tracking friendly objects. However, this utilization would be reclaimed by the critical hostile tracking tasks if the number of hostile objects exceeds normal operating conditions.

In this chapter, we subsequently characterize this requirement and develop solutions for achieving this in both uniprocessor and multiprocessor settings.

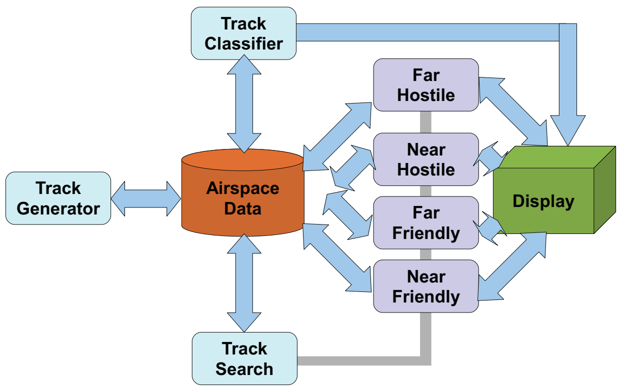


Figure 6.1: Architecture of Radar Surveillance System

The architecture of the ESA radar simulation used in this chapter is shown in Figure 6.1. A *Track Generator* is used to input synthetic targets into the *airspace database*. The *Track Search* process scans the airspace database for new targets that are not currently being tracked by the radar system. Each candidate target is illuminated two consecutive times to avoid spurious objects from getting detected due to noisy data. Each detected track is classified by the *Track Classifier*, which, depending upon its proximity and identity assigns it to one of the following processes:

1. *Far Hostile*: Responsible for hostile tracks operating in the far range. This is of high criticality but requires only a low sampling rate.
2. *Far Friendly*: Handles friendly tracks operating in the far range. This is of low criticality and requires a low sampling rate.
3. *Near Hostile*: Deals with the hostile tracks operating in the near range. This is of high criticality and requires a high sampling rate.
4. *Near Friendly*: This tracks the friendly objects operating in the near range. This is of low criticality but requires a high sampling rate.

The tracks are shown in a display setup, where current tracks are updated and dropped tracks are visually tagged. The parameters of each task are given in Table 6.1. In this radar surveillance

Table 6.1: Radar Surveillance Task Set

$Task\ (\tau_i)$	C_i (ms)	C_i^o (ms)	$Period$ T_i (ms)	$Criticality$ $\kappa_i \in \{1, 2\}$
Near Hostile	40	58	100	1
Far Hostile	83	106	200	1
Near Friendly	40	58	100	2
Far Friendly	83	106	200	2

scenario, it is clearly desirable that tracking friendly objects never interferes with tracking hostiles, and hostile target tracking be allowed to steal resources from friendly object tracking under overloaded conditions. We now proceed to formalize this property using the *ductility* metric.

6.2 System Ductility

In order to evaluate the effectiveness of mixed-criticality scheduling, we need a metric that captures the semantics of mixed-criticality systems. At first glance it may appear that we have two objectives to fulfill: (i) protect the higher-criticality tasks in case of overload, and (ii) achieve high schedulable utilization. Such multi-objective optimization problems have been studied by trade-off approaches such as *Multiple Criteria Decision Analysis* [54] and others more specific to resource allocation like the *QoS Resource Allocation Model* [98]. These approaches use some form of quantification of user preference (e.g. user utility) in order to compare the value obtained from assigning a unit of resource to increase one objective or another. This encoding assumes that at different points of the unit-by-unit allocation process assigning resources to one objective function will return the highest value and at some other point assigning to another will returned the highest.

However, the value of the mixed-criticality objective functions cannot be characterized as a user-preference function where the resource allocation preference switches from one objective function to another. Specifically, there is no point in the allocation process where getting more schedulable utilization is more valuable than protecting higher-criticality tasks. In fact, if we consider that the main purpose of mixed-criticality systems is to protect higher-criticality tasks from being affected by lower-criticality ones then it is clear that this multi-objective function reduces to a hierarchical one. In this case, the first objective is to protect critical tasks and the second one is to obtain as much utilization as possible. In order to capture this formally, we introduce a metric called *ductility matrix* to fully describe the potential behavior of the system with respect to two factors: (1) the level of overload faced by tasks, and (2) tasks that miss their deadlines due to a given overload. In order to characterize the system performance, it is first essential to characterize the possible workloads presented to the system at which performance can be measured. Hence, we now describe the possible workloads in mixed-criticality tasksets, and develop an encoding of the *system workload*.

6.2.1 System Workload

As described earlier, the task model under consideration introduces two new parameters for each task τ_i: (i) an overload execution budget C_i^o, and (ii) a criticality value κ_i (with $\kappa_i \in \{\zeta\}$). The system workload can therefore be in any of 2^k states since each of the k criticality levels can either be *normal* or *overloaded*. The workload of the system under consideration can thus be characterized using a binary encoding called the *workload vector* $< W_1, W_2, ..., W_m >$, where W_k is an indicator variable that denotes the operating state of all tasks τ_j with criticality value $\kappa_j = k$. $W_k = 0$ denotes that all tasks at criticality level k are in the normal operating state. $W_k = 1$ denotes that a task with criticality k is in the overload operating state.

As an example of system workloads, consider the mixed-criticality radar surveillance taskset described in Table 6.1 with two criticality levels. In this task set, the system workload could be in any of the 4 possible states: (i) Both hostile and friendly tracking tasks are overloaded $< 1, 1 >$, (ii) Hostile tracking task is overloaded while friendly tracking task is not overloaded $< 1, 0 >$, (iii) Friendly tracking task is overloaded while hostile tracking task is not overloaded $< 0, 1 >$, and (iv) Both hostile and friendly tracking tasks are not overloaded $< 0, 0 >$.

We can also define a scalar equivalent known as *system workload* (w) that is a comparable quantity to work with, which is computed from the *workload vector* as:

$$w = \sum_{g=1}^{k} \{w_i W_g\}$$

The *system workload* as defined above is thus a weighted sum of the overloads faced by different criticality levels.

As described earlier, it is desirable that there exists a strict ordering among the overloads faced by different criticality levels. One way of guaranteeing this ordering is to assign criticality level g a weight of $w_g = 2^{k-g}$, therefore, any additional overload in criticality level g results in more *system workload* (at least 2^{k-g} additional system overload) than it is possible to add through the maximum overloading of all lower-criticality levels $\{l\}\ \forall l > g$ (at most $\sum_{l=g+1}^{k} 2^{k-l} = (2^{k-g} - 1)$ additional system workload). This property captures the requirement that a high-criticality level be treated as more important than all the other low-criticality levels combined [1]. It is important to note here that the system workload is not a quantification of the amount of workload per-se. Instead, it quantifies the criticality of the system overload.

The idea behind the workload vector is to evaluate the scheduling decisions in the light of the presented workload. For example, given a workload vector $< 1, 0, .., 0 >$, it is desirable that the scheduler meets the peak resource requirements of tasks at the highest criticality level, even if it requires stealing resources from lower criticality tasks. On contrary, given a workload vector of $< 0, 0, ..., 0 >$, the scheduler should meet the normal resource requirements of tasks in all criticality levels.

[1]Other system workload functions could be defined to satisfy other specific requirements.

6.2.2 Ductility Matrix

The *ductility matrix* defines the system timing behavior under possible overload conditions. It is a $2^k \times k$ matrix, where the rows correspond to different possible system workload values in decreasing order $\{2^k - 1, 2^k - 2, ..., 0\}$ i.e. row r ($r \in [1, 2^k]$) has a *system workload* of $w_r = 2^k - r$. The columns correspond to the different possible criticality values in the decreasing order $1, 2, ..., k$. The ductility matrix $D = (d_{r,c})$ encodes whether a task in criticality level c meets its deadlines under a *system workload* of w_r.

- When $d_{r,c} = 1$, it means that a task in criticality level c is guaranteed to meet its deadlines under $w_r = 2^k - r$.
- When $d_{r,c} = 0$, it means that a task in criticality level c is *not* guaranteed to meet its deadlines under $w_r = 2^k - r$.

$$D = \begin{pmatrix} d_{1,1} & d_{1,2} & ... & d_{1,m} \\ d_{2,1} & d_{2,2} & ... & d_{2,m} \\ ... & ... & ... & ... \\ d_{2^k,1} & d_{2^k,2} & ... & d_{2^k,k} \end{pmatrix}$$

The ductility matrix describes the system performance under all possible overload scenarios with respect to criticality levels and all possible overload scenarios. Given the taskset and scheduling algorithm used in the system, the ductility matrix can be calculated to describe the complete behavior of the system under a mixed-criticality scheduler. The ductility matrix is thus an *evaluation tool* chosen for applications where strict criticality is required. There are many potential example application scenarios including smart grids (e.g. to keep security devices energized by turning off entertainment ones), medical devices(e.g. keeping a safe heart beat even if the pacemaker does not have enough resources to report results remotely), radar (as presented), and robotics (e.g. avoiding collisions at the cost of route planning).

It is worth noting that the ductility matrix is similar in nature to the Decision Making (DM) matrix-based multi-attribute decision making solution presented in [128]. However, in our case the columns are strictly ordered given that meeting the deadline of a higher-criticality task is always more important than meeting the deadline of any lower-criticality task.

6.2.3 Ductility

The ductility matrix is a comprehensive description of the system performance with respect to criticality levels. To simplify the evaluation of different scheduling algorithms, we define a scalar equivalent of the ductility matrix that can be ordered based on the magnitude. P is a projection mapping function that maps a matrix M to a scalar value S. Let us define *ductility* d, which is a scalar equivalent of the *ductility matrix* D using the projection function P_d, where:

$$P_d(D) = \sum_{c=1}^{k}\{\frac{1}{2^c}\frac{\sum_{r=1}^{2^k} d_{r,c}}{2^k}\}$$

The key properties of this projection function P_d are:

1. All entries within each column belong to the same criticality level, and are therefore treated equally without assigning different weights to each row (using $\sum_{r=1}^{2^k} d_{r,c}$). Under non-anomalous scheduling algorithms [10], it can be expected that if the tasks meet their deadlines under overloaded conditions, they will continue to meet their deadlines under non-overloaded conditions.
2. The contribution to the final scalar $P_d(D)$ of having a 1 in any row in column c is larger than the contribution of having all ones in all other columns l with lower criticality. Thus, every task in criticality level c is treated as absolutely more important than all the tasks of all the other lower criticality levels $l \; \forall c < l \leq k$. This is accomplished by normalizing the contribution of each column c to the range $[0, 1]$ (by applying a scale of $\frac{1}{2^k}$ to $\sum_{r=1}^{2^k} d_{r,c}$) and subsequently applying a weight of $\frac{1}{2^c}$ to impose a strict ordering among columns.

Note that the maximum value of d obtained using P_d will be $\sum_{c=1}^{k} \frac{1}{2^c} = 1 - \frac{1}{2^k}$. Therefore, we obtain the normalized ductility ν (normalized to the range $[0, 1]$) as:

$$\nu = \frac{P_d(D)}{1 - \frac{1}{2^k}}$$

In this chapter, we will use this *normalized ductility* ν to compare the performance of various scheduling algorithms. It should be emphasized here that *Ductility* represents one possible quantification of the system resiliency to critical overloads. Many other projection functions are also possible for the ductility matrix. However, we believe that P_d succinctly captures the mixed-criticality scheduling requirements from the system designer's perspective. Multiple projection functions themselves are not an integral part of the ductility matrix. The one presented here is agnostic to the type of overload because in the absence of an overload profile it conveniently assumes the worst-case profile of all tasks in a criticality level overloading. The interesting property is really the level of resiliency offered to the most critical tasks under any type of overload including the worst-case profile. The exploration of other overload profiles is left for future work.

6.2.4 Illustration

Consider the set of four tasks given in Table 6.1. *Near Hostile* and *Near Friendly* are examples of near-range (Home Perimeter) tracking algorithms that require a higher sampling rate (10Hz or a 100ms period), whereas, *Far Hostile* and *Far Friendly* are examples are far-range (Non Perimeter) tracking algorithms that only need a lower sampling rate (5Hz or a 200ms period). The criticality levels reflect whether the tasks are used to track hostile (*Near Hostile* and *Far Hostile*) or friendly (*Near Friendly* and *Far Friendly*) objects.

Assume partitioned rate-monotonic scheduling (say scheduling algorithm R), with tasks *Near Hostile* and *Far Hostile* assigned to processor P_1, and tasks *Near Friendly* and *Far Friendly* assigned to processor P_2. In this scenario, we will now illustrate the development of the ductility matrix, and subsequently calculate the normalized ductility ν.

The ductility matrix $D(R)$ under this scenario is given by:

$$D(R) = \begin{array}{l} w_1 = 3 =< 1, 1 > \\ w_2 = 2 =< 1, 0 > \\ w_3 = 1 =< 0, 1 > \\ w_4 = 0 =< 0, 0 > \end{array} \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

The ductility matrix is developed as follows:

1. When $w = w_1 = 3 =< 1, 1 >$, both criticality levels 1 and 2 are overloaded. Under rate-monotonic scheduling, the *Far Hostile* task (in criticality level 1) will miss its deadline in processor P_1 ($d_{1,1} = 0$)and *Far Friendly* task (in criticality level 2) will miss its deadline in processor P_2 ($d_{1,2} = 0$).
2. When $w = w_2 = 2 =< 1, 0 >$, criticality level 1 is overloaded. Under rate-monotonic scheduling, the *Far Hostile* task (in criticality level 1) will miss its deadline in processor P_1 ($d_{2,1} = 0$) and all other tasks will meet their deadlines ($d_{2,2} = 1$).
3. When $w = w_3 = 1 =< 0, 1 >$, criticality level 2 is overloaded. Under rate-monotonic scheduling, the *Far Friendly* task (in criticality level 2) will miss its deadline in processor P_2 ($d_{3,2} = 0$) and all other tasks will meet their deadlines ($d_{3,1} = 1$).
4. When $w = w_4 = 0 =< 0, 0 >$, both criticality levels are in normal conditions. Under rate-monotonic scheduling, all tasks meet their deadlines ($d_{4,1} = d_{4,2} = 1$).

The ductility is given by $d(R) = (\frac{1}{2}\frac{1}{2} + \frac{1}{2^2}\frac{1}{2}) = 0.375$, since tasks in criticality level 1 meets their deadlines only under $w = 1$ and $w = 0$ and gets a weight of $\frac{1}{2}$ (high criticality), while criticality level 2 meets its deadlines only under $w = 2$ and $w = 0$ and gets a weight of $\frac{1}{2^2}$ (low criticality).

The normalized ductility $\nu(R) = \frac{0.375}{0.75} = 0.5$ (since the maximum ductility for two criticality levels is $1 - \frac{1}{2^2} = 0.75$).

The normalized ductility metric describes the performance in the context of various overload conditions in mixed-criticality systems. In order to improve the normalized ductility, we need to focus on both (i) scheduling within each processor, and (ii) allocation of tasks to processors. First, we develop the zero-slack scheduling algorithm to improve overload performance in each individual processor.

6.3 Zero-Slack Scheduling

6.3.1 Criticality Inversion in Uniprocessors

Traditional uniprocessor scheduling algorithms such as Rate-Monotonic Scheduling (RMS) and Earliest-Deadline First (EDF) [75] aim at maximizing the schedulable processor utilization, while ensuring that task deadlines are still satisfied. These algorithms assume that tasks do not execute beyond their worst-case execution times (WCETs), and do not have a well-defined mechanism for dealing with overloads when tasks do overrun their WCETs. This poses a challenging problem in the context of cyber-physical systems such as the radar surveillance setup, where task behavior is tightly coupled with the operating physical environment, resulting in task WCETs that are often hard to characterize and potentially highly pessimistic. RMS and EDF also assign scheduling priorities to jobs based on either the task period (in the case of RMS) or the absolute deadline (in the case of EDF). This leads to additional problems in mixed-criticality settings, where scheduling priorities assigned by RMS or EDF may not correspond to the criticality of tasks, giving more processor time to a task τ_{lc} that has lower criticality than to a higher criticality task τ_{hc} due to its priority assignment. We identify this behavior as *criticality inversion*. This behavior can lead to deadline misses of high criticality tasks due to processing time assigned to low criticality tasks when an overload occurs.

A straightforward approach for dealing with the criticality inversion problem is to assign scheduling priorities based on criticality (CAPA). However, this could result in significantly low schedulable utilization due to priority inversion [108] arising from tasks with low rate-monotonic scheduling priority that have a high criticality. In order to address this issue, we have proposed a Zero-Slack scheduling [41] algorithm for dealing with criticality inversion in uniprocessors, while still improving schedulable processor utilization.

Zero-Slack (ZS) scheduling is a meta-scheduling algorithm that is designed to work with other priority-driven scheduling algorithms such as RMS. It uses the observation that criticality inversion only matters under overload conditions. Under ZS, the execution of each task τ_i is divided into two different modes: N (normal) and C (critical). In the N mode, all active and otherwise non-suspended tasks in the system are considered to be *ready* for scheduling purposes. Whereas in the C mode of task τ_i, all the tasks with lower criticality than τ_i are considered *suspended* or *blocked* for scheduling purposes. Our admission control algorithm then calculates the execution time available for each mode. It is worth noting here that these two modes (*normal* and *critical*) are scheduling modes that correspond to satisfying the *normal* and *overload* budget requirements of tasks.

We now define the scheduling guarantee of zero-slack scheduling.

6.3.2 Scheduling Guarantee for ZS

ZS performs admission control, and if admitted [2], a task τ_i is guaranteed to run up to C_i^o if no higher criticality task τ_h exceeds its C_h. From the perspective of the ductility matrix, this translates to any taskset schedulable under ZS having a ductility matrix $D(ZS)$ with $d_{r,c} = 1$ for all $r \geq 2^{c-1}$, since r would correspond to a workload $2^k - r$ (where k is the number of criticality levels), where none of the tasks τ_h with higher criticality than c would exceed their C_h.

[2] In the context of zero-slack scheduling, when we use the term schedulable, we refer to the property of satisfying this scheduling guarantee.

$$D(ZS) = \begin{pmatrix} d_{1,1} & d_{1,2} & ... & d_{1,m} \\ d_{2,1} & d_{2,2} & ... & d_{2,m} \\ ... & ... & ... & ... \\ d_{2^{k-1},1} & d_{2^{k-1},2} & ... & d_{2^{k-1},k} \\ d_{2^{k-1}+1,1} & d_{2^{k-1}+1,2} & ... & d_{2^{k-1}+1,k} \\ ... & ... & ... & ... \\ d_{2^k,1} & d_{2^k,2} & ... & d_{2^k,k} \end{pmatrix} = \begin{pmatrix} 1 & x & ... & x \\ 1 & x & ... & x \\ ... & ... & ... & ... \\ 1 & x & ... & x \\ 1 & 1 & ... & x \\ ... & ... & ... & ... \\ 1 & 1 & ... & 1 \end{pmatrix}$$

where x can be either 0 or 1 depending on the actual task set.

Tasksets with k criticality levels schedulable under ZS are thus guaranteed to have a ductility:

$$P_d(D(ZS)) = \frac{1}{2} + \frac{1}{2^3} + ...\frac{1}{2^{2k-1}} = \frac{2}{3} - \frac{1}{3(2^{2k-1})}$$

and, a normalized ductility

$$\nu(ZS) = \frac{\frac{2}{3} - \frac{1}{3(2^{2k-1})}}{1 - \frac{1}{2^k}}$$

The ZS guarantee follows the separation of the overloaded from the non-overloaded situation. This separation allows us to make two strategic decisions. First, when no overload condition is present, we should schedule the task with the objective of maximizing utilization. And secondly, when the system experiences an overload, we avoid modifying the utilization-maximization schedule until the last instant necessary to satisfy our guarantee.

For the purposes of this dissertation, due to space considerations, we restrict our discussion to a self-contained description of the zero-slack rate-monotonic scheduling algorithm (ZSRM), which we leverage for intra-processor scheduling in Section 6.5. An interested reader is referred to [41] for a discussion on the generalized ZS algorithm and associated properties.

6.3.3 Zero-Slack Rate-Monotonic Scheduling

The Zero-Slack Rate-Monotonic (ZSRM) scheduling algorithm applies the principle of zero-slack scheduling in the context of RMS. In this setup, the *zero-slack* instant of any task τ_i is defined as the latest time instant relative to the start of its period at which *blocking* tasks with lower criticality than τ_i will ensure that jobs of τ_i meet their ZS scheduling guarantee under RMS. The *zero-slack* instant of a task τ_i thus defines the relative time instant at which jobs of τ_i switch from their N mode to their C mode, if they do not complete earlier. Avoiding any blocking of lower criticality tasks until the zero-slack instant enables us to optimistically maximize the total schedulable utilization when tasks consume less than their WCETs, while still meeting the ZS scheduling guarantee under overruns of *normal* execution budgets.

The ZSRM algorithm is divided into (i) an off-line admission control algorithm that determines the zero-slack instant for each of the tasks in a task set, and (ii) a runtime enforcement mechanism that prevents interference (i.e. *blocks* tasks) based on the criticality and zero-slack instant of each task.

Admission Control for ZSRM

The admission control test for ZSRM is based on the set of tasks that can preempt or interfere with any task τ_i under consideration. Under ZSRM, there are two different interfering tasksets Γ_i^c and Γ_i^{rm} for each task τ_i corresponding to the two execution modes (C and N=RM). We use C_j^e to represent the *effective* execution time that will be considered for each interfering task τ_j. In particular, in the RM mode, the task set that would interfere with task τ_i is defined as:

$$\Gamma_i^{rm} = H_i^{lc} \cup H_i^{hc} \cup L_i^{hc} \cup S_i$$

where,

* H_i^{lc} is the set of tasks with higher rate-monotonic priorities but lower criticality. These tasks are considered to be running for C^o units. That is, $\forall \tau_j \in H_i^{lc}$, $C_j^e \Leftarrow C_j^o$.

* H_i^{hc} is the set of tasks with higher rate-monotonic priorities and higher criticality. These tasks run for their respective non-overloaded computation time. If a higher-criticality task executes beyond its non-overloaded computation time, then the scheduling guarantee of lower-criticality task τ_i need not be honored. Hence, $\forall \tau_j \in H_i^{hc}$, $C_j^e \Leftarrow C_j$.

* L_i^{hc} is the set of tasks with lower rate-monotonic priorities and higher criticality. These tasks are considered to be running in their C mode. In fact, we only need to consider the part of computation that runs in its C mode. That is, assuming that $a_j^{rm,i}$ contains the slack available to task $\tau_j \mid \tau_j \in L_i^{hc}$ in RM mode, while honoring the guarantee of task τ_i, then $C_j^e \Leftarrow (C_j - a_j^{rm,i})$. Initially, we assume that there is no slack available in RM mode ($a_j^{rm,i} = 0$), but as we move the zero-slack instants of the tasks towards the end of their period (to be discussed shortly) we will discover additional slack in RM mode reducing C_j^e further and increasing, in turn, the available slack in RM mode.

* S_i is the set of tasks with the same level of criticality but higher priority. These tasks are running at their C^o, i.e., $\forall \tau_j \in S_i$, $C_j^e \Leftarrow C_j^o$.

The interfering task set for C mode of task τ_i is defined as:

$$\Gamma_i^c = H_i^{hc} \cup L_i^{hc} \cup S_i$$

with the same set of definitions as before. Note that the low criticality tasks are excluded since the lower-criticality tasks are blocked during τ_i C mode of execution.

Given these interfering tasksets, the admission control scheme iteratively determines the zero-slack instant Z_i for each task τ_i. First, the algorithm starts with a zero-slack instant $Z_i = 0$ for all tasks in the system. The algorithm then iteratively discovers the slack guaranteed to be available for each task by assuming that all higher-criticality tasks execute for their normal execution time. Subsequently, the computation in the $N = RM$ mode can be increased, the computation in C mode can be decreased, and the zero-slack instant Z_i can be increased. This iterative procedure continues until the zero-slack instants cannot be increased any further.

Runtime Behavior of ZSRM

Given the off-line zero-slack instant calculations, the dual-mode implementation is realized using a runtime scheduler mechanism. It is important to keep the overhead of such a mechanism as lean as possible. We can achieve this by using the zero-slack instants as timers to switch to C mode. These timers then mark the time when the lower-criticality tasks are suspended. For this reason, we need to keep track of the criticality level of the tasks to be able to distinguish which tasks need to be suspended. At the same time, we need to keep track of which tasks are in C mode in order to know which tasks to resume when a task finishes its C mode. In a resource reservation framework, the admission test can be performed at the time of the resource reservation and the corresponding zero-slack instants can be computed.

6.3.4 Worst-Case Phasing of Dual-Mode Tasks

Key to the calculation of the zero-slack instants when rate-monotonic scheduling is used is the phasing of the tasks. For a single-mode execution, Liu and Layland [75] proved that the phasing that creates the maximum preemption for a task τ_i happens when every task $\tau_j | priority(\tau_j) < priority(\tau_i)$ arrives at the same time as τ_i. However, in a dual-mode task, this worst-case phasing does not hold. This is because, when tasks reach their zero-slack instants, they will suspend lower-criticality tasks. On the one hand, this suspension acts, as intended, to avoid preemptions suffered by task τ_i from lower-criticality tasks. However, it also acts as a preemption when higher-criticality tasks suspend τ_i. Hence, to calculate the worst-case delay imposed by this type of preemption, we need to align all the suspensions in the same way as the period arrivals. Unfortunately, if we align the zero-slack instants of the higher-criticality tasks, we may misalign the arrival of higher-priority tasks. In other words, it is not always possible to align both the worst-case arrival of the tasks and the zero-slack instants. The implication of this misalignment is that we cannot create a single integrated critical zone based on the alignment of both types of preemptions. As a result, we take a pessimistic approach by assuming that the effects of both

Table 6.2: Zero-Slack-RM Scheduled Task set

Task	C	C^o	T	Criticality	Priority	ZS Instant
τ_0	10	50	100	2	0	80
τ_1	20	100	200	1	1	60
τ_2	40	200	400	0	2	200

alignments always happen.

Although the worst-case phasing may not exist, it provides an upper bound on the total interference imposed on task τ_i. This can be shown as follows. Before the zero-slack instant, the maximum interference from higher-priority tasks happens when they are released simultaneously with τ_i (from [75]). After the zero-slack instant, τ_i effectively blocks all the lower -criticality tasks. Therefore, the interference can only arise from higher-criticality tasks. By switching all the higher-criticality tasks to their critical mode (C) along with τ_i, the interference suffered by τ_i in the critical mode (C) is also maximized.

6.3.5 A Zero-Slack-RM Scheduling Example

Let us use an example to illustrate the characteristics of the zero-slack-RM scheduler. Table 6.2 presents a task set with the priorities assigned by the rate-monotonic scheduler and the zero-slack instants calculated by our algorithm.

Due to space limitations, we will focus our discussions on τ_1. Figure 6.2 presents the critical zone of this task. In this figure, we can see the preemption from τ_0 in the N mode of τ_1 for 50 units of time. After this, τ_1 runs for 10 units and then reaches its zero-slack instant at time 60, switching to C mode. In C mode, it suspends the lower-criticality task τ_0, but at the same time it is suspended by the higher-criticality task τ_2. This suspension is the pessimistic approach we use due to the absence of an exact worst-case phasing (as discussed in Subsection 6.3.4). τ_2 then runs for C_2 (40) units and resumes the lower-criticality tasks. However, in order to maintain the criticality order, this resumption is implemented as a stack, meaning that it only returns to the previous criticality level (leaving τ_0 suspended). Then, τ_1 can continue executing completing its

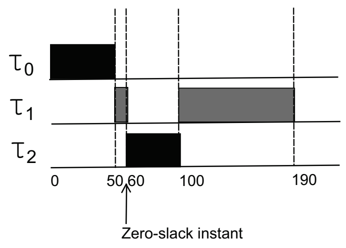


Figure 6.2: Critical Zone of *Task 1*

C_1^o (100) at time 190.

Each task in the task set has its own (pessimistic) critical zone similar to the one presented in Figure 6.2, but they are unfortunately not necessarily aligned with each other.

Properties of The Zero-Slack-RM Scheduler

Theorem 5. *Any task set schedulable under Criticality-As-Priority Assignment(CAPA) is also schedulable under the zero-slack scheduling scheme.*

Proof. The admission control for zero-slack scheduling starts with assigning $Z_i = 0$ for all tasks τ_i. Under this assignment of zero-slack instants, the zero-slack scheduler behaves essentially like a CAPA scheme, since whenever τ_i is released all the lower criticality tasks are immediately blocked due to τ_i switching to its critical mode ($Z_i = 0$). Therefore, if the task set is schedulable under CAPA, it should be schedulable with zero-slack instants of 0. In this scenario, we now inductively prove that each task τ_i remains schedulable over subsequent iterations.

During subsequent iterations of the zero-slack calculation, additional computation from the critical mode (C) is transferred to the normal mode (N). This transfer is performed only to use up the slack available in the normal mode (N) up to the zero-slack instant. Considering any task τ_i, this transfer of computation does not increase the blocking terms suffered by τ_i from higher criticality tasks executing in their C mode. The normal mode N of τ_i remains unaffected, since additional computation is transferred to only fill up available slack. Therefore, the response time of τ_i only reduces in subsequent iterations. Hence, if τ_i was schedulable in the previous iteration, it continues to be schedulable. This completes the induction. □

Theorem 6. *Any task set schedulable under rate-monotonic scheduling is also schedulable under the zero-slack scheduling scheme.*

Proof. For any task set Γ consisting of tasks $\tau_i = (C_i, T_i)$ schedulable under the RM scheduling scheme, consider an equivalent Γ_z with tasks $\tau_i^z = (C_i, C_i^o, T_i, \kappa_i)$ with $C_i = C_i^o = C_i$ and $\kappa_i = \pi_i$, where π_i is the priority assigned to task τ_i under RM scheduling. Scheduling the task set Γ_z using CAPA produces the same schedule as the RM scheduler, since the priorities are completely aligned with the criticality under the chosen κ_i values. Hence, Γ_z is also schedulable under CAPA, since it is schedulable under RM scheduling. Using the property that zero-slack scheduling subsumes CAPA, it follows that Γ_z is also schedulable under zero-slack scheduling. □

Having considered the problem of scheduling independent mixed-criticality sequential tasks, we now consider the problem of task synchronization in such systems.

6.4 Task Synchronization in Mixed-Criticality Systems

Resource-sharing tasks, scheduled with zero-slack schedulers, can suffer resource-locking delays that break the scheduling assumptions in both their *normal* and *critical* execution modes. In normal mode, a low-priority task τ_i can be holding the lock of a resource that a high-priority task τ_j requests. In such a case, τ_j has to wait for τ_i to release the lock to be able to continue executing. Because such waiting has the same effect as assigning a priority higher to τ_i than to τ_j during this waiting time, it can be seen as a sub-optimal priority assignment, effectively reducing the schedulable utilization. For this reason, the priority inheritance schemes have the objective of minimizing this waiting time. In critical mode, these same two tasks may need a completely different solution. For instance, if the criticality level of τ_i is higher than that of τ_j, then the waiting of τ_j for τ_i to release a lock does not create a delay that reduces the schedulable

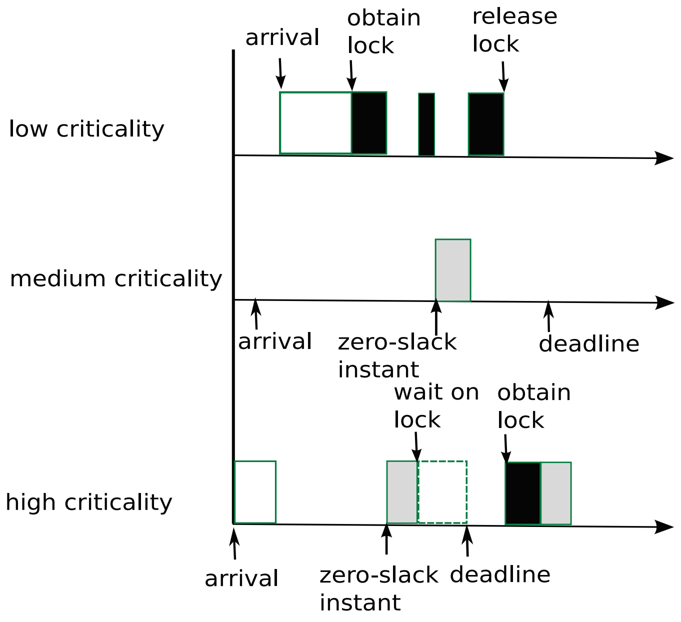


Figure 6.3: Priority and Criticality Inversion

utilization. However, if τ_i waits for τ_j to release the lock, it creates an unwanted delay because, under zero-slack scheduling, τ_j should not be interfering with τ_i in its critical mode.

The priority inversion between a high-priority task τ_j and a low-priority task τ_i due to resource locking can get worse if multiple medium-priority tasks preempt the execution of τ_i while it is holding a lock that τ_j is waiting for. This is known as the *unbounded priority inversion* problem. To avoid this problem, priority-inheritance schemes raise the priority of the task holding the lock, thereby avoiding preemptions from medium-priority tasks. In the case of criticality inversions, a similar problem occurs with medium-criticality tasks. For this problem, an equivalent solution (inheriting the criticality) needs to be developed. In other words, when a task τ_i runs in critical mode, it can suffer *unbounded criticality inversion* due to resource blocking if a low-criticality task τ_j holds the lock of a resource requested by τ_i and multiple medium-criticality tasks preempt τ_j. To avoid this problem, our scheme stipulates that the low-criticality task τ_j inherit the criticality of high-criticality task τ_i.

Let us now classify the waiting scenarios possible in both execution modes of a task. A task

τ_i can be waiting for a lock being currently held by a task that belongs to one of the following subsets:

- H_i^{hc}: the subset of tasks with higher priority and higher criticality than τ_i. Task τ_i waiting on such tasks creates neither priority inversion nor criticality inversion in any of the execution modes of τ_i.
- H_i^{lc}: the subset of tasks with higher priority but lower criticality than τ_i. Task τ_i waiting on a task in this subset does not create priority inversion in normal mode but it can create criticality inversion in critical mode.
- L_i^{hc}: the subset of tasks with lower priority but higher criticality than τ_i. Task τ_i waiting on a task in this subset can create priority inversion in normal mode but does not create criticality inversion in critical mode.
- L_i^{lc}: the subset of tasks with lower priority and lower criticality than τ_i. Task τ_i waiting on a task in this subset can create both priority inversion in normal mode and criticality inversion in critical mode.

In the rest of the discussion, we will also consider that a job belongs to one of these subsets if its corresponding task belongs to it.

Depending on the execution mode, the scheduler uses both *priority* and *criticality* as scheduling criteria. We will use the term *eligibility* to denote the resulting scheduling attribute. Consider any job J of task τ_i. All jobs belonging to subset H_i^{hc} have *higher eligibility* than J. All jobs belonging to subset L_i^{lc} have *lower eligibility* than J. Any job belonging to subset H_i^{lc} has *higher eligibility* than J when J is in its normal execution mode, and *lower eligibility* than J when J is in its critical mode. Any job J' belonging to subset L_i^{hc} has *higher eligibility* than J when J' is in its critical mode, and *lower eligibility* than J when J' is in its normal mode.

Based on these subsets, we now define the conditions under which a job $J_h \in \tau_i$ can be considered as *blocked* by a critical section $z_{l,k}$ of a lower eligibility job J_l.

Definition 1. *A job J_h is said to be blocked by a critical section $z_{l,k}$ of job J_l if J_l has a lower* *eligibility than J_h but J_h has to wait for J_l to exit $z_{l,k}$ in order to continue execution.*

Conditions for Blocking in Zero-Slack Scheduling

A job J_h waiting for a job J_l to exit critical section $z_{l,k}$ is considered to be *blocked* at time t, *if and only if* one the following conditions is satisfied at t:

1. The priority of J_l is lower than J_h's priority and J_l is running in its $normal$ mode.
2. The criticality of J_l is lower than J_h's criticality and J_h is running in its $critical$ mode.

These conditions will be subsequently used in the discussions of the properties of our protocols.

The duration for which a job waits for a lower-priority task in normal mode and a lower-criticality task in critical mode is defined as its *blocking time*.

6.4.1 Bounding the Priority and Criticality Inversions

In this section, we present two protocols to bound the priority and criticality inversions due to resource sharing.

Priority-and-Criticality Inheritance Protocol (PCIP)

In the Priority and Criticality Inheritance Protocol (PCIP), a task τ_i that holds a lock to a resource inherits both the priority of the highest-priority task (say τ_j) and the criticality of the highest-criticality task (say τ_k), waiting on the lock held by τ_i. Note that τ_j and τ_k could be the same task, i.e. the same task could have both the highest priority and highest criticality among all tasks waiting on the lock held by τ_i. The inheritance under PCIP is as follows:

- τ_i inherits the priority of τ_j if τ_j's priority is higher. This inherited priority has an immediate effect on the scheduling of τ_i.
- τ_i inherits the criticality of τ_k if τ_k's criticality is higher. This criticality is used by τ_i immediately as soon as τ_k requests the lock held by τ_i.

The priority inheritance works just as the basic priority inheritance protocol from [108] during the normal execution period of τ_i, if only the priority is inherited. If the criticality value is inherited by τ_i, it is used immediately to prevent τ_i from getting suspended by tasks that have a lower criticality than the newly inherited criticality of τ_i. The inherited priority (if any) is used to continue the scheduling of the rest of the tasks. Once τ_i releases all its locks, the priority of τ_i is restored to its normal execution priority. However, if τ_i has already entered its critical mode, then its criticality is restored to ζ_i and the appropriate tasks are unblocked. As with the basic priority inheritance protocol in [108], PCIP is also transitive.

PCIP Execution Modes Since the priority and the criticality of a critical section can be inherited from different tasks, we need to consider all combinations to understand the worst-case blocking time of a task. Under PCIP, a critical section in task τ_i can be executing in one of the following modes:

i) Normal mode. In this mode, the critical section is executing in τ_i's priority. This happens when no other higher-priority task has attempted to acquire the lock, and τ_i is in its *normal* mode.

ii) Inherited priority mode. In this mode, the critical section is executing with the priority inherited from a task τ_j. This happens when a number of tasks have already requested to acquire the lock, τ_j has the highest priority among them, and τ_j has a higher priority than τ_i. As other tasks try to acquire the lock, the highest-priority task requesting the lock can change, and so does the inherited priority.

iii) τ_i's critical mode with normal priority. In this mode, τ_i's zero-slack instant has already elapsed, and the task has entered its own critical mode. In this mode, all tasks with lower criticality than τ_i are suspended and τ_i's own priority is used to schedule the execution of its critical section. Also, no other higher-criticality or higher-priority (or both) task has attempted to acquire the lock.

iv) τ_i's critical mode with inherited priority. In this mode, τ_i's zero-slack instant has already elapsed and the task has entered its own critical mode. Before entering its critical mode, a higher-priority but lower-criticality task τ_j has requested the lock, and hence τ_i has inherited this priority but not its criticality.

v) Inherited criticality with normal priority. In this mode, τ_i is executing while holding locks, while a task τ_k with higher criticality but lower priority executes and requests a lock already locked by τ_i. At this time, τ_i inherits τ_k's criticality but not its priority.

vi) Inherited criticality with inherited priority. In this case, both a task τ_k with higher criticality and a task τ_j with higher priority have requested the lock, and τ_i has inherited both criticality and priority from them.

An important requirement under zero-slack scheduling is that whenever the zero-slack instant of a job of any task τ_k is reached, all jobs with lower criticality (inherited or otherwise) than τ_k are suspended, regardless of whether or not τ_k itself is blocked.

Note that since other tasks can keep requesting the lock, the critical section could be in modes (ii), (iv), (v), and (vi) with different inherited criticalities and priorities.

Properties of PCIP

Before describing the properties of PCIP, let us first introduce some notation that we will be using in the rest of our discussion. Let $\beta_{i,j}$ denote the set of all critical sections $z_{j,l}$ from J_j that have either a lower priority or a lower criticality (or both) than J_i, and can block J_i. Observe that, under PCIP, J_j could either be a job that directly shares a resource with J_i, or J_j could be a job that gains a criticality or priority higher than J_i due to the resources it shares with other jobs of either higher criticality or higher priority (or both).

Since we restrict our attention to properly nested critical sections, the set $\beta_{i,j}$ of blocking critical sections can be partially ordered. This partial ordering enables the definition of the set of maximal elements of $\beta_{i,j}$ denoted as $\beta^*_{i,j}$(see [108]). This set $\beta^*_{i,j}$ contains the longest critical sections of J_j that can block J_i. Let $\lambda(\beta^*_{i,j})$ denote the length of the longest critical section in set $\beta^*_{i,j}$.

Let Ψ_i denote the set of all locks, which could be held by jobs that block J_i, when they block J_i. For any lock $\psi_{i,k} \in \Psi_i$, let $\Lambda(\psi_{i,k})$ denote the worst-case execution time within the critical

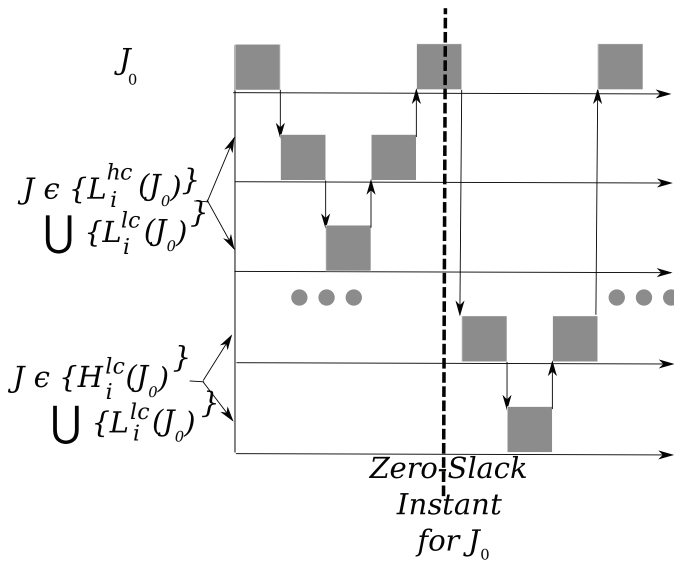

J_0
$J \in \{L_i^{hc}(J_0)\} \cup \{L_i^{lc}(J_0)\}$
$J \in \{H_i^{lc}(J_0)\} \cup \{L_i^{lc}(J_0)\}$
Zero-Slack Instant for J_0

Figure 6.4: Possible blocking scenario for job J_0 (i) from $J \in \{L_i^{hc}(J_0)\} \cup \{L_i^{lc}(J_0)\}$ in its normal mode, and (ii) from $J \in \{H_i^{lc}(J_0)\} \cup \{L_i^{lc}(J_0)\}$ in its critical mode

section protected by $\psi_{i,k}$, over all tasks that can hold $\psi_{i,k}$.

Theorem 7. *Under the priority-and-criticality inheritance protocol, given a job J_0 for which there are n jobs $\{J_1, ..., J_n\}$ with $J_i \in \{H_0^{lc} \cup L_0^{lc} \cup L_0^{hc}\}$, $1 \leq i \leq n$, job J_0 can be blocked for at most the duration of one critical section in each of $\beta_{0,i}^*$, $1 \leq i \leq n$.*

Proof. The blocking suffered by J_0 is shown in Figure 6.4. Let us consider the following scenarios for any job J_i that can block J_0:

- $J_i \in L_0^{lc}$**:** Under PCIP, J_0 can be blocked by a lower-priority and lower-criticality job J_i only if J_i shares a resource with J_0, or J_i gains a priority or criticality greater than J_0. In both these scenarios, it is the case that J_i holds a critical section, upon completion of which J_0 will regain access to the processor. J_i can never block J_0 after releasing the critical section since J_0 will never relinquish the processor to J_i which is both lower priority and lower criticality, thereby preventing J_i from further acquiring locks before completion of J_0.
- $J_i \in L_0^{hc}$**:** J_i executing in its *normal mode* could block J_0 in its *normal mode* by either

holding the lock required by J_0 or gaining a priority greater than J_0. In both cases, it holds a critical section, upon completion of which J_0 will regain access of the processor. J_i in its *normal mode* can never block J_0 after releasing the critical section since J_0 will never relinquish the processor to J_i unless J_i reaches its zero-slack instant. If J_i does reach its zero-slack instant and enters its *critical mode*, it can be no longer considered as blocking J_0 since J_i is given scheduling precedence anyway in its *critical mode* by virtue of its higher criticality.

- **$J_i \in H_0^{lc}$:** J_i in its *critical mode* could block J_0 in its *critical mode* by either holding the lock required by J_0 or gaining a criticality greater than J_0. In both cases, it holds a critical section, upon completion of which J_0 will regain access to the processor. J_i can never block J_0 in its *critical mode* after releasing the critical section since J_0 will never relinquish the processor to J_i before completion. In the *normal mode* J_i cannot be considered as blocking J_0 since J_i is given scheduling precedence anyway in its *normal mode* by virtue of its higher priority.

Therefore, J_i can block J_0 for at most the duration of one critical section in $\beta^*_{0,i}$. □

Theorem 8. *Under the priority-and-criticality inheritance protocol, if there are m locks which can block job J, then J can be blocked at most m times in its normal mode and blocked at most m times in its critical mode.*

Proof. In the *normal mode* of execution for job J, each of the m locks could be held by m jobs. If any of these jobs J_i is of lower priority, it can block J under PCIP till J_i exits the maximal critical section. Once J_i exits the maximal critical section, J_i cannot block J in its normal mode since J will never relinquish the processor to J_i, thereby preventing them from acquiring further locks. If the blocking job J_i is of higher criticality and acquires the processor by entering its critical mode and preempting J executing in its *normal mode*, J_i is no longer considered to block J since it has scheduling precedence.

Under the *critical mode* of execution for job J, each of the m locks could be held by m jobs that are of lower criticality. These jobs can block J under PCIP till they exit their maximal critical section. Once they exit their maximal critical section, they cannot block J since J will never relinquish the processor to them, thereby preventing them from acquiring further locks. $\square$

The Priority-and-Criticality Ceiling Protocol (PCCP)

The priority-and-criticality ceiling protocol follows the structure of the priority ceiling protocol. In this case, each lock is assigned both a *priority ceiling* and a *criticality ceiling*. The priority ceiling of a lock is defined as [108] the highest priority of any possible locker of the lock. Similarly, the criticality ceiling is defined as the highest criticality of any possible locker of the lock.

Both the priority and criticality ceiling values are inherited as priority and criticality values when a resource locker enters a critical section. At that moment, the priority becomes effective, preventing lower-priority jobs from acquiring the processor. With this approach, the properties of the original priority ceiling are preserved in normal mode. Observe that inheriting both the priority and criticality ceiling values whenever a resource locker enters its critical section is similar to the approach used in [13].

Under PCCP, there is no need for using a priority ordering in lock queues, centralizing all scheduling decisions in the scheduler.When a job J_i requests a lock S_l, no other potential locker J_p of S_l is executing or holding S_l since J_p cannot be preempted by J_i when J_p holds S_l due to the immediate nature of the priority and criticality ceiling.

Even though the criticality of a task is not used until its zero-slack instant elapses, the inherited criticality of a job is activated as soon as the locker enters the critical section. This ensures that when a job J enters its critical section, we prevent preemptions from (i) the *normal mode* of jobs with lower priority than the inherited priority of J, (ii) the *critical mode* of jobs with lower criticality than the inherited criticality of J, and (iii) the *normal mode* of jobs with higher priority but lower criticality than J when the zero-slack instant of J elapses. The avoidance of lower-criticality job preemptions under (ii) and (iii) is important because they can acquire additional locks and cause blocking terms which, in turn, affect important properties of our protocol, as we show in Section 6.4.1.

Properties of PCCP

Lemma 8. *Under the priority-and-criticality ceiling protocol, no job J_k can preempt another job J_i while J_i holds a lock (i.e. is inside the critical section) that is also accessed by J_k.*

Proof. The job J_k can preempt J_i's critical section if it has either higher priority or higher criticality, and is executing in its critical mode. However, by the definition of PCCP, when a task enters a critical section for a lock it inherits the higher priority among potential lockers of the lock, including J_k, and the highest-criticality (upon immediately entering a critical section) among the potential lockers, including J_k. As a result, J_k cannot preempt J_i's critical section. $\square$

Definition 2. *Transitive blocking is said to occur if a job J is blocked by J_i which, in turn, is blocked by another job J_j.*

Theorem 9. *The priority-and-criticality ceiling protocol prevents transitive blocking.*

Proof. Let us assume that transitive blocking occurs between jobs J_i,J_j, and J_k. Specifically, J_k is waiting for a lock that is held by J_j which, in turn, is waiting for a lock that is held by J_i. To be able to get to this situation, J_j should have had preempted J_i's critical section. This cannot happen as per Lemma 8 since J_j later waits for a lock held by J_i. This is a contradiction. $\square$

Theorem 10. *Under the priority-and-criticality ceiling protocol, a job J_w can only be blocked for at most one critical section in each of its execution modes (normal mode and critical mode)*

Proof. Any job J_w can be preempted only by higher priority jobs or jobs with higher criticality executing in the critical mode. J_w can be blocked for one lower-priority critical section when released since it cannot preempt a lower-priority job holding a lock that is also accessed by

J_w (using Lemma 8). Additionally, J_w can be blocked for at most one lower-criticality critical section when switching to its critical mode, since it cannot preempt a lower-criticality job holding a lock that is also accessed by J_w (using Lemma 8).

□

Corollary 3. *Under the priority-and-criticality ceiling protocol, a job J_w can only be blocked for at most two critical sections.*

Proof. The proof follows from Theorem 10, since each job J_w has only two execution modes (normal and critical mode). □

Theorem 11. *Under the priority-and-criticality ceiling protocol, a job J_w cannot be blocked for more than one critical section by another job J_l.*

Proof. Let us assume that J_w is blocked twice by job J_l. Using Theorem 10, job J_w can only be blocked once in its normal mode, and once in its critical mode. Therefore, job J_l should be both lower priority and lower criticality than J_w to block J_w twice.

For J_w to be blocked twice by J_l, J_l has to block J_w once in its normal mode when J_w is released. This can happen if J_l is already holding a lock S_l at a priority higher than J_w. J_w needs to execute on the processor before J_l can block J_w once more in the critical mode. Observe that J_l cannot hold any lock S_l with higher priority or criticality ceiling than J_w, when J_w executes on the processor. Therefore, J_l will never get a chance to execute on the processor after J_w acquires the processor and before J_w finishes, since J_w is both higher priority and criticality than J_l, and J_l does not hold any lock with a higher priority or criticality ceiling than J_w. Hence, job J_l cannot block job J_w twice.

□

Theorem 12. *The priority-and-criticality ceiling protocol prevents deadlocks.*

Proof. Let us assume that a deadlock is formed among a set of jobs $J_1..J_n$. This means that each job $J_i \in (J_1..J_{n-1})$ at least holds a lock and is waiting for the release of another lock held by some job J_j, such that $J_j \neq J_i$ and $J_j \in (J_1..J_n)$. Job J_n holds a lock and is waiting on a lock held by some $J_k \in (J_1..J_{n-1})$ closing a waiting cycle. However, in order for this cycle to happen transitive blocking must be possible. Transitive blocking is required to even create the smallest cycle of two, where a job J_i would be blocked by a job J_j while J_i blocks J_j as well. However, by Theorem 9, we know that transitive blocking cannot happen. Hence, this theorem follows from contradiction. □

6.4.2 Blocking Term Calculation

We now quantify the blocking terms under both PCIP and PCCP.

PCIP Blocking Term

For any task τ_i, the worst-case blocking term B_i under PCIP is given by:

$$B_i = \min\Big(\sum_{\tau_j \in \{H_i^{lc} \cup L_i^{lc} \cup L_i^{hc}\}} \lambda(\beta^*_{i,j}), \sum_{\psi_{i,k} \in \Psi_i} 2\Lambda(\psi_{i,k})\Big) \tag{6.1}$$

This follows from Theorems 7 and 8, which imply that any job of τ_i cannot be blocked more than once by each lower-criticality or lower-priority task, and no more than twice per each lock that can be held when a task blocks τ_i. This also leverages the assumption that there is no more than one task per criticality level. For task sets with multiple tasks per criticality level, the interference from tasks with equal criticality and equal- or higher-priority level should also be considered. It should be noted here that although the blocking term is shown to be bounded in Equation (6.1), this assumes that there are no deadlocks i.e. deadlocks are avoided using alternative mechanisms. For instance, a total ordering of locks and ensuring access only in this order, would enable the system to avoid deadlocks.

PCCP Blocking Term

For any task τ_i, the worst-case blocking term B_i under PCCP is given by:

$$B_i = \max_{\tau_j \in \{H_i^{lc} \cup L_i^{lc} \cup L_i^{hc}\}} 2\lambda(\beta^*_{i,j}) \tag{6.2}$$

This follows from Theorem 9 and Corollary 3.

Utilization Bounds

A task τ_i is determined to be schedulable under zero-slack scheduling with synchronization, if:

$$\sum_{\tau_h \in H_i^{hc}} \frac{C_h}{T_h} + \frac{\sum\limits_{\tau_l \in L_i^{hc}} C_l + C_i^o + B_i}{T_i} \leq UB(rms) \tag{6.3}$$

where, $UB(rms)$ is the utilization bound for rate-monotonic scheduling [74]. This test follows from [41] by adding the appropriate blocking term B_i to account for synchronization, which is calculated using Equation (6.1) for PCIP, and Equation (6.2) for PCCP. Equation (6.3) uses a simpler version of the utilization bound test in [41] since we assume no more than one task per criticality level. It should be noted here that Equation (6.3) is an utilization bound as opposed to a recommended admission control test, since it makes the pessimistic assumption that zero-slack instants are 0 i.e. it degenerates to Criticality-As-Priority-Assignment (CAPA). In any taskset instance where zero-slack instants are greater than 0, the interference from lower-priority higher-criticality tasks ($\sum\limits_{\tau_l \in L_i^{hc}} C_l$) can be reduced appropriately.

Illustration

Consider the taskset shown in Table 6.3. A lower value for priority represents higher priority, and a lower value for criticality represents higher criticality. Let task τ_0 and τ_1 share a resource within a critical section that is protected by lock S_1. Let τ_1 and τ_2 share a resource within a

Task	C	C^o	T	D	Criticality	Priority
τ_0	10	70	100	100	2	0
τ_1	20	100	200	200	1	1
τ_2	40	200	400	400	0	2

Table 6.3: Example Taskset to Illustrate PCIP and PCCP

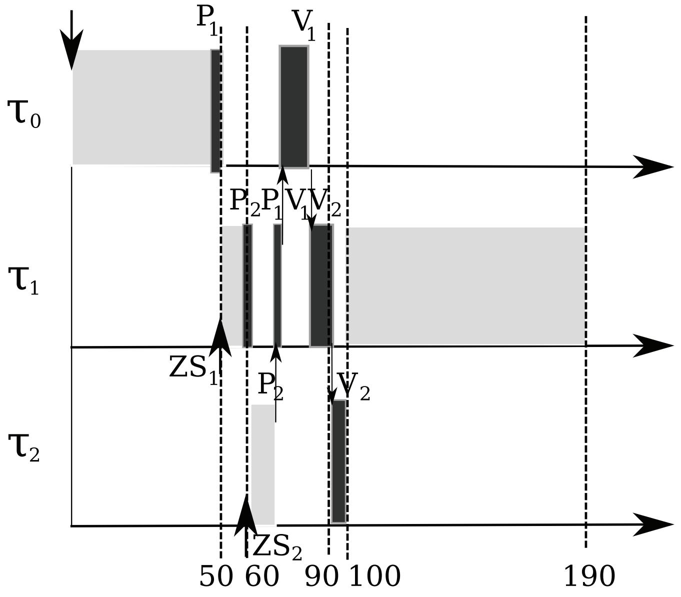


Figure 6.5: Illustration of PCIP

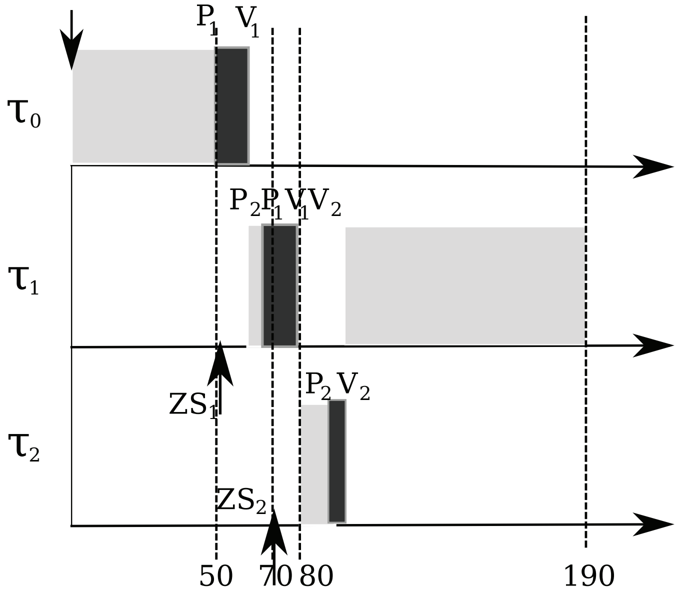


Figure 6.6: Illustration of PCCP

critical section that is protected by lock S_2. Let τ_1 first acquire the lock S_2 then acquire S_1 and release S_1, before releasing S_2. Let τ_0 execute its critical section for a duration of at most 10 time-units, τ_2 execute its critical section for a duration of at most 10 time-units, and τ_1 execute its nested critical section for a total duration of at most 10 time-units. Let P_i denote the request to acquire lock S_i, and V_i denote the release of lock S_i.

The application of PCIP to this example is shown in Figure 6.5 and PCCP is shown in Figure 6.6.

We will first illustrate the usage of PCIP in Figure 6.5. The blocking term for task τ_1 is 10 time units (since τ_1 is not required to execute any cycles in the normal mode in the worst-case and it can be blocked by τ_0 for 10 time units in critical mode). The blocking term for task τ_2 is 20 time units (since τ_2 is not required to execute any cycles in the normal mode even in the worst-case and it can be blocked for 20 time units by both τ_0 and τ_1 under criticality inheritance). Correspondingly, the zero-slack instant for τ_1 is 50 $(200 - 100 - 40 - 10)$, while that of τ_2 is 180 $(400 - 200 - 20)$. These zero-slack instants are due to the fact that from τ_1's perspective when τ_0 overloads no execution cycles may be available before time 50 (since $C_0^o = 70$), and from τ_2's perspective when τ_0 and τ_1 overload no execution cycles may be available before 180 (since $C_0^o = 70$ and $C_1^o = 100$).

Let us consider a scenario in which a job of task τ_0 (released at time 0) is executing at higher scheduling priority acquires the lock S_1. Consider a phasing of 0 for τ_1, under which the zero-slack instant of τ_1 occurs at time 50. It preempts τ_0 due its higher criticality and enters its critical section by acquiring a lock on S_2. Consider a phasing of -120 for τ_2, when the zero-slack instant of τ_2 occurs at time 60 (tasks τ_0 and τ_1 could be overloaded resulting in no cycles for τ_2 in interval $[-120, 60]$). τ_2 enters its *critical mode* and preempts τ_1, however, it needs the lock on S_2 to enter its critical section. Therefore, τ_1 inherits the criticality of τ_2 and continues its execution. However, when τ_1 needs S_1, it needs to block on S_1, therefore, τ_0 inherits the (inherited) criticality of τ_1 and continues its execution. When τ_0 completes the critical section protected by S_1, it returns control to τ_1, which in turn completes its critical section protected by the S_2 and returns control to τ_2. As this example shows, the higher criticality task τ_2 incurred a criticality inversion from τ_1 and a criticality inversion from τ_0, leading to a total blocking of 20 time-units in its critical mode. τ_1 on the other hand suffers a maximum of 10 time-units of blocking in its critical mode.

We now illustrate the usage of PCCP in Figure 6.6. Consider the same setup as before. The blocking term for task τ_1 is 10 time units (since τ_1 is not required to execute any cycles in the normal mode in the worst-case and it can be blocked by τ_0 for 10 time units in critical mode). The blocking term for task τ_2 is 10 time units (since τ_2 is not required to execute any cycles in the normal mode even in the worst-case and it can be blocked for 10 time units by either τ_0 or τ_1 under criticality ceiling). Correspondingly, the zero-slack instant for τ_1 is 50 $(200-100-40-10)$, while that of τ_2 is 190 $(400 - 200 - 10)$. Task τ_0 (released at 0) executing at higher scheduling priority acquires the lock S_1. On acquiring the lock, τ_0 inherits the priority ceiling 0 and criticality ceiling 1 of lock S_1. Consider the same phasing of 0 as before for τ_1, where the zero-slack instant of τ_1 occurs at time 50. When it attempts to preempt τ_0 due its higher criticality, the inherited criticality of τ_0 prevents τ_1 from preempting it. When τ_0 finishes its execution, it reverts back to its original criticality of 2. τ_1 now acquires the processor and continues its execution by acquiring the lock S_2 followed by lock S_1. By acquiring S_1, τ_1 inherits a priority of 1 and criticality of 0. Considering the same phasing as before of -120 for task τ_2, the zero-slack instant of τ_2 expires at time 70 (tasks τ_0 and τ_1 could be overloaded resulting in no cycles for τ_2 in interval $[-120, 70]$). At time 70, τ_2 cannot still preempt τ_1, which is executing at its criticality ceiling of 0. After τ_1 exits its critical section, it reverts back to its original criticality ceiling of 1, thereby enabling τ_2 to preempt τ_1. This leads to a total blocking of 10 time-units for τ_2 in its critical mode. τ_1 suffers a maximum of 10 time-units of blocking in its critical mode from τ_0.

6.4.3 Calculating the Zero-Slack Instant with Blocking Terms

As described earlier, under the zero-slack scheduling algorithm, jobs of each task τ_i are released in the *normal mode* and they switch to their *critical mode* at the last possible time instant known as the *zero-slack instant*. A key requirement in zero-slack scheduling is a mechanism to calculate the zero-slack instant of each task off-line, which ensures that tasks can meet their overload computation requirements as long as higher criticality tasks do not overload themselves. During runtime, jobs of each task switch to their *critical mode* if they have not finished by their respective *zero-slack instants*. In order to deal with the penalty arising from task synchronization, we have modified the zero-slack instant calculation algorithm as shown in Algorithm 3 (originally developed for independent tasks in [41]) including the blocking term B_i described in Section 6.4.2.

In Algorithm 3, note that B_i is a blocking term suffered by τ_i, which does not directly affect the schedulability of tasks with low priority or tasks with low criticality. Hence, the original execution times (C_i, C_i^o) of τ_i can be used to calculate vectors of available slack for tasks other than τ_i. This calculation of vectors of available slack ($GetSlackVector$) remains unchanged from the original algorithms presented in [41]. Calculating start of the trailing slack interval

Algorithm 3 GetSlackZeroInstantWithBlocking(i, D_i): Calculate the t_1 (before $t = D_i$) when slack of τ_i is 0

```
1: $V_i^n \Leftarrow GetSlackVector(i, \Gamma_i^n)$
2: $V_i^c \Leftarrow GetSlackVector(i, \Gamma_i^c)$
3: $C_i^c \Leftarrow C_i^o + B_i$ ; $C_i^n \Leftarrow 0$
4: **repeat**
5:     $t_1 \Leftarrow StartOfTrailingSlack(i, C_i^c, V_i^c)$
6:     **if** $t_1 \geq 0$ and $t_1 \leq D_i$ **then**
7:         $k_u \Leftarrow SlackUpToInstant(V_i^n, t_1) - C_i^n$
8:         $k_u = \max(\min(k_u, C_i^c), 0)$
9:         $C_i^c \Leftarrow C_i^c - k_u$
10:        $C_i^n \Leftarrow C_i^n + k_u$
11:    **else**
12:        $k_u \Leftarrow 0$
13:    **end if**
14: **until** $k_u = 0$
15: **return** $t_1$
```

($StartOfTrailingSlack$) and total available slack up to a time t_1 ($SlackUpToInstant$) are straight forward given a vector of available slack. For more details on calculating such vectors of available slack refer [41].

Optimization:

As described in Section 6.4, whether a task τ_i is blocked by τ_j really depends on the execution modes and scheduling attributes of both τ_i and τ_j. Depending on the possible task execution modes, some optimizations are possible in the zero-slack calculation.

In order to determine when it is possible to eliminate the blocking critical sections of a task, let us first analyze the role of the zero-slack instant in job scheduling. First, when a task τ_i's zero-slack instant is equal to its period, it is certain that τ_i will never execute in critical mode, and hence any blocking suffered in its critical mode can be eliminated. Secondly, if τ_i's zero-slack instant is smaller than its period but not zero, we need to determine whether it will run only in critical mode or in both normal and critical modes. When the zero-slack analysis calculation returns a zero-slack instant for τ_i such that, even under the worst case, τ_i is not required to execute any cycles in its normal mode, we can eliminate $\beta_{i,j}$ from all τ_j with lower priority but higher criticality than τ_i.

Blocking from $\beta_{i,j}$ can also be eliminated when τ_j is running in a mode that invalidates the blocking. In particular, when τ_i has a higher priority than τ_j but τ_j has a higher criticality than τ_i, and if we are certain that τ_j only runs in critical mode (zero-slack instant equal to zero), then we can consider $\beta_{i,j}$ to be empty.

The optimizations discussed above can be iteratively applied using the original algorithm. We first calculate the zero-slack instants with blocking that does not take into account the zero-slack instants. Then, the appropriate $\beta_{i,j}$ sets are adjusted and another iteration is performed. The optimization iteration is terminated when no new $\beta_{i,j}$ elements are eliminated.

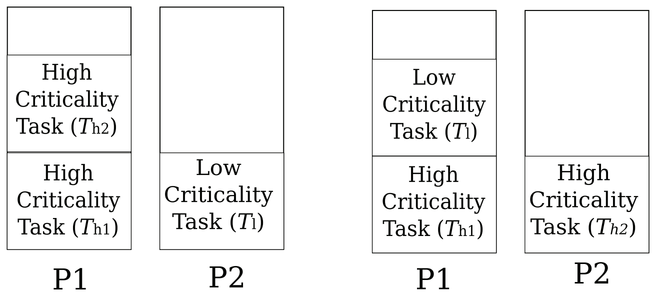
Leveraging the zero-slack rate-monotonic scheduling (ZSRM) algorithm for dealing with uniprocessor criticality inversion, we can now consider the task allocation problem in the context of multiprocessor mixed-criticality systems.

6.5 Task Allocation with Compress-on-Overload Packing

6.5.1 Criticality Inversion in Task Allocation

Within a processor, the criticality inversion problem can be addressed using ZSRM (as we will illustrated in Section 6.3), which guarantees that the low-criticality tasks cannot interfere with high-criticality tasks under overloads. Assuming ZSRM support, we first present the criticality inversion problem that arises during task allocation.

The criticality inversion problem also arises at the task-allocation phase in multiprocessors. A given allocation could favor a low-criticality task at the expense of a high-criticality one. For instance, consider three tasks τ_{h1}, τ_{h2}, and τ_l of high, medium, and low criticality respectively, each having a normal utilization $\frac{C_i}{T_i}$ of 35%. If we only have two processors P_1 and P_2, τ_{h1}



High Criticality Task (T_{h2})
High Criticality Task (T_{h1})
Low Criticality Task (T_l)
P1
P2
Low Criticality Task (T_l)
High Criticality Task (T_{h1})
High Criticality Task (T_{h2})
P1
P2

(a) Task Allocation that leads to Criticality Inversion

(b) Task Allocation that avoids Criticality Inversion

Figure 6.7: Illustration of Criticality Inversion in Task Allocation

is already deployed on P_1, and the three tasks do not fit on P_1 together, then we are forced to pack either τ_{h2} or τ_l on P_1 and the other to P_2. Packing τ_{h1} and τ_{h2} together, and τ_l by itself is an allocation decision leading to a criticality inversion (see Figure (a)). In this scenario, if τ_{h1} overloads, τ_{h2} may miss its deadline but τ_l will not i.e. our allocation decision protected τ_l (a low criticality task) at the expense of τ_{h2} (a medium criticality task). Conversely, deploying τ_{h1} and τ_l together and τ_{h2} by itself removes this criticality inversion (see Figure (b)).

As illustrated by the above example, allocating tasks to processors can introduce criticality inversion, which affects the system performance under overload scenarios. A criticality-aware task allocation algorithm can mitigate this problem by enabling more critical tasks to meet their deadlines under overload scenarios. In this section, we develop such a criticality-aware task allocation algorithm called Compress-on-Overload Packing (COP).

6.5.2 Implication to System Ductility

Let us now revisit the example scenario given in Subsection 6.2.4 to showcase the importance of packing and scheduling, consider the task allocation under which *Near Hostile* and *Far Friendly* are allocated to processor P_1, and tasks *Near Friendly* and *Far Hostile* are allocated to processor P_2. Assume also that ZSRM scheduling is used in each of the processors, which guarantees that tasks in any criticality level meet their deadlines as long as the higher-criticality levels are not overloaded (implicitly the highest criticality level is always guaranteed to meet its deadline when schedulable). We denote this scheduling scheme as Z.

The ductility matrix $D(Z)$ under this scenario is given by:

$$D(Z) = \begin{array}{l} w_1 = 3 =< 1,1 > \\ w_2 = 2 =< 1,0 > \\ w_3 = 1 =< 0,1 > \\ w_4 = 0 =< 0,0 > \end{array} \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

The ductility matrix is developed as follows:

- When $w = w_1 = 3 =< 1,1 >$, both criticality levels 1 and 2 are overloaded. Under ZSRM, the *Near Friendly* task (in criticality level 2) will miss its deadline in processor P_2 and *Far Friendly* task (in criticality level 2) will miss its deadline in processor P_1 ($d_{1,2} = 0$) and all other tasks will meet their deadlines ($d_{1,1} = 1$).
- When $w = w_2 = 2 =< 1,0 >$, criticality level 1 is overloaded. Under ZSRM scheduling, all tasks meet their deadlines ($d_{2,1} = d_{2,2} = 1$).
- When $w = w_3 = 1 =< 0,1 >$, criticality level 2 is overloaded. Under ZSRM scheduling, all tasks meet their deadlines ($d_{3,1} = d_{3,2} = 1$).
- When $w = w_4 = 0 =< 0,0 >$, both criticality levels are in normal conditions. Under ZSRM scheduling all tasks meet their deadlines ($d_{4,1} = d_{4,2} = 1$).

The ductility under ZSRM is given by $d(Z) = (\frac{1}{2} + \frac{1}{2^2}\frac{3}{4}) = 0.625$ since tasks in criticality level 1 meets their deadlines only under all overloads and gets a weight of $\frac{1}{2}$ (high criticality), while criticality level 2 meets its deadlines for $w < 3$ and gets a weight of $\frac{1}{2^2}$ (low criticality).

The normalized ductility $\nu(Z) = \frac{0.6875}{0.75} = 0.9167$ (since the maximum ductility for two criticality levels is 0.75).

As can be seen from this illustration, the scheduling scheme Z achieves a much higher *normalized ductility* (0.9167) than scheduling scheme R (0.5). This shows the importance of considering task criticalities during allocation and scheduling in mixed-criticality systems.

Given the above illustration of the importance of task allocation decisions, we now develop a criticality-aware bin-packing algorithm called *Compress-on-Overload Packing* (COP) to improve the ductility of partitioned ZSRM scheduling on multiprocessor systems.

6.5.3 Compress-on-Overload Packing

Traditional partitioned fixed-priority scheduling algorithms employ standard bin-packing heuristics such as Best-Fit Decreasing (BFD) and First-Fit Decreasing (FFD). For fixed-priority scheduling, it has been proven that both FFD and BFD have an utilization bound of $(m+1)(2^{\frac{1}{2}}-1)$ with rate-monotonic scheduling, on m processors [59]. In the context of mixed-criticality systems, the task allocation problem also needs to consider the task criticality during partitioning. As described in the earlier illustration, accommodating task criticality during allocation plays a key role in determining the system ductility under overloads. Ideally, it would be desirable to allocate each high criticality task to its own processor from the perspective of isolation from other tasks. However, this approach would be prohibitively expensive and unnecessarily waste resources. An alternative approach would be to leverage the scheduling guarantee provided by criticality-aware uniprocessor scheduling algorithm such as ZSRM, and design the task allocation algorithm around it.

Compress-on-Overload Packing (COP) is an algorithm that uses a two phased approach to allocating mixed-criticality tasksets. In phase I, from the perspective of system resiliency, it is desirable that as many high criticality tasks be guaranteed to meet their deadlines under any possible overload scenario. COP therefore sorts the tasks in the order of decreasing criticality (from high to low criticality), and uses only the overload execution budget C_i^o for the first phase. Within each criticality level, we use the Best-Fit Decreasing (BFD) heuristic for task allocation, since it is a known approximation of the optimal allocation for each criticality level. Any task that cannot be allocated in the first phase is kept aside. It should be noted here that under

Rate-Monotonic Scheduling and criticality-by-criticality Best-Fit Decreasing (BFD), these tasks would be deemed as unschedulable, however, we will accommodate these tasks in the second phase leveraging the scheduling guarantee provided by ZSRM. At the end of the first phase, all the allocated tasks are guaranteed to meet their deadlines irrespective of the system overload under consideration, since they are allocated using their overload execution budget.

Phase II uses the zero-slack scheduling algorithm presented in Section 6.3, which guarantees that each task meets its overload execution budget, as long as the higher criticality tasks do not exceed their normal execution budgets. Leveraging this guarantee, when using ZSRM for scheduling on the individual processors, we can handle any tasks that are unschedulable in the first phase. Again from the perspective of system resiliency, we wish to schedule and guarantee the deadlines of as many high criticality tasks as possible. Therefore, COP continues to consider the remaining tasks in decreasing order of criticality but uses both normal C_i and overload execution budgets C_i^o in the context of the ZSRM admission control algorithm. This is the *compression* phase where we pack additional tasks that would otherwise be unschedulable under conventional rate-monotonic scheduling. In this phase, we wish to distribute the load as much as possible and use up any available slack in the already allocated processors, therefore, we consider the Worst-Fit Decreasing (WFD) heuristic for allocation in phase II.

Given this description of COP, we can contrast its behavior with criticality agnostic bin-packing heuristics such as BFD. Algorithms such as BFD and FFD would pack as many tasks as possible without considering criticality, hence potentially resulting in a much lower system resiliency from the perspective of ductility. By considering the tasks in criticality order, COP guarantees that the resources are first made available to the high criticality tasks before any low criticality tasks are considered for scheduling.

6.5.4 Extensions to Compress-on-Overload Packing

The approach adopted in this chapter thus far has been the traditional partitioned scheduling without splitting. The key goal in here has been to describe the benefits of allocating the worst-case for the critical tasks and spreading the lower criticality tasks among the processor cores. This approach can still be extended with the task splitting algorithm in place, where the Best-Fit Decreasing heuristic can be replaced with the semi-partitioned task allocation. Phase II can still remain intact since the goal there is to spread the tasks among the processor cores as opposed to fully utilizing the processor cores. For synchronizing tasks, we can still adopt the approach of creating *composite* tasks with all the synchronizing tasks, required to be in the same criticality level, being considered as a single object for packing purposes.

A key characteristic of the Zero-Slack scheduler is its capability to split the normal execution segment from the critical execution segment. This feature can prove useful from the perspective of task splitting. While the approaches described earlier in this dissertation attempt to split a task at arbitrary locations, the presence of the zero-slack scheduler enables us to split the task during the interface between the normal execution mode and the critical execution mode. Therefore, the critical execution segments can be assigned their criticality as priority and allocated to a separate set of processor cores. The enforcement of the zero-slack scheduling guarantee happens automatically in such a scenario due to the properties of fixed-priority scheduling. However, we should consider the migration costs and preemption overheads during such splitting. Exploring this alternative task splitting approach for mixed-criticality task sets and comparing it with the standard approach to task splitting forms a key part of our future work.

6.6 Evaluation

The performance of mixed-criticality scheduling algorithms needs to be evaluated along two dimensions: (i) normal schedulability, and (ii) overload behavior. Classical bin-packing algorithms

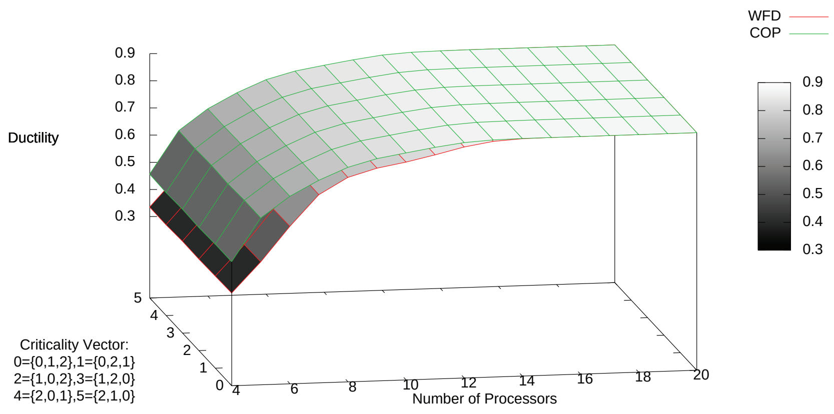


Figure 6.8: Surface of Average Performance (Harmonic Tasks)

for (non mixed-criticality) multiprocessor systems are typically evaluated exclusively along the dimension of normal schedulability. For any given taskset, the performance of different bin-packing algorithms along the dimension of normal schedulability can be compared by determining the number of processors required by each algorithm. However, in our case, we want to evaluate the effectiveness of the algorithms to extract the maximum ductility out of a given number of processors. Therefore, the ductility that the algorithms can obtain for different processor counts is compared. Our COP algorithm is compared to the WFD given that it is designed to balance the load, and hence the slack, across all the available processors.

Figure 6.8 shows the average ductility achieved using COP in comparison with the average ductility achieved using WFD (both using Zero-Slack Rate-Monotonic (ZSRM) within each individual processor). These results were obtained using randomly generated tasksets having 30 tasks each. In order to isolate the effects of bin packing from any rate-monotonic scheduling effects that may arise from non-harmonic task period ratios, we constrained our task sets to have harmonic task periods T_i from the set $\{100,200,400,800,1600\}$. The overloaded computation time C_i^o of each task was chosen in an uniformly random fashion between $\frac{1}{6}T_i$ and $\frac{1}{2}T_i$. Subse-

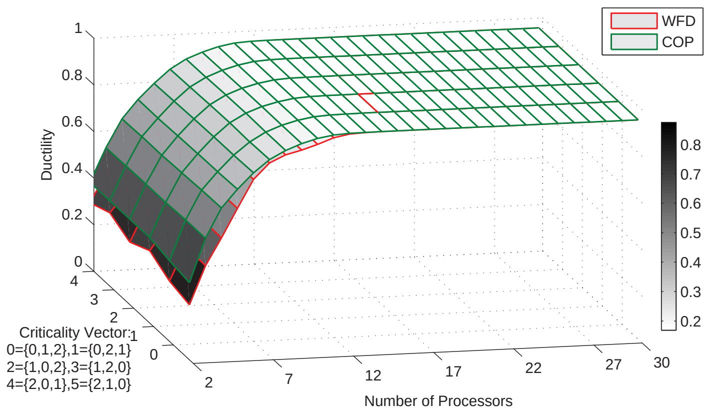


Figure 6.9: Surface of Average Performance (Task Periods: Uniform [10,100])

quently, the normal computation time C_i of each task was chosen in an uniformly random fashion between $\frac{1}{12}C_i^o$ and $\frac{1}{2}C_i^o$. We did not choose an overload utilization greater than $\frac{1}{2}$ since such tasks are typically allocated to their own processors. The overload workload was also restricted to be within a factor of two from the normal workload to focus on more stressful task sets. The tasks were also assigned to a criticality level in an uniformly random fashion from $\{L_1, L_2, L_3\}$.

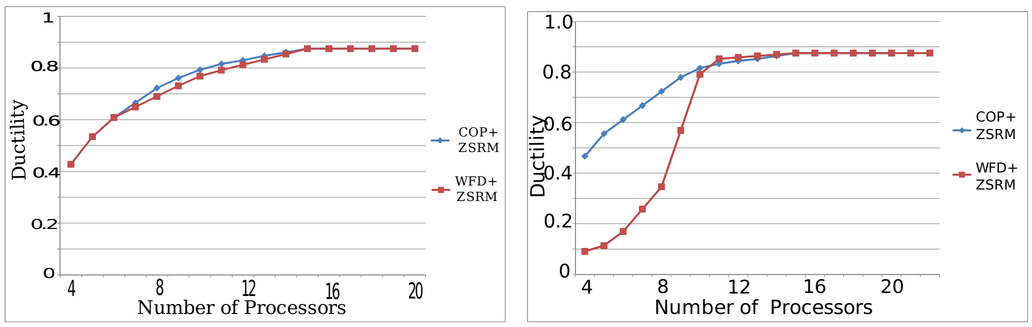
The criticality values for levels $\{L_1, L_2, L_3\}$ are varied along the x-axis as $\{0,1,2\}$,$\{0,2,1\}$, $\{1,0,2\}$,$\{1,2,0\}$,$\{2,0,1\}$, $\{2,1,0\}$. The number of available processors was increased from 4 to 20 along the y-axis. The z-axis presents the average ductility value (100 experiments for each data point) given the criticality assignment and number of available processors. Results in Figure 6.8 show that COP outperforms WFD significantly when the system has a fewer number of available processors. This behavior is largely due to the fact that WFD performs its allocation decisions in a criticality-agnostic fashion, thereby potentially packing high-criticality tasks in the same processor resulting in poor performance under overload conditions. COP, on the other hand, spreads the high-criticality tasks among the available processors, thus resulting in much better performance during system overloads. As the number of available processors increases, both COP and WFD are able to allocate more slack in each processor, which leads to better overload behavior. When the number of available processors is increased beyond 15 all tasks become schedulable even with their overloaded utilization $\frac{C_i^o}{T_i}$. Therefore, both COP and WFD achieve the maximum ductility of 0.875 (for three criticality levels, the maximum ductility is $1 - \frac{1}{2^3} = 0.875$).

Subsequently, we relaxed the constraint of harmonic task periods. We chose the task periods in a uniformly random fashion from $[10, 100]$. As with the previous experimental setup, we used randomly generated tasksets having 30 tasks each. The overloaded computation time C_i^o of each task was chosen in an uniformly random fashion between $\frac{1}{6}T_i$ and $\frac{1}{2}T_i$, and the normal computation time C_i of each task was chosen in an uniformly random fashion between $\frac{1}{12}C_i^o$ and $\frac{1}{2}C_i^o$. The obtained ductility values for COP and WFD at different criticality vectors and varying number of available processors is shown in Figure 6.9. As can be seen in these results, the behavior is quite similar to the case with harmonic task periods. However, under non-harmonic task periods, the obtained ductility values are observed to significantly change with the criticality vectors.

We now study the performance of COP and WFD on the specific taskset shown in Table 6.4. Each task of type τ_i was assigned to criticality level $\{L_i\}$, and the criticality vector $\{L_1, L_2, L_3\}$ was varied as before. Figure (a) shows the performance at criticality assignment $\{0, 1, 2\}$. In this scenario, the task criticalities are assigned to task priorities. As shown in Figure (a), both COP and WFD exhibit very similar performance since there is no criticality inversion. However, under a criticality assignment of $\{2, 1, 0\}$, the taskset experiences maximum criticality inversion since the criticalities are exactly in the reverse order of priorities. Figure (b) shows that the COP achieves significantly better performance compared to WFD when a small number of processors is available (almost five-fold in extreme cases). As the number of processors increases, the performance difference between COP and WFD decreases. When the number of available processors approaches a large enough value that is sufficient to schedule the overloaded tasksets themselves, WFD performs slightly better than COP. This is largely due to the approximate nature of the heuristics themselves. COP uses a modified BFD algorithm in its first phase, which can perform worse than WFD for specific tasksets. This shows that although the average-case

Table 6.4: Task Types

Task	C	C^o	T
τ_1	10	50	100
τ_2	20	100	200
τ_3	30	200	400



(a) Comparison at criticality vector $\{0,1,2\}$ (b) Comparison at criticality vector $\{2,1,0\}$

Figure 6.10: Comparison at different criticality vectors

performance of COP as shown in Figure 6.8 seems to indicate that COP always outperforms WFD, there do exist tasksets and processor counts at which WFD performs better than COP.

Our evaluation results show that COP performs better for mixed-criticality systems compared to traditional WFD, by taking into account both task criticality and sizes. We illustrated this using the ductility metric developed in Section 6.2. Based on the evaluation, COP is best suited for mixed-criticality systems where (i) there are fewer number of processors than required to schedule the overloaded taskset itself, and (ii) criticality of tasks are misaligned with their priorities.

Table 6.5: Deadline Misses

Packer	Deadline misses		
	2300 Tracks	2400 Tracks	2500 Tracks
WFD	0	$\frac{9}{37}$ Far Hostile	$\frac{16}{23}$ Far Hostile
COP	0	$\frac{9}{87}$ Near Friendly	$\frac{28}{49}$ Near Friendly

6.6.1 Application to Radar Surveillance

The radar-surveillance simulation setup was hosted on a Lenovo laptop with the Intel Core 2 Extreme Processor having four cores clocked at 2.526 GHz each, 32KB Instruction Cache, 32KB Data Cache, and 6MB Unified L2 Cache. There are four available processing cores P_1,P_2,P_3, and P_4. The *Track Generator*, *Track Classifier*, and *Track Search* tasks were allocated to processing core P_3, and the processing core P_4 was dedicated for display. For simplicity of illustration, we restrict our attention to the allocation and scheduling of the *Near Hostile*, *Far Hostile*, *Near Friendly*, and *Far Friendly* tasks among processing cores P_1 and P_2.

Performance Evaluation

The summary of our performance evaluation is shown in Table 6.5. Under Worst-Fit Decreasing (WFD) task allocation and Zero-Slack Rate-Monotonic (ZSRM), the *Near Hostile* and *Far Hostile* tasks are assigned to processor P_1, and the *Near Friendly* and *Far Friendly* tasks are assigned to processor P_2. As seen in Table 6.5, as the system workload increases beyond the maximum normal workload of 2300 tracks, the WFD packing scheme results in *Far Hostile* missing deadlines due to the overloading of *Near Hostile* on processor P_1. This behavior is clearly not desirable since the resource allocation decision of assigning *Far Hostile* to P_2 would not only protect *Far Hostile*, but also enable it to steal cycles from other tasks under overload. In terms of the ductility metric, this results in deadline misses at criticality level 1, which has a high weight of $\frac{1}{2}$.

Under Compress-on-Overload Packing (COP) and Zero-Slack Rate-Monotonic (ZSRM) scheduling, the *Near Hostile* and *Near Friendly* task are allocated to processor P_1, and the *Far Hostile* and *Far Friendly* tasks are allocated to processor P_2. As seen in Table 6.5, as the system workload increases beyond the maximum normal workload of 2300 tracks, the COP packing scheme results in *Near Friendly* missing deadlines due to the overloading of *Near Hostile* on processor P_1. This is much better behavior since *Far Hostile* was protected from the overload behavior of *Near Hostile*. In terms of the ductility metric, this results in deadline misses at criticality level 2, which has a lower weight of $\frac{1}{4}$. In our experimental surveillance setup, the *Near Hostile* task overloaded before the *Far Hostile* task, therefore, the *Far Friendly* task did not miss any deadlines on processor P_2.

As can be seen from the radar surveillance example, allocating high-criticality tasks by spreading them across the processing cores enables them to expand under overload, resulting in much better system *ductility*. Compress-on-Overload Packing (COP) therefore achieves higher *ductility* value than conventional criticality-agnostic task allocation heuristics.

6.7 Summary

Mixed-criticality tasks introduce interesting challenges in emerging cyber-physical systems, where multi-core processors can be effectively leveraged. The overload behavior plays a vital role in such systems, as shown by our radar surveillance case study. In this chapter, we formally captured the desired overload behavior of mixed-criticality systems using the *ductility* metric. Systems with higher ductility must guarantee that, under overload conditions, the high-criticality tasks continue to meet their deadlines by stealing resources from low-criticality tasks. We first developed a Zero-Slack (ZS) scheduling algorithm to provide high ductility in uniprocessor settings. We then showed that task allocation decisions also play a vital role in determining system ductility for multi-core settings. Subsequently, we developed the Compress-on-Overload Packing (COP) algorithm for allocating tasks to processors in order to improve system ductility. Evaluation results show that ZS subsumes both (i) the rate-monotonic scheduling (RMS) priority assignment used for maximizing schedulable utilization, and (ii) the Criticality-As-Priority Assignment (CAPA) algorithm used for better overload behavior. From a task allocation perspective, COP is shown to strictly dominate the standard worst-fit decreasing (WFD) heuristic used for load balancing. In resource-limited settings, COP can achieve up to five times better ductility than WFD. Finally, we applied our solution to the radar surveillance application and illustrated the practical benefits of using criticality-aware scheduling and task allocation.

Chapter 7

Distributed Resource Kernel Framework

In this chapter, we describe our Distributed Resource Kernel (RK) framework, which was used to implement the algorithms mentioned in chapters 3 through 6. This framework not only supports multi-core processors but also extends the notion of Resource Kernels to provide resource management in distributed multi-core systems. The main motivation behind this implementation is to enable easy adoption and evaluation of the ideas presented in this dissertation in real-world application pipelines. The implementation of the algorithms presented in chapters 3 and 4 are described in Section 7.5, along with implementation considerations for realizing the fork-join scheduling model discussed in chapter 5.

The rest of this chapter is organized as follows. We first introduce the resource kernel paradigm that we build upon. Next, we develop the design goals of our framework. Following which, the task graph model used to represent application pipelines in our framework is described. The actual design and implementation of the framework then follow. We finally provide an evaluation of the framework along with measurements of task splitting overheads and the implementation costs for our RT-MAP library based on `pthreads`.

7.1 Resource Kernels

Resource kernels [90, 99] represent resource-centric approaches for building real-time kernels that provide timely, guaranteed and enforced access to system resources. The applications using the system can specify their resource demands and the kernel assumes the responsibility of satisfying the specified demands using its own resource management policies. The kernel also guarantees that the demands will be met throughout the lifetime of the application by performing admission control tests before admitting any new application into the system, and enforcing the applications that overrun their pre-specified demands. Thus, the key features of the resource kernel architecture are the use of a uniform resource model for dynamic sharing of different resources, and the provision of fine-grained timing guarantees and temporal isolation through admission control and enforcement.

Resource kernels have been shown to provide guaranteed and timely access to various resources including processor cycles [83], disk bandwidth [107], network bandwidth [53] and physical memory [44]. The core first-class resource management entities introduced in resource kernels are:

1. A **Reserve** represents a share of any available resource in the system. Applications in the system make use of the available resources through reserves. The resource can be any shareable resource including processor time, memory, network bandwidth, disk bandwidth or physical memory.
2. A **Resource Set** represents an aggregation of reserves on various resources. This aggregation, to which applications can attach themselves, serves as the basic resource management unit. Resource allocation policies are enacted at the granularity of resource sets.

7.2 Design Goals For Distributed Resource Kernels

In this section, we describe the major goals of our framework. Along with each goal, we summarize how our framework strives to accomplish the goal. The goals are roughly in the order of importance: earlier goals are fundamental to the framework, while the subsequent goals are necessary for practical acceptance, and for ease of programming, diagnostics, adoption and usage.

7.2.1 End-to-End Deadline Guarantees

The major motivation behind our framework is to enable real-time application pipelines to meet their end-to-end deadline requirements. This can only be made possible by doing appropriate resource management and scheduling at each multi-core node, using the principles described in this dissertation, along with the control of timing interfaces across processor boundaries. We describe the distributed RK approach to meeting end-to-end deadlines in more detail in section 2.3.

7.2.2 Temporal Resource Isolation

Timing guarantees provided by the framework to each independent application should hold good irrespective of the behavior of other applications, given that other applications may be spawned dynamically or mis-behave at run-time. Hence, the framework supports end-to-end admission control and enforcement mechanisms to ensure that the timing behavior of a task is not compromised by the behavior of the other tasks. High-resolution timers are employed to provide fine granularities of temporal isolation. Dynamic modification of resource reservation parameters is also feasible.

7.2.3 Powerful End-to-End Application Abstractions

Each application in the system may use a number of nodes with different resource demands on multiple resource types, including processor cycles, network bandwidth, disk bandwidth and physical memory. Abstractions that collect these requirements into a single entity and present the view of a virtual isolated operating environment, are very desirable. This goal is closely related to the previous goal of achieving temporal resource isolation, since each application can ignore the others sharing the system and the underlying distributed infrastructure guarantees such isolation. Such a programming abstraction will also greatly facilitate application development, while the temporal isolation support will significantly ease testing requirements and serve as the containment boundary for timing faults. Our framework proposes an abstraction called the distributed resource container, which collects the distributed resource reservations on multiple nodes and presents a unified view to the application. This abstraction decouples the application requirements from the resource management policies used by the system. Therefore, extensions to our framework along the dimensions of security and fault-tolerance can easily leverage this abstraction in the future.

7.2.4 Efficient System Utilization

An important goal of this framework is to utilize resources efficiently and effectively. In other words, the admission control tests and enforcement mechanisms used, should obtain provably good resource utilizations at acceptably low overheads. The framework leverages the scheduling approaches presented in this dissertation to meet this goal at the individual node level. However, the actual allocation of tasks to cores is performed by user-space resource managers, which are built on top of our framework. The allocation tools can leverage the bin-packing algorithms developed in this dissertation, in addition to any placement constraints specified by the application itself.

7.2.5 Remote Resource Management

For scalability and ease of administration, the framework should enable all resource management operations to be carried out remotely. The resource management API should be uniform and transparent to the locations of the resources being managed. This enables the administrators to view the overall system as a collection of resources under their control. The resource managers in the individual nodes should present a view of the unallocated resources (abstracting away the already allocated resources) thereby enabling the administrators to view the resources that are still free and therefore usable. Our framework has utilities for remote creation, deletion, modification and deletion of application threads and associated distributed resource containers.

7.2.6 Lightweight Runtime Monitoring

The administrator of the system has to be able to monitor the resource utilization levels, of all the deployed applications. The resource usage statistics may be useful for diagnostic and performance monitoring tools, which monitor the health of the system. Utilization levels can also be used to track potential bugs in the applications. In order to perform such run-time debugging, the framework has to support light-weight monitoring of the entire system. In our framework, the individual nodes are responsible for monitoring their resources and aggregation is performed at the administrator node on a need-basis.

7.2.7 Ease Of Application Deployment

Deploying applications can be cumbersome process in large-scale distributed multi-core real-time systems. Pinning application tasks to processor cores is already supported in the Linux environment using the `sched_setaffinity()` interface. It would therefore be convenient to automate the process and have application deployment utility as a part of the framework, which can read the subtask allocation to resources from a configuration file, remotely create the corresponding reserves, attaching them to processor cores, and aggregating them onto distributed resource containers automatically.

We next describe the distributed task graph model used to represent distributed real-time applications for our framework.

7.3 Task Graph Model

Real-time application pipelines may be represented using task graphs, where each application is divided into a set of communicating sub-tasks. This division happens at the logical-boundaries of the applications, such that each sub-task waits for one or more inputs from other sub-tasks, performs local computation and generates one or more outputs. The dependences between these sub-tasks can now be represented as edges, to obtain the distributed task graph (a simple task graph for sequential and otherwise independent tasks is shown Figure. 7.1). This representation is analogous to control-flow graphs commonly used in code-representation, here the basic-blocks can be compared to sub-tasks and the actual control-flows correspond to communication dependencies. In our model the graphs are acyclic, since it would otherwise not be possible to analyze the end-to-end timing guarantees without analyzing the source code and the run-time inputs, which may govern the number of number of times the control flows along each loop in the graph.

Given a distributed task graph, the next step is to obtain a sub-task to processor core allocation. Most real-world systems are distributed either because of the physically distributed nature of the problem being solved (e.g. sensor-actuator systems) or to exploit parallelism and redundancy(e.g. modern ship-board computing platforms). In heterogeneous scenarios, the allocation is predominantly decided by the resources (e.g. sensing tasks are allocated to sensor nodes, signal processing tasks are alloted to DSPs). In homogeneous scenarios, parallelism is usually the goal and the allocation decisions are predominantly governed by the timing properties of the sub-tasks. The sub-task to processor core allocation problem is therefore solved using the techniques

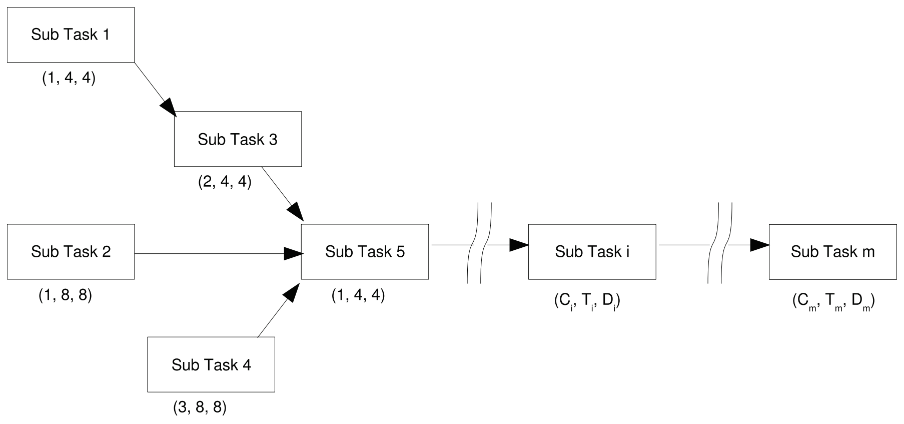


Figure 7.1: Distributed Task Graph Model

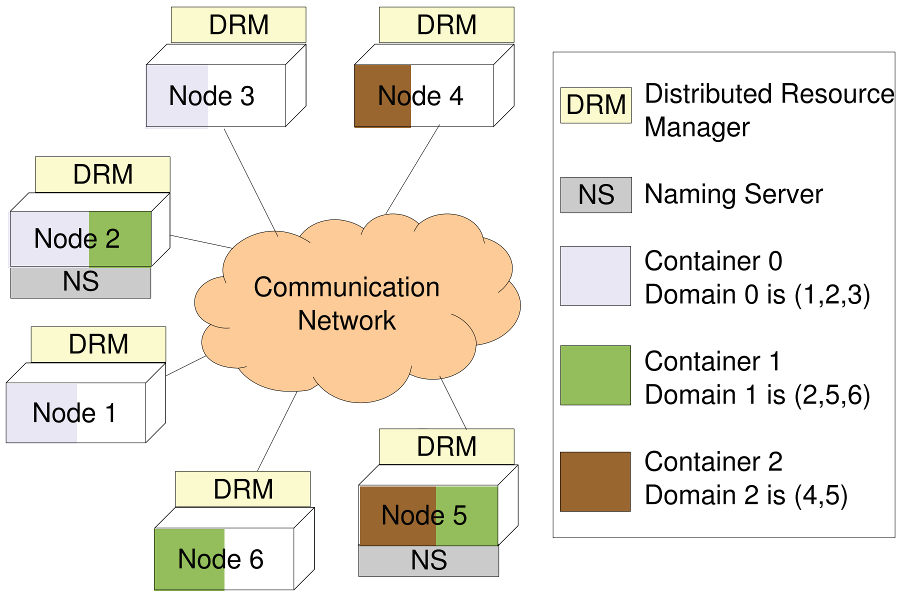

DRM
Node 3
DRM
Node 4
DRM
Node 2
NS
Communication Network
DRM
Node 1
DRM
Node 6
DRM
Node 5
NS
DRM Distributed Resource Manager
NS Naming Server
Container 0 Domain 0 is (1,2,3)
Container 1 Domain 1 is (2,5,6)
Container 2 Domain 2 is (4,5)

Figure 7.2: Example deployment for the Distributed Resource Kernel Framework

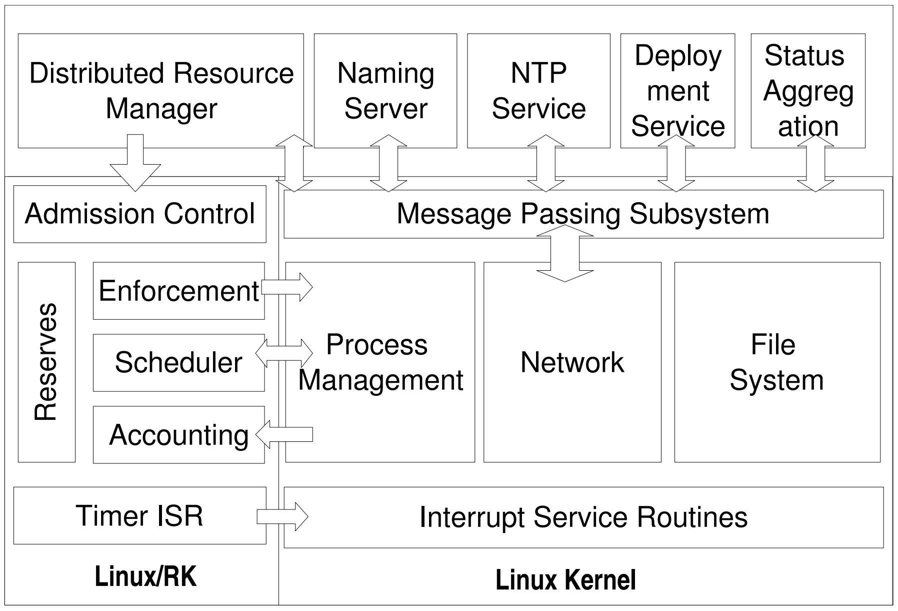


Figure 7.3: Local architecture of the Distributed Resource Kernel Framework

described earlier in this dissertation. A wealth of other literature also exists on the task allocation problem in distributed real-time systems (for example [57, 121]) on task graphs similar to the one we have described. The focus of our framework is to provide end-to-end timing guarantees and temporal isolation, given the distributed task graph and its corresponding task allocation.

We now describe the architecture of our framework and how its various sub-systems achieve the previously mentioned goals.

7.4 Distributed RK Architecture

The architecture of the Distributed RK framework is shown in Figure. 7.3. The major components of the framework are described next.

7.4.1 Distributed Resource Manager Daemons

Every node in the system has a resource manager daemon that listens for resource reservation requests from its counterparts. This daemon is responsible for all the tasks allocated to processor cores in its purview on the same node. Whenever a reservation request is received, it is evaluated, and if the admission control succeeds on the specified processor core, the resource is reserved. A reserve handle is piggybacked along with the reply to the requesting resource manager. In order to present a uniform interface for resource reservation, the framework identifies each reserve by the resource address and automatically dispatches the reservation request to the corresponding resource management daemon.

The resource manager daemon is thus the most important component of the distributed RK framework, it is responsible for:

- Managing the processor cores present in the local node
- Coordinating with the resource managers on other nodes to remotely reserve resources, and
- Aggregating the remotely managed resource reservations into a distributed resource container and registering it with the name service (discussed next)

It is desirable for fault-tolerance and scalability purposes that there be no central command and control center. There can also be multiple administrative domains within the same system, sharing the resources, and therefore having a single administrator node is not practical. It was therefore required that the distributed RK framework be built to support multiple co-operating administrator nodes. This led to development of distributed resource manager daemons, which can seamlessly enable systems with different administrative setups.

7.4.2 Name Service

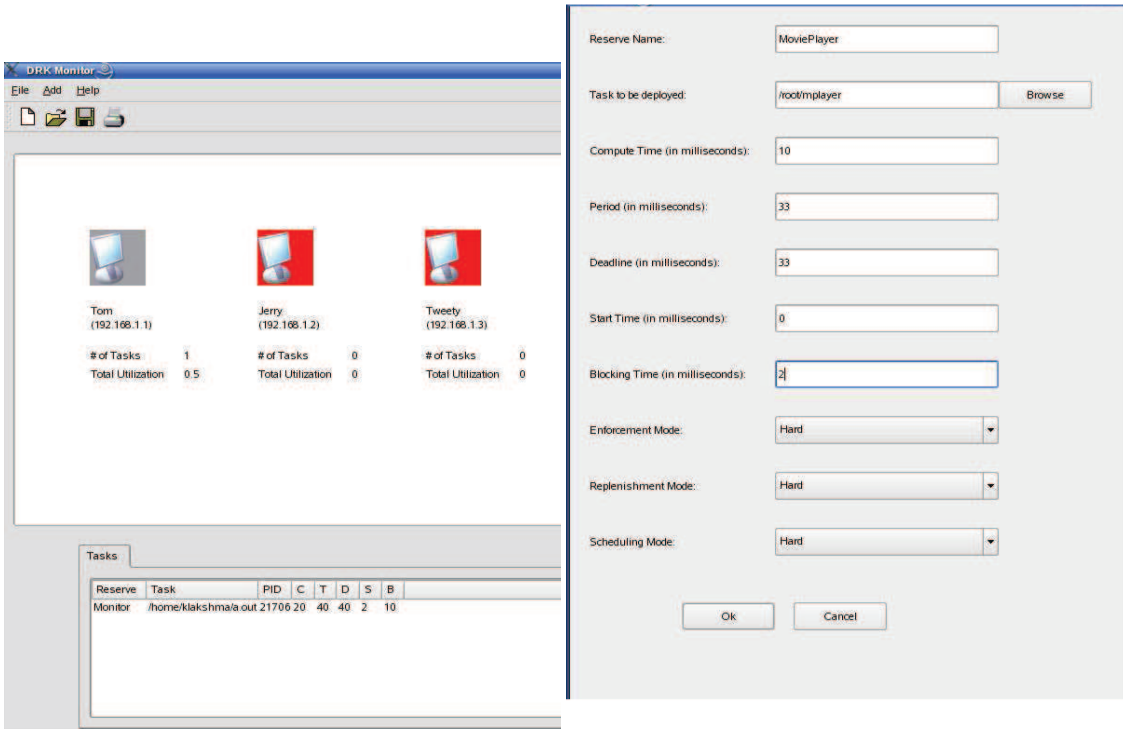
Location-transparency and Portability are key requirements for the distributed RK framework. Subtasks in the distributed system need not be aware of their host resource or any location information. Such transparency enables our framework to be extended easily for fault-tolerance purposes through replication and migration. In order to support such a transparent operation, it is required that each application creates a distributed resource container with a unique id and the sub-tasks of the application attach themselves to the container with this id. It is to be noted here that such an id will be decided during the build-time of the individual applications. When the sub-tasks are deployed on different nodes, they can use this globally unique id to query a name server provided as a part of the distributed RK framework, and inherit the resources allocated to the application. The id can also be used to query the location of other sub-tasks of the application and set up the communication paths during application initialization.

The naming service is provided by a group of name servers running in the system. The name servers have a globally partitioned set of ids for which they are responsible. Name servers advertise their presence periodically and the resource managers can query them, whenever they wish to resolve a global id. When application are to cleaned up, the name servers are notified to update their entries. The global ids used in our framework are currently alphanumeric strings of fixed length.

7.4.3 Message Passing Subsystem

The distributed resource-manager daemons need to coordinate with each other frequently during the system operation. The message passing subsystem acts as an efficient kernel-level communication framework to enable such co-ordination. It is lightweight, and currently uses datagrams for point-to-point communication. Each message has the following format:

1. **Message Type:** Specifies the command requested by the message (e.g. remote resource set creation)



Tom (192.168.1.1)

# of Tasks	1
Total Utilization	0.5

Jerry (192.168.1.2)

# of Tasks	0
Total Utilization	0

Tweety (192.168.1.3)

# of Tasks	0
Total Utilization	0

Reserve	Task	PID	C	T	D	S	B
Monitor	/home/klakshma/a.out	21706	20	40	40	2	10

Reserve Name:	MoviePlayer
Task to be deployed:	/root/mplayer
Compute Time (in milliseconds):	10
Period (in milliseconds):	33
Deadline (in milliseconds):	33
Start Time (in milliseconds):	0
Blocking Time (in milliseconds):	2
Enforcement Mode:	Hard
Replenishment Mode:	Hard
Scheduling Mode:	Hard

Figure 7.4: GUI Frontend with provisions to (1)add Nodes, (2)add applications and (3)monitor system status

2. **Sequence Number:** Used to discard duplicates and manage buffers. A timeout mechanism is built into the sender and receiver state machines.
3. **Length:** Identifies the payload size of the message
4. **Payload:** Specifies the parameters needed to carry out the command specified by the message type or the return values from the result of the command. The format of the payload varies with message type

The message passing subsystem in our framework is similar to the other message passing systems like MPI [60]. The major difference in our architecture is that our communication framework is implemented at the kernel-level rather than the user space. Our primary intent was to reap the performance benefits of the kernel-level implementation obtained by various other servers like kHTTPD [15]. We did not use a connection-oriented protocol since the framework was primarily designed to communicate very small commands with limited number of arguments that usually fit in a single packet. The reliability of the communication infrastructure in our target environments are also envisioned to be orders of magnitude more than the Internet. This custom message-passing subsytem therefore brings in a performance benefit for the relatively frequently communicated messages for co-ordination between resource managers.

7.4.4 Deployment Utility

The deployment manager is a utility that can automatically deploy one or more end-to-end applications on the nodes of the system, on to the specific processor cores. A system configuration file (shown in Table. 7.5.5) describing the allocation of tasks, obtained from the actual allocation engine, is used by the deployment manager. The utility loads each application described in the configuration, automatically creates a distributed resource container, creates the remote resource sets on the required nodes, aggregates them into the distributed resource container and automatically spawn the sub-tasks of the application on the different nodes. When the applications exit, the deployment manager is notified and it automatically cleans up the resource sets and associated distributed resource containers. The deployment utility thus enables the resource management layer to be operate independent of the applications in the system. The GUI frontend presents a global view of the system, with facilities to add new nodes, application pipelines and monitor existing ones. (See 7.4)

7.4.5 Status Aggregation Engine

The status aggregation engine is used to monitor the resource utilization statistics. The utilization information is exported by local resource kernel on the individual nodes. The message passing infrastructure is then used to export this information on a need-basis, to the administrator nodes. Various performance monitors and diagnostic tools can benefit from such global system-wide status information. Unusual patterns in resource utilization levels are commonly used to detect timing bugs in applications deployed in the system.

7.4.6 Global Time Service

Distributed RK provides a global time service through a separate time daemon. A client for this service is spawned in each of the nodes and they listen for messages from the time servers. This service is important to synchronize all the nodes in the distributed system and hence incur less penalty in scheduling, during period enforcements. Global timing information is also useful for distributed applications to compute *global event-orders*. Our time service closely resembles the commonly used NTP [84] service for synchronization over the Internet, except for our simplifications and optimizations towards the target network topologies with limited span.

7.5 System Implementation

We have implemented the distributed resource kernel framework described in the previous section. Our current implementation runs on X86 architectures using the Linux/RK platform as a building block. Linux/RK is available as a kernel module for a minimally modified version of the 2.6.32 kernel.

Our current implementation of distributed resource kernels supports only remote reservation of processor cycles (CPU Reservations). In the future, we will also provide support for network and disk bandwidth reservations.

Our message passing subsystem consists of the following message types: (1) Advertisement messages - Sent by the name server (2) Request messages - Sent for creating, deleting and modifying resource reserves, resource sets and distributed resource containers (3) Reply messages - Success or Error messages corresponding to the requests. The message passing framework supports the following wrappers over the kernel implementation of the UDP protocol: `rk_create_socket()`, `rk_bind_socket()`, `rk_release_socket()`, `rk_recv()` and `rk_sendto()`. This API is used to communicate between the various distributed resource manager daemons.

The distributed resource manager daemon is implemented as a kernel thread *rk_local_rm* and encapsulated within the Linux/RK module as a single module. The *name server* and *global time service* are available as separate modules. The name server advertises its presence using the advertisement messages and maintains the naming registry as a table hashed by distributed resource container ids. The time service is a collection of local *global time service* daemons, one for each node in the system. These time daemons are used to correct the local clocks, based on the synchronization service provide by time servers in the system. The `rk_get_current_time()` can be used by applications to obtain the global time as maintained by the service.

7.5.1 Task Splitting Implementation

Resource sets in the resource kernel architecture are bound to specific processor cores. This enables the kernel to perform standard fixed-priority scheduling admission control tests on the processor core, while admitting reservations and provide accurate per-period accounting of processor usage. Given this reservation model, we now describe the implementation of task splitting in resource kernels.

Tasks that are split across processor cores are attached to the appropriate reservations on the individual processor cores. When a split task exhausts its reservation on the first processor core, it automatically gets assigned to the next available reservation on the successive processor core. Therefore, fixed-priority task splitting is easily realized using the underlying resource kernel infrastructure.

For example, consider a task that requires a computation time of 4 milliseconds with a period of 10 milliseconds. Using the PDMS_HPTS approach, let this task (C, T, D) be a split task across processor core $P1$ and $P2$, as $(2, 10, 10)$ and $(2, 10, 8)$. Observe that the highest priority nature of this task guarantees its completion within 2 milliseconds on processor $P1$, hence it is sufficient for the second sub task to have a deadline of 8 milliseconds. In order to realize this splitting, a reservation with an execution capacity of 2 milliseconds and period 10 milliseconds is created on processor $P1$. Another reservation of capacity 2 milliseconds and period 10 milliseconds is created on processor $P2$ with a deadline of 8 milliseconds. During the actual runtime execution, when the task exhausts its capacity on processor $P1$, it gets enforced, and the next available reservation on processor $P2$ is picked up. During this transition, the `sched_setaffinity()` call is invoked to migrate the task to the corresponding host processor core for the next available reservation. When the job completes, during its `rt_wait_for_next_period()` call, it gets migrated back to the first reservation on its list.

7.5.2 RT-MAP Library Implementation

With respect to the synchronization protocols, we have implemented *MPCP:Suspend* and *MPCP:Spin* as a part of our RT-MAP (Real-Time Multi-core Application Package) Library , which is built around `pthreads` [77] for Linux.

Task allocation is performed using the `pthread_setaffinity_np()` API. The CPU-affinity masks are modified to reflect the allocation of threads CPUs. We use the built-in real-time thread scheduling support in Linux, using the `pthread_setschedparam()` interface. Our implementation provides wrappers to create real-time threads, perform admission control on created threads, and allocate tasks using our synchronization-aware scheme.

The synchronization schemes are implemented as special data-structures with priority queues, which are internally protected using the `pthread_mutex_lock()` and `pthread_mutex_unlock()` interfaces. The priority of the task is raised to the highest priority level before accessing the synchronization data-structures, in order to mask preemptions during updates to the priority queue.

7.5.3 Implementation Considerations for Fork-Join Task sets

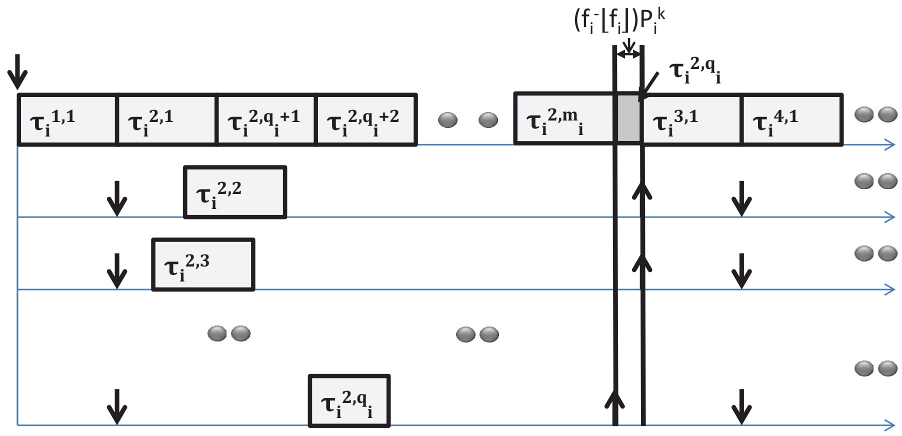


Figure 7.5: Implementation Considerations for Fork-Join Tasks.

As mentioned in Chapter 5, the task *stretch* transform can be accomplished by the OS scheduler, and does not require any changes to existing code. The master string of each task τ_i with $C_i > T_i$ must be assigned its own processor core. This can be easily accomplished in many standard operating systems, for example, using the `sched_setaffinity` call which tells the scheduler to run a task on a particular core or processor. The parallel threads τ_i^{2s,q_i+1} through τ_i^{2s,m_i} need to be coalesced with the master string, and this can be realized by assigning these threads to the same core as the master string. The processor assignment for parallel threads $\tau_i^{2s,2}$ through τ_i^{2s,q_i} $\forall 1 \leq s \leq (s_i - 1)/2$ can be determined by a task partitioning algorithm such as the one we describe in the next section. These threads can also be *pinned* to their cores using the `sched_setaffinity` call. The only parallel threads that need special treatment are τ_i^{2s,q_i}, since all other parallel threads fully execute on the same core. The thread τ_i^{2s,q_i} needs to have a *budget* of r_i^{2s} on its assigned processor, upon exhausting which it needs to be migrated to the core assigned for the master string (see Figure 7.5). This can be readily implemented on our Linux Resource Kernel platform.

7.5.4 Distributed Resource Kernel System Calls

Table. 7.1 lists the current list of system calls supported by the distributed RK framework. This table also includes a brief description of each supported system call. Return values and error conditions are not included for sake of brevity.

7.5.5 Application Deployment In Distributed RK

We now illustrate the typical sequence of steps involved in developing and deploying an application using the distributed RK framework.

(1) First, a resource container is created using the `rk_distributed_resource_container_create()` system call. This system call hands control to the distributed resource manager daemon, which creates an entry in the local registry (maintained as a hash table) and sends a registration request to the name server. The calling task is suspended until a response is received from the name server. Once the name server responds, the local registry is updated and the calling task is awakened by the resource manager.

(2) The next step is to create resource sets in the remote nodes to scope and use the resource on those nodes. The `rk_distributed_resource_set_create()` system call is used to accomplish this objective. The distributed resource manager uses the distributed resource container handle and talks to the remote resource managers to create the resource sets. The resource set handle stored in the local registry and a copy of the handle is returned to the calling task.

Table 7.1: Distributed Resource Kernel API

1. `containerHandle rk_distributed_resource_container_create (name)`	Creates a new distributed resource container with the specified `name` and registers it with the Name server for uniqueness.
2. `int rk_distributed_resource_container_destroy (containerHandle)`	Cleans up the resource allocated for the container corresponding to `containerHandle` and Notifies the Name server.
3. `resourcesetHandle rk_distributed_resource_set_create (containerHandle, name, address)`	Remotely creates resource set with given `name` at given location (`address`) and aggregates it under the `containerHandle`
4. `int rk_distributed_resource_set_destroy (resourcesetHandle)`	Cleans up reserves allocated as a part of the remote resource set.
5. `reserveHandle rk_distributed_cpu_reserve_create (resourcesetHandle, reserveAttributes)`	Creates remote reserve under `resourcesetHandle` using `reserveAttributes` (`computeTime`, `period`, `deadline`, `blockingTime`, `startTime`, `reserveType`) Reserve type is HARD, SOFT or FIRM for replenishment & enforcement (Fig. 7.7)
6. `int rk_distributed_cpu_reserve_destroy (reserveHandle)`	Used to destroy the remotely created cpu reserve pointed by `reserveHandle`
7. `reserveHandle rk_distributed_cpu_reserve_modify (reserveHandle, reserveAttributes)`	Modifies the attributes of the remote cpu reserve pointed by `reserveHandle`, with the `reserveAttributes`.
8. `int rk_distributed_resource_set_attach_process (resourcesetHandle, processid)`	Attaches a process with the given pid (`processid`), running on remote node to `resourcesetHandle`
9. `int rk_distributed_resource_set_detach_process (resourcesetHandle, processid)`	Detaches a process with the given pid (`processid`), running on remote node from `resourcesetHandle`.
10. `void rk_get_current_time (timeAttribute)`	Returns current global time value as a 64bit unsigned integer . in `timeAttribute` for synchronization purposes.

(3) The reserves with the exact parameters are created using the `rk_distributed_cpu_reserve_create()` system call. The resource set handle is used to obtain information about the location of the remote resource set and the distributed resource manager daemon then co-ordinates with the corresponding manager to create the reserve. The reserve is then linked with the remote resource set.
(4) The final step is to attach the sub-task process to the resource set created in the above steps. This can be accomplished using the `rk_distributed_resource_set_attach_process()` system call. In order to perform this operation, the process id of the task being attached remotely should be known. This information is propagated from the remote resource manager beforehand.
(5) When an application exits, the corresponding cleanup system calls are executed to recover the allocated resources.
We present some sample code in Table. 7.5.5 as an example of the above initialization sequence and the real-time processing loop of an application. In summary, the system calls of the distributed RK framework are designed to facilitate easy deployment, maintenance and management of distributed real-time applications.

7.5.6 Distributed Hartstone Benchmarks

In our next set of experiments, we ran key Distributed Hartstone (DH) Benchmarks [82] to quantify the real-time scheduling capabilities of our Distributed RK framework. For our DH experiments, we used two nodes whose configurations are shown in Table. 7.2.
The DH benchmarks were developed specifically to evaluate the performance of distributed real-time systems. This is accomplished through stressing the different parts of the OS subsystem that are critical for real-time tasks. The breaking point of the system, at which deadlines are missed is noted as the performance metric value for the system under evaluation. The idea behind these tests is that the entire system is only as strong as the weakest sub-system. The core workloads of the DH benchmarks specified in 1990, are dated and have to be scaled up for use in modern architectures that have significantly higher performance. The workload in the original DH

Sample Deployment Code For Distributed RK

```
...
cpu_attr_p11->compute_time.tv_nsec=10000000;
cpu_attr_p11->period.tv_nsec=80000000;
cpu_attr_p11->deadline.tv_nsec=80000000;
cpu_attr_p11->blocking_time.tv_nsec=0;
cpu_attr_p11->start_time.tv_nsec=0;
...
dcnt = rk_distributed_resource_container_create(
             "my_test_resource_container");
rst[0] = rk_distributed_resource_set_create(
             dcnt, "resource_set_p11", "192.168.1.1");
rsv[0] = rk_distributed_cpu_reserve_create(
             rst[0], cpu_attr_p11);
...
rst[1] = rk_distributed_resource_set_create(
             dcnt, "resource_set_p12", "192.168.1.2");
rsv[1] = rk_distributed_cpu_reserve_create(
             rst[1], cpu_attr_p12);
...
rk_distributed_resource_set_attach_process
(rst[0],pid1);
rk_distributed_resource_set_attach_process
(rst[1],pid2);
...
rk_distributed_cpu_reserve_destroy(rsv[0]);
rk_distributed_cpu_reserve_destroy(rsv[1]);
rk_distributed_resource_set_destroy(rst[0]);
rk_distributed_resource_set_destroy(rst[1]);
rk_distributed_resource_container_destroy(dcnt);
...
```

Sample Application Code For Distributed RK

```
while (application_processing_not_done)
{
    ... /* Do the application processing here */

    rt_wait_for_next_period();
}
```

Example Code for Using Resource Containers and Reservations

```
my_test_resource_container

resource_set_p11 192.168.1.1 1
application_11
cpu
compute_time      10000000
period            80000000
deadline          80000000
blocking_time     0
start_time        0

resource_set_p12 192.168.1.2  1
application_12
cpu
compute_time      30000000
period            80000000
deadline          80000000
blocking_time     0
start_time        0
```

Example Configuration File

Table 7.2: Setup For DSH Tests

Node_1	PentiumD, 2793.082MHz, 2046KB cache, 1GB Main Memory, DELL Ethernet Controller
Node_2	AMD Athlon XP 2400+, 1997.207MHz, 256KB cache, 1GB Main Memory, nForce2 Ethernet Controller

Table 7.3: DSH Task Sets

Task	**Workload**	**Period (DSHcl)**	**Period (DSHpq)**
τ_1	4.2MWS	80ms	160ms
τ_2	4.2MWS	160ms	320ms
τ_3	8.4MWS	320ms	640ms
τ_4	8.4MWS	640ms	1280ms
τ_5	33.6MWS	1280ms	2560ms
τ_{server}	variable	N/A	N/A

benchmarks were developed to evaluate the ARTS kernel on a SUN3 platform which 4.2ms to execute a synthetic workload of 1 kilo-Whetstone. We had to scale this workload to 4.2 Million Whetstones, reflecting a performance increase of 4200!

DSHcl Series: Communication Latency

The task set for this benchmark is given in Table. 7.3. As shown, the system has five client $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$ and a server task τ. All the client tasks were hosted on $Node\_2$ and the server task was hosted on $Node\_1$ in our experiment setup. Each client sends a request to the server, the non-preemptible server computes a synthetic workload for a specific time and then responds. The client suspends itself till the response is received. Once the response is received, the client computes its corresponding workload specified in MWS (as shown in Table. 7.3). All these stages have to be completed before the task deadline. This task set is used to test the communication latency of the system. Prioritization of requests is done at the application level by the server task.

In our experiment setup, the nodes were connected by a 100Mbps Ethernet link with a round-trip network latency of about 0.2ms. The workload of the server was increased until it caused one of the tasks to miss a deadline. We found that the deadlines were missed by task τ_1 when the computation time of the server was increased to 37ms. Thus the performance of the distributed RK platform on this benchmark is DSHcl-37.

The reason that the task τ_1 misses the deadline is that when the server execution time is 38ms, there are cases when τ_2, τ_3, τ_4 or τ_5 begin executing at exactly the same instant when τ_1 is released. When τ_1 sends a request to the server, it may have to wait for 38ms since the server is non-preemptible and then it takes another 38ms for its request to be serviced, leading to a total of 76ms, which when followed by the 4.2ms computation delay causes it to miss the deadline of 80ms.

This result can be compared to the performance metric of DSHcl-35 reported for the ARTS kernel in [82]. The reason for the extra 2ms time possible in our testbed is that service times have decreased tremendously due to modern processing speeds. From this experiment, we can conclude that the distributed RK framework performs efficient and appropriate real-time scheduling.

DSHpq Series: Priority Queueing

The next set of experiments tests for the priority queueing of communication packets, attempting to show the advantage of priority queuing over conventional FIFOs used in communication. The task-set for this experiment is given in Table. 7.3. It should be noted here mthat we refer to a basic workload of 4.2 Million whetstones whereas in the original benchmark this was in kilo-Whetstones. The setup for this experiment is the same as that for the DSHcl benchmark series described in the previously. The task-set workload was scaled until the first deadline was missed. Under the distributed RK framework, deadline were missed when the server workload was 18 times the workload of 4.2 MWS, resulting in DSHpq-18. This result is the same as that reported for the ARTS kernel. The first deadline is missed due to the same scenario described in the previous section. In this case, the server time could be increased to 75.6ms due to the scaled task periods (Table. 7.3). Priority queuing at the application level does not affect the performance of the system.
In conclusion, Distributed Hartstone Benchmarks demonstrate the capability of distributed RK provide prioritized real-time scheduling across processor and communication boundaries.

7.5.7 Temporal Isolation

Our next goal was to evaluate the temporal isolation support provided by the distributed RK framework. We designed two applications such that, the first application has a reservation of 20ms of CPU time every 40ms each on $Node\_1$ and $Node\_2$. The second application has a reservation of 20ms every 120ms on $Node\_1$ and 40ms every 120ms on $Node\_2$. To maximize potential interference between tasks, we attached infinite loop processes to each of these reservations. These applications were deployed using our automatic deployment framework and the results were monitored using our status aggregation engine. If there were no temporal isolation, one of these tasks will never get the chance to execute in a purely fixed-priority-based preemptive scheduling environment.

Figure. 7.7 presents the CPU consumption on $Node\_1$ (left column) and $Node\_2$ (right column) by these two tasks. Each point in these graphs represents the fraction of cycles obtained by the application sub-task during its most recent reservation period. We studied the performance of the system under both hard reserves and soft reserves. Hard reserves represent the policy where the application will be strictly enforced after its CPU reserve, whereas in soft reserves the application will also be allowed to execute when no other real-time tasks are utilizing the processor.

It can be seen from the results that application 1 obtains 50 percent on $Node\_1$ (as P11) and 50 percent on $Node\_2$ (as P12), corresponding to its reserve of 20ms every 40ms. Application 2 obtains roughly 17 percent on $Node\_1$ (as P21) and 33 percent on $Node\_2$ (as P22) under hard reserves. When application 2 was attached to soft reserves, it can be seen that it steals the extra cycles whenever no other application is utilizing the processor.

The applications as mentioned before, were infinite loops and this behavior would not be seen in other purely priority based real-time operating systems, since the tasks would never suspend themselves. The observed behavior illustrates the distributed enforcement capabilities of distributed RK and hence its ability to provide temporal isolation.

7.5.8 Audio Processing Pipeline

We finally conducted a set of experiments to further validate the temporal isolation properties and deadline guarantees, provided by the distributed RK framework for practical applications. Towards this goal, we created an audio processing application that is pipelined across three nodes

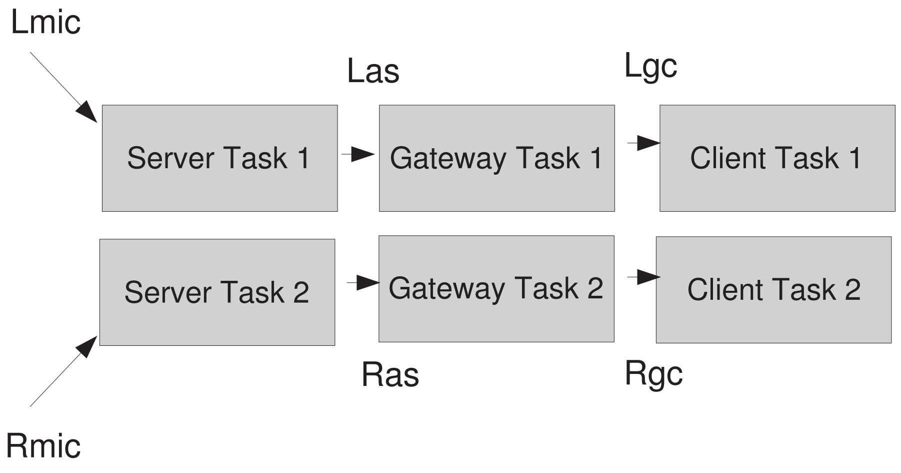


Figure 7.6: Audio Processing Pipeline Experiment Setup

managed by distributed RK. The pipeline, shown in Figure. 7.6, consists of an audio serer which reads in two different audio streams, forwards them to a gateway, which in turn delivers the content to a client node. The audio server was exclusively dedicated to the streaming service, while the gateway and client had other applications sharing the processing cycles. The audio signal was sampled at 8KHz, with 8 bits per PCM sample. Several hundred samples were batched together for each transmission.

The end-to-end deadline for each stream was set at 300ms and the individual subtask deadline was set as 80ms. The remaining 60ms is to accommodate the network delays and time synchronization bounds. Buffers along the pipeline were modified to reflect these timing constraints, such that the buffers would be overwritten whenever newer data arrives and the older data has not be dequeued. The behavior of this setup was compared with 3 nodes running the standard Linux kernel. The results of our comparison are shown in Figure. 7.8. The graphs on the left column correspond to stream 1 and right column correspond to stream 2. The top row corresponds to the audio stream as seen at the server (first pipeline stage). The second row shows the streams seen at the gateways under standard Linux (no RK) and the third row shows the audio stream received by the clients running Linux (no RK). It can be clearly seen that when standard

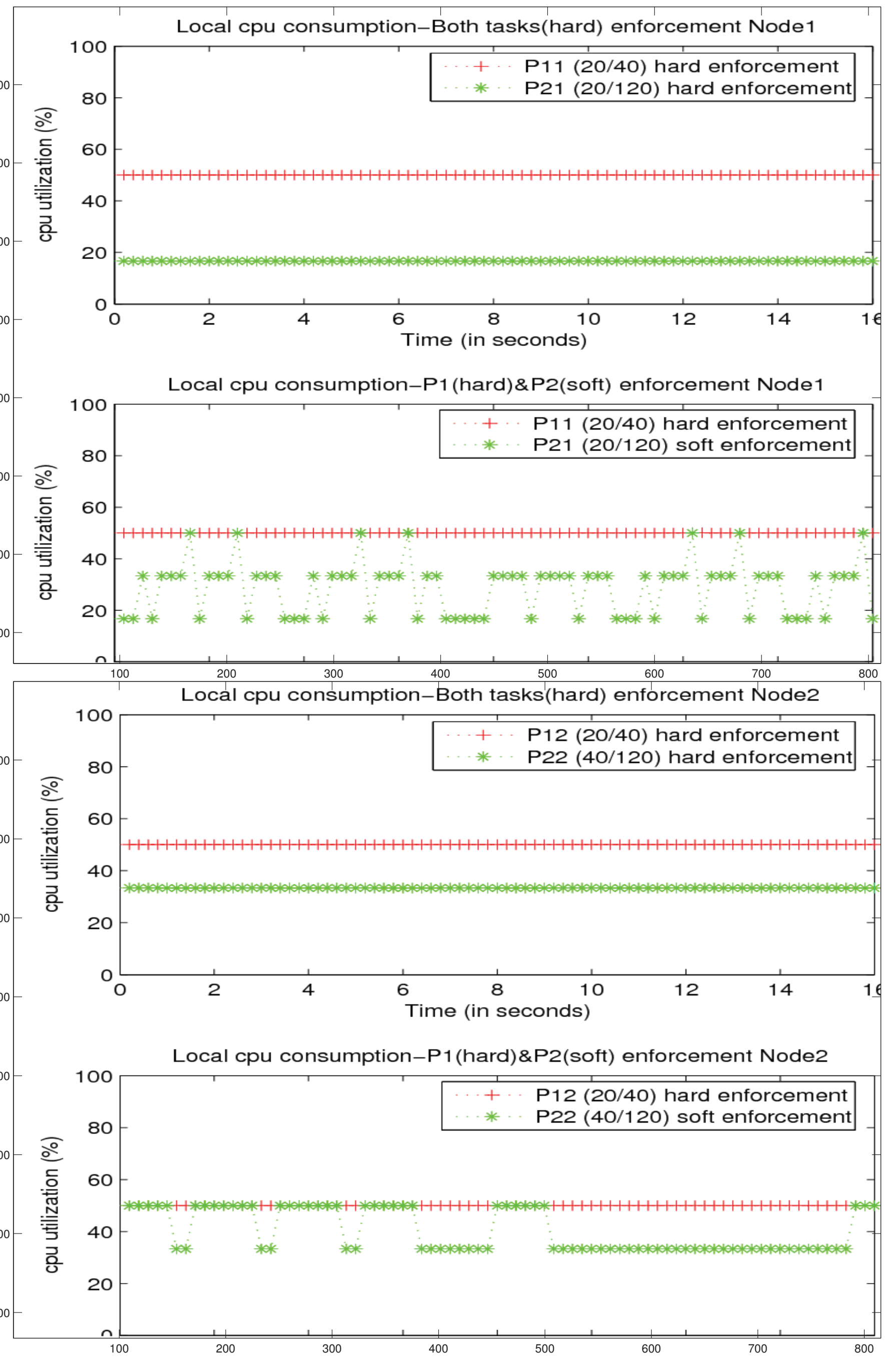


Figure 7.7: Behavior of Hard (top) and Soft (bottom) enforcement schemes in Distributed RK

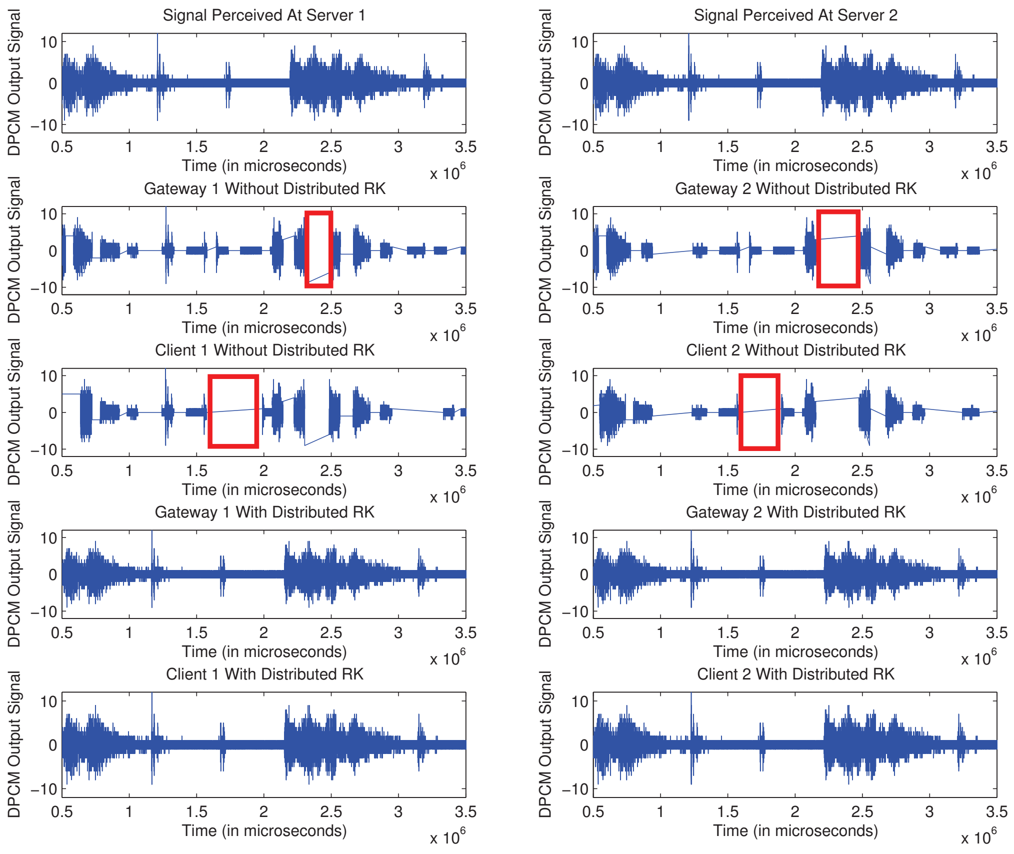


Figure 7.8: Performance Evaluation with the Audio Processing Pipeline

Linux is used, each audio stream exhibits discrete pauses, when the other stream or application gets scheduled. The Linux kernel does not inherently understand the timing properties of the audio streams and therefore cannot guarantee its end-to-end timing constraints. The results obtained with the distributed RK framework are shown in the fourth and fifth rows. The reservation parameters were set to be (30ms, 80ms, 80ms) for (C,T,D) and the audio streams were analyzed. It was seen that the packets were received and processed without any pauses or bursty packet drops, despite the presence of other reserved and unreserved applications.

The end-to-end delays were measured as follows. Each application was time stamped at the source(using the globally synchronized clock source) and the subtasks dropped any packets the exceeded their end-to-end deadlines. We found that a sizeable fraction of packets missed their deadlines with standard Linux, but not with the distributed RK reservations in place. Thus, these experiments demonstrate the capability of the distributed RK framework to provide end-to-end deadline guarantees and provide temporal isolation.

We therefore conclude that the distributed RK framework performs efficient and appropriate real-time scheduling with temporal enforcement and provides end-to-end timing guarantees.

7.5.9 Case Study of Task-Splitting

In this section, we present a case study of task migration on the Intel Core 2 Duo processor to characterize the practical overheads of task-splitting. The processor has 2 cores with a private L1-caches and a shared L2-cache. The L1-cache (64KB) is a split cache with both Instruction and Data caches having a size of 32KB each. The L2-cache is a unified cache of size 4MB. Both L1 and L2 are on-chip cache resources, and the cache line size is 64 bytes. There are 512 cache lines in the individual L1 caches (32KB). The access time for L1 cache is about 3 cycles, while the access time for the L2 cache is anywhere from 11 to 14 cycles. The data bus between the L1 and L2 cache has a width of 256 bits.

In order to understand the impact of task migration on cache performance, we evaluated a series of synthetic cache-workloads. These workloads had varying working-set sizes (from 1KB to 64KB) and stride-lengths (1 to 64bytes). Performance of these workloads is shown in Figures 7.9(a) and 7.9(b).

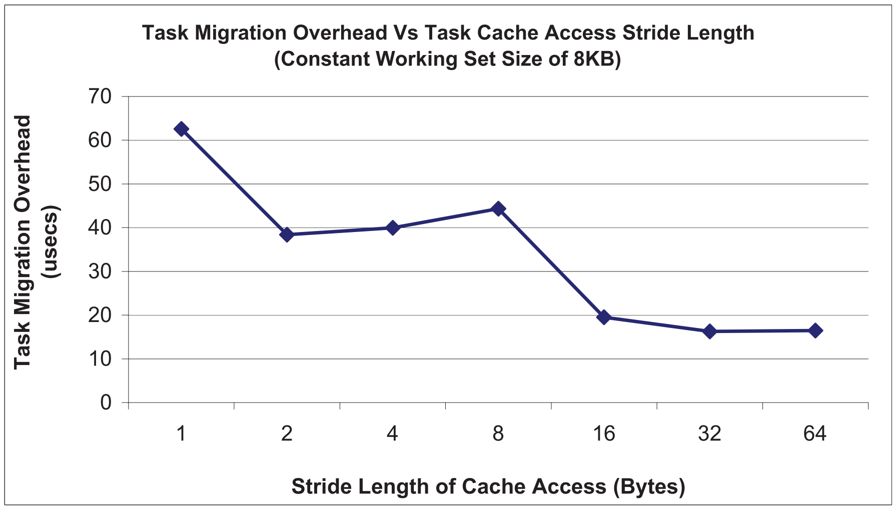
The overall overhead was acceptably low (less than 80 microseconds) for these cache-workloads. These workloads exercised only the data cache, and the instruction cache effects were neglected in these experiments. The timing measurements were done at a fixed processor frequency of 2GHz.

The results show that the overheads are generally lower at smaller working set sizes, as expected. At lower stride lengths, the cache overhead of task-splitting is higher. As the stride length increases, the performance difference introduced by task-splitting diminishes. When a split-task migrates from one core to another, it has to re-create the cache state on the new core. Task with spatially sparse access patterns will load its entire cache state much faster than those with sequential access patterns. Most modern cores provide extensive support for out-of-order execution, load-store queues, and cache pre-fetching, which reduce the impact of task migration. In such cores, tasks with spatially sparse access patterns generate multiple parallel requests to different cache lines, whereas, sequential access patterns result in stalling on the same cache line.

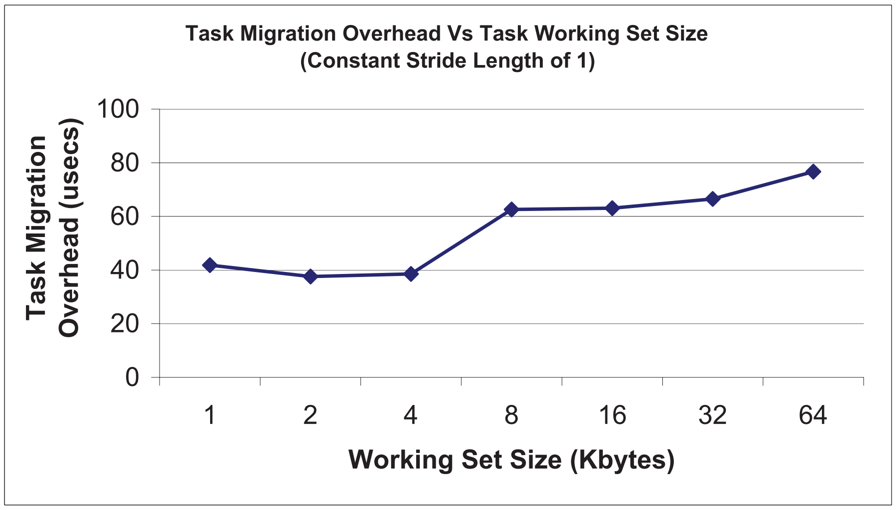
This analysis was done with R/W access patterns; however, similar behavior was also observed with read-only access. Although not shown here, tasks with low temporal locality will also have lower cache overheads due to task splitting.

In an effort to characterize the impact of task migration on some real-world applications and benchmarks, we looked at the media player application called *MPlayer* and the standard OpenGL *Gears* benchmark in isolation. The results of these experiments are shown in Table 1. The overhead due to task-splitting was negligible in both the cases.

MPlayer was scheduled such that the $decode\_frame$ module gets split across the two cores at exactly mid-way through the processing. The FFmpeg library was used and a wmv3 file was being played. The core computational loop in the video player is comprised solely of $decode\_frame$ in addition to minor accounting updates. Without task splitting, the average time



(a) Migration overheads *vs.* Cache access stride lengths



(b) Migration overheads *vs.* Working-set sizes

Application	*Time Taken (ms)* *w/o Task Splitting*	*Time Taken (ms)* *w/ Task Splitting*
MPlayer	5.79	5.84
GL Gears	4.5	4.5

Table 7.4: Task splitting overhead on real-world applications

Table 7.5: Cost of primitives used in RT-MAP

Primitive	**Avg. Time (μs)**	**Max. (μs)**	**Min. (μs)**
Priority Switching	2.98	5.84	0.92
Lock Request	2.1	6.62	0.03
Unlock Response	0.09	0.28	0.05

taken by a single call to $decode\_frame$ was 5.79ms. After performing task splitting, the average time taken for a $decode\_frame$ call was increased by approximately 0.05ms.

The reason for the overhead of about 50μs in the case of *MPlayer* is probably the fact that it incurs heavy cache overheads. The video was being played at approximately 25 frames/second (a period of 40ms), and this cache overhead translates to 0.125% in terms of the *decoder* task utilization. It is to be noted here that the overhead numbers include both OS overhead in migrating the task, as well as cache-overheads.

There was no real overhead seen with the *Gears* application. This is due to the fact that *Gears* has a very low working set size. Most of the time is spent in rendering the image in the video buffer and the cache is not utilized very frequently.

The evaluation of *MPlayer* and *Gears* shows that the cache overheads due to task-splitting can be expected to be negligible in multi-core platforms. Hence task-splitting is a practical mechanism of improving the overall system utilization in partitioned real-time multi-core scheduling.

7.5.10 Implementation Costs for Task Synchronization

We implemented RT-MAP, a `pthreads`-based library for real-time support in multi-core processors. Task allocation and synchronization support are provided through easy-to-use APIs. The implementation costs of the underlying primitives are listed in Table 1. The results are from

1000 measurements performed on an Intel Core i7 Duo processor at 3.4GHz.

The task allocation algorithm is only performed during initialization, and it therefore does not affect application performance. The values listed in Table 1 only reflect the cost of updating the underlying scheduler data-structures. The major penalty arises from the cost of preemption. However, the true cost of preemption is determined by the effect of preemption on cache content, which is application-specific and depends on the working set-size of application threads.

A basic primitive used in our implementation is that of priority switching. It is required twice per global critical section, once to elevate the task priority and once to restore the task priority to normal. In our implementation, each thread suspends/spins on its own local data-structure and the task releasing the lock is responsible for notifying the appropriate pending thread. The implementation cost is thus evenly distributed between the lock request and the unlock response. Although our implementation does not have the benefits of an in-kernel implementation, the prevalent use of `pthreads` in multi-threaded applications enables us to transparently support a wide-range of applications. Other than the `pthread_setaffinity_np()` for CPU binding, all the other interfaces are expected to be readily portable.

In all our experiments presented in the earlier section, we do not consider any additional preemption costs (caused due to both loss of cache state and context switching overheads). This is due to multiple reasons: (i) we do not wish to bias the results towards our particular implementation, (ii) preemption costs are highly dependent on the particular architecture under consideration - for instance, a multicore system with tightly-coupled memory has an extremely low preemption cost compared to a cache-based multicore processor, and (iii) context-switching costs vary widely across operating system implementations (as low as tens of cycles in ThreadX). Implementation costs, however, cannot always be ignored for a given system. We therefore examined the impact of various preemption costs on the different execution control policies.

Spinning suffers from fewer preemptions compared to the suspension-based scheme. We, therefore, investigated the tipping point at which implementation costs favor *MPCP:Spin* over *MPCP:Suspend*. For each additional job released in the system, both schemes incur an additional

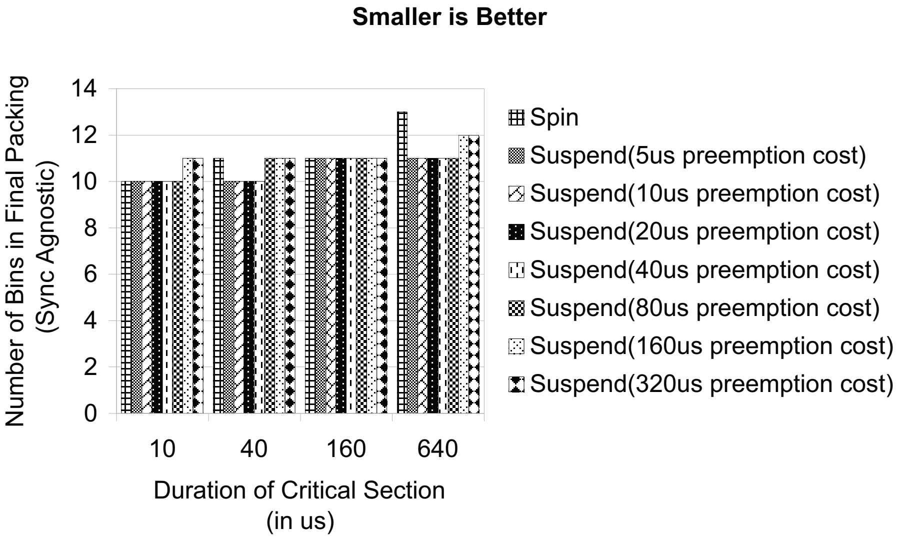


Figure 7.9: Impact of Preemption Costs (at different critical section lengths)

preemption cost. This baseline cost, therefore, is assumed to be accounted in job worst-case execution times. *MPCP:Suspend*, however, incurs additional preemptions during each remote-blocking interval.

Our experimental results for various preemption costs are shown in Figure 7.9, for a workload of 8 fully-packed processors, 2 critical sections per task, and 2 lockers per mutex. For critical sections longer than 40μs, even a 320μs preemption cost does not favor spinning over suspension. On the other hand, at lower preemption costs, *MPCP:Suspend* performs much better. Architecture and application characteristics therefore play a vital role in choosing the execution control policy.

In order to quantify the impact of the preemption overhead on tasks with more critical sections per task, we conducted the experiments summarized in Figure 7.10. In order to magnify the effect, a 5μs critical section was chosen and two different preemption costs (45μs and 200μs) were considered. The workload was maintained a constant at 8 fully-packed processors and 2 lockers per mutex. As can be seen, with more critical sections per task, the preemption costs tend to significantly affect the suspension-based scheme.

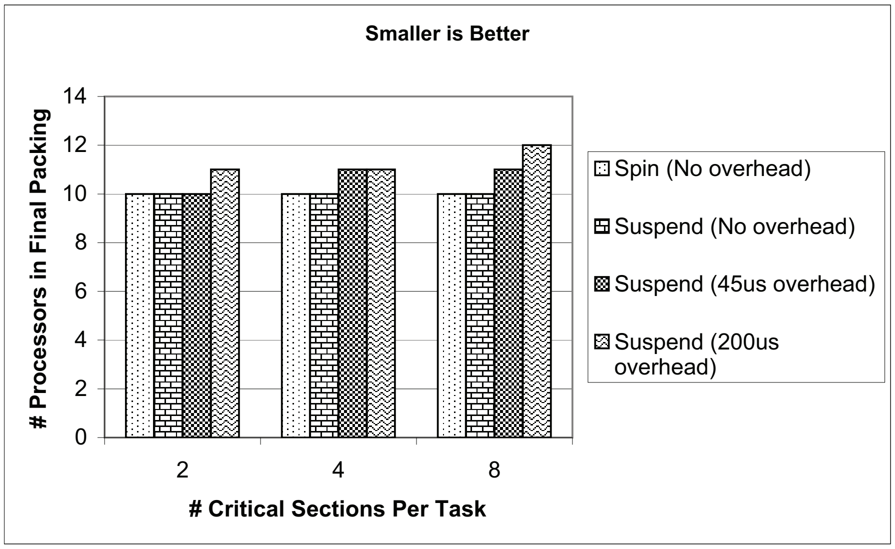


Figure 7.10: Impact of Preemption Costs for a 5μs critical section (with increasing number of critical sections per task)

Although we have empirically shown the high preemption costs and low critical section durations for which *MPCP:Spin* outperforms *MPCP:Suspend*, in practice, such situations are rare. Assuming symmetric read/write cache bandwidths, it is unlikely that a 5μs preemption by a lower priority task's global critical section evicts 45μs worth of cache state. Many application-level critical sections also tend to be much longer than those found at the kernel-level ([69, 70]), especially in massively multicore systems. Spinning is also not a viable candidate for local critical sections, which would require the operating system to treat such scenarios as special cases.

7.6 Distributed Resource Container Abstraction

The primary reason behind the development of distributed RK is to encapsulate the end-to-end properties of distributed real-time applications and transparently guarantee those properties. In order to achieve this, the distributed resource container abstraction was developed and the timing properties of the applications were specified. The distributed RK framework then transparently guaranteed these timing properties through appropriate admission control, reservation and enforcement. This distributed resource container abstraction can be further extended to incorporate security properties of the distributed application like access-capabilities and confidentiality requirements. For example, the distributed resource container could specify the level of confidentiality required for communication and the underlying system would transparently guarantee this property through appropriate encryption techniques on the messages passed between the subtasks of the application. Fault-tolerance properties can also be added to the distributed resource container abstraction in a similar fashion and properties like redundancy-requirements can be automatically handled by the operating-system framework. Distributed RK was designed as the foundation platform for enabling such a unified interface where applications specify their properties and the system guarantees that they are satisfied.

7.7 Summary

In this chapter, we have described the Distributed Resource Kernel framework to realize the algorithms presented in this dissertation. It accomplishes the important additional requirements of guaranteeing end-to-end deadlines that span multiple processor core boundaries, temporally isolating each end-to-end application from the other and offering facilities to ease development, deployment and diagnostics. The major highlights of our framework include end-to-end enforcement, strong predictability, analyzability and excellent performance.

The architecture supports an abstraction of a distributed resource container, that represents a virtual operating environment for each application. Name services and time synchronization services are also provided as a part of our framework. The framework was implemented on top of Linux. Detailed evaluation using benchmarks, artificial but stressful workloads, and realistic application scenarios show that our goals are satisfied. Implementation overheads were also shown to be acceptable.

Using this framework, the cache overhead of task-splitting has been evaluated on the Intel

Core 2 Duo and task migration overheads were seen to be very low on this platform. The implementation costs for our task synchronization primitives have also been quantified in the multi-core context. We have also provided considerations for future implementation of the proposed parallel real-time task scheduling algorithm in the context of fork-join task sets.

In the future, the distributed resource kernel abstraction can be extended to incorporate fault-tolerance semantics and security attributes. The current assumption of trust between different administrative domains need not hold good in general and the framework needs to operate in such environments. More benchmarks should also be developed to evaluate comprehensive aspects of distributed real-time systems.

Chapter 8

Conclusions and Future Work

In this dissertation, we have studied the problem of scheduling and synchronizing real-time periodic tasks on multi-core processors, using fixed-priority scheduling and static off-line task allocation. Our solution is an overall framework that enables efficient allocation of tasks to processor cores, synchronization between tasks, parallelization of tasks, and handling of overloads in mixed-criticality settings. This framework has been developed in the context of a resource kernel implementation, which demonstrates the practicality of our solution. This dissertation presents key advancements with respect to *utilization bounds*, *resource augmentation bounds*, and *response-time analyses* for multi-core processors.

8.1 Contributions

The major contributions of this dissertation are summarized in the following categories:

- Scheduling Independent Sequential Tasks on Multi-core Processors
- Multi-core Task Synchronization
- Scheduling Parallel Real-Time Tasks on Multi-core Processors
- Mixed-Criticality Scheduling for Multi-core Systems

- Distributed Resource Kernel Framework

More details of these individual contributions follow.

8.1.1 Scheduling Independent Sequential Tasks on Multi-core Processors

Bin-packing approaches to scheduling real-time tasks on multi-core processors have traditionally suffered from a 50% worst-case utilization bound. Although researchers had previously proposed task splitting approaches, the results from this dissertation were the first to increase the bound to 65% for fixed-priority scheduling. The key observation used in achieving this bound is that the highest-priority task has the shortest possible worst-case response time, hence resulting in the maximal deadline for the residual task to be allocated elsewhere. This observation, when combined with tasks being allocated in decreasing order of densities, results in the above-mentioned worst-case utilization bound of 65%.

The main benefit of the developed algorithm is its semi-partitioned nature, which results in off-line task allocation and statically defined task migrations. It also ensures that no more than one task per core migrates across cores, thereby minimizing the number of tasks being migrated. The fixed-priority scheduling approach also makes it practical for implementation in operating systems like Linux. The only additional requirement for implementing the algorithm is a per-task software timer to trigger the actual task migration.

In addition to providing analytical worst-case utilization bounds, we also measured the task migration overheads on an actual platform. The key difficulty in quantifying these overheads is their dependence on task working-set sizes, cache access patterns, and the actual hardware cache architecture itself. In our experimental evaluation, we have explored these different attributes using synthetic benchmarks, and also measured the actual overheads from inter-core migration of real-world applications such as MPlayer and GLGears.

8.1.2 Multi-core Task Synchronization

The main bottleneck in effectively utilizing multi-core processors is the synchronization requirement between tasks. Although existing task synchronization protocols such as the Multiprocessor Priority Ceiling Protocol (MPCP) provide bounded synchronization delays, such delays can still result in non-trivial scheduling penalties. In this dissertation, we developed an analysis of the blocking durations resulting from multi-core synchronization. We individually studied two different Execution Control Policies (ECPs) viz. *suspend* and *spin*, each resulting in very different blocking durations and scheduling penalties.

In order to minimize the penalty of inter-core task synchronization, we developed a coordinated approach to task allocation, scheduling, and synchronization, which leverages MPCP to provide bounded blocking delays and avoid inter-core task synchronization when possible. This approach was quantitatively evaluated for its utilization benefits and implementation overheads.

Our experimental results showed that synchronization-aware task allocation protocols could result in up to 50% fewer processor cores compared to synchronization-agnostic approaches. These results follow largely from the significant scheduling penalties arising from inter-core synchronization, which often outweigh any bin-packing benefits from allocating synchronizing tasks to different processor cores.

From a systems perspective, we implemented these synchronization protocols as extensions to the `pthreads` Application Programming Interface. This results in a practical solution that can be employed for synchronization in multi-core real-time systems.

8.1.3 Parallel Real-Time Task Scheduling

The advent of multi-core processors and the evolution to *many-core* systems signal the need for a shift in programming paradigms towards parallel task models. Unfortunately, the real-time systems literature has largely not studied such models, largely due to their lack of relevance in embedded systems, before the introduction of multi-core processors. In this dissertation, we have provided a *task transformation* based approach to scheduling *parallel real-time tasks* with fork-join task structures on multi-core processors.

The transformation developed in this dissertation efficiently converts the parallel task scheduling problem into the previously studied sequential scheduling problem. This enables us to take advantage of existing results and still analytically bound the performance of overall solution. In this dissertation, we have quantified the impact of this task transformation in terms of a bound on the increase in the overall task Demand-Bound Function (DBF). This enables us to apply this transformation in conjunction with other bin-packing algorithms as well.

The main intuition behind the task transformation developed in this dissertation is to isolate the master thread from the auxiliary threads. The master thread is then allocated its own processor cores, while the auxiliary threads can be handled once their deadlines are set appropriately. We use the semi-partitioning idea to ensure that the auxiliary threads fill up the processor core allocated to the master thread before using other processor cores.

8.1.4 Scheduling in Mixed-Criticality Multi-core Systems

Systems with single-criticality domains have received significant attention from the real-time systems community. An important consequence of multi-core processors is that application vendors are now actively considering *functional consolidation*, where tasks from different criticality domains are hosted on the same multi-core processor. This introduces many new challenges from the perspective of scheduling and synchronization.

In this dissertation, we introduced the zero-slack scheduling algorithm to handle mixed-criticality tasks in uni-core processors. We divide the task execution into *normal* and *critical* execution modes, ensuring that tasks can overload as long as they do not affect higher-criticality tasks. This enables tasks to handle overload conditions at the expense of lower-criticality tasks, which are less vital for system operation. We have developed a metric called *ductility* to succinctly quantify this property of zero-slack scheduling.

Given the existence of both semantic priority (i.e. criticality) and scheduling priority, we have identified the shortcomings of standard priority inheritance and ceiling protocols in such mixed-criticality contexts. In order to address these issues, we have developed the Priority and Criticality Inheritance Protocol (PCIP) and the Priority and Criticality Ceiling Protocol (PCCP) to provide bounded synchronization with proper overload behavior in mixed-criticality systems.

Next, having addressed the issue of scheduling and synchronization in a uni-core processor context, we have extended the solution to handle multi-core systems. The idea here is to pack tasks in criticality order from the highest criticality level to the lowest criticality level. Within each criticality level, we can follow the standard approach of allocating tasks in decreasing order of size. We use the overloaded execution times to determine the size of higher-criticality tasks, while using the normal execution times to determine the size of lower-criticality tasks. This approach enables us to guarantee that the higher-criticality tasks can overload at the expense of lower-criticality tasks, when necessary.

Our contribution with respect to mixed-criticality systems is illustrated using a real-world case-study of a radar surveillance application, where tasks have different criticality levels. We have implemented this system on a multi-core processor with our Linux Resource Kernel framework and demonstrated the benefits of the developed mixed-criticality scheduling over traditional criticality-agnostic fixed-priority scheduling.

8.1.5 Distributed Resource Kernel Framework

The systems contribution of this thesis is the incorporation of multi-core processors in the distributed resource kernel framework. In this dissertation, we have extended the Linux Resource Kernel framework to support multi-core and also distributed systems. The resulting framework is a comprehensive and practical solution for task allocation, scheduling, synchronization, parallelization, and overload provisioning in multi-core processors.

The main reason for the practicality of our implementation on the Linux Resource Kernel is the approach adopted in this dissertation, which focuses on fixed-priority, static task allocation, and statically defined task migration points. All of these aspects are easily supported on Linux and could significantly ease the adoption of the developed solutions in real-world systems. The extended distributed resource kernel framework can also serve as a platform for implementing and evaluating future multi-core scheduling and synchronization algorithms.

8.2 Future Work

Multi-core processors are relatively recent developments in the arena of real-time and embedded systems. There are many possible avenues for future work with regards to this dissertation and the proposed approach. We now discuss some of the major topics, where work from this dissertation can be applied and extended.

8.2.1 Worst-Case Execution Time Analysis for Multi-core Processors

Among the parameters used in the task model for this dissertation, determining the exact Worst-Case Execution Time ($C_{i,k}^j$) for each individual task segment is the most challenging requirement. In multi-core processors, the presence of multiple levels of cache and shared on-chip resources complicates the estimation of task worst-case execution times. Although highly pessimistic estimates are easily obtainable for worst-case execution times, these are often much larger than the actual values and result in significantly over-provisioned systems.

Although recent research [127, 129] has started investigating worst-case execution times in shared cache architectures, there are many other aspects that are not well modeled. For instance, cache misses from the multiple processor cores could have to be serviced using the same memory bus. Main memory [87, 88] is also a shared resource that could become a potential source of interference. Research [65] has also looked at the trade off between fairness and system throughput. Applying these principles in a real-time systems context is a promising research direction. Also, developing modeling tools for accurately estimating such interference in multi-core processors is important future work for efficiently utilizing these processors in real-time systems.

8.2.2 Non-Uniform and Heterogeneous Multi-core Processors

One assumption throughout this dissertation has been that of uniform and homogeneous multi-core processors, where each of the processing cores is identical in terms of the instruction set and processing speed. Relaxing this assumption leads to the first interesting area of future work. These non-uniform and heterogeneous multi-core processors are already being considered in embedded systems due to their energy benefits and potentially richer functionality. Although recent research efforts have started studying this problem, there is significant potential for interesting and efficient solutions.

From the perspective of this dissertation, heterogeneity can be represented as constraints to the bin-packing algorithm and task splitting should be restricted to either homogeneous or compatible sets of processor cores. The possibility of non-uniform processor cores executing at different speeds can be handled using bins of varying sizes. For tasks that can be accelerated on specific processor cores, one could consider the task sizes to be smaller when considering such hardware-accelerated processor cores.

Given the above mapping of heterogeneous multi-core scheduling to the bin-packing approach adopted in this dissertation, there are still many questions with respect to the actual bin-packing algorithm to be adopted and significant trade-offs to be made across different allocation choices. All of these aspects make for an interesting avenue of future research in semi-partitioned fixed-priority scheduling for heterogeneous and non-uniform multi-core processors.

8.2.3 Energy Management

The main motivation behind the adoption of multi-core processors is their associated power and thermal benefits. Energy management is therefore expected to be a key objective in scheduling real-time tasks on embedded multi-core processors. The continuously evolving hardware architectures of multi-core processors result in a challenging design space for such energy-aware scheduling algorithms. Future many-core processors are expected to have multiple voltage island and clock domains on the same chip. This characteristic results in constraints such as cores within the same clock domain having to operate at a coupled frequency and the cores within a voltage domain having to operate at the same voltage. In some processor architectures, the different processor cores need to shift to a low-power state simultaneously for the entire chip to enter a low-power state, which requires coordination among the individual processor schedules to achieve true low-power operation.

From the approach proposed in this dissertation, we can represent frequency scaling as a reduction in the bin sizes. We could pack the tasks on bins with artificially smaller sizes and then translate the resulting allocation back to the original processor operating at a lower frequency. In fact, such an approach enables us to employ traditional energy-agnostic multi-core scheduling algorithms in the context of energy management.

Co-ordination across processor cores can be achieved by creating a *per-core sleep task*, which could execute at the highest scheduling priority on the processor core. Synchronized release of such sleep tasks could result in the different processor cores simultaneously entering a low-power state, which enables the entire chip to enter a low-power state.

Depending upon the actual processor architecture, there are interesting trade-offs between turning off processor cores and actively operating fewer cores at a higher frequency *versus* actively operating more cores at a lower frequency. Quantifying these trade-offs on a real-world processor architecture and developing an appropriate task allocation algorithm would be interesting.

8.2.4 Soft-Real Time and Aperiodic Tasks

Another major assumption in this dissertation has been that of hard deadlines and periodic tasks. Relaxing these assumptions results in a wide range of future work. For instance, deferrable servers, sporadic servers, constant bandwidth servers, and total bandwidth servers, are all well-studied mechanisms in the real-time systems literature. Extending and optimizing these tools for multi-core processors would provide an interesting direction of future research.

There are interesting design questions that arise with respect to designing *server* mechanisms for a multi-core processor. Addressing these questions could result in future *server* designs to handle aperiodic tasks on multi-core processors.

- Should the aperiodic tasks be distributed across processor cores or restricted to a subset of processor cores?
- How to provision the server budgets when the aperiodic tasks do not have known worst-case execution times and deadlines?
- What are the scheduling penalties from splitting such *server* tasks across cores?
- Could these *server* mechanisms service tasks from a global queue or are the tasks statically allocated to *servers* when the aperiodic tasks are known apriori?

Developing a comprehensive solution to handle systems with hard and soft deadlines, and periodic/aperiodic tasks is an important problem faced in adopting multi-core processors for real-time systems. An implementation of the solution in the distributed resource kernel framework would bring the system closer to adoption in real-world applications.

8.2.5 Graphics Processing Unit (GPU) Scheduling

Graphics Processing Units (GPUs) have evolved rapidly over the past few years from special-purpose hardware-acceleration units to mainstream computing processors. In fact, some of the GPUs have orders of magnitude more processing cores than general-purpose processors. The key challenge introduced by GPUs with respect to scheduling is their Single Instruction Multiple Data stream (SIMD) architecture. This is very different from the traditional assumption of an Multiple Instruction Multiple Data stream (MIMD) architecture made in this dissertation. There are many interesting problems to be addressed in effectively utilizing GPUs in future real-time systems.

The first major issue is the co-scheduling problem between the general-purpose processor core and the graphics-processor core. In architectures with multiple general-purpose processor cores and multiple graphics-processor cores, this problem is complex and unwieldy. The commands from different general-purpose cores could be queued up on the graphics-processor core, resulting in significant blocking delays to the applications. Effectively scheduling these commands on the GPU and minimizing these blocking delays is an interesting challenge in GPU scheduling.

Another important challenge with current generation GPUs is to deal with their non-preemptive nature. In most GPUs, the general-purpose processor does not have control over when the commands sent to the GPU are executed. The non-preemptive nature of GPUs further adds to delays in executing the commands from the host processor, which can result in performance issues. Bounding the delays from an analytical perspective and just-in-time scheduling of the commands at the host processor to GPU interface are interesting directions for future work.

Lastly, memory management is an interesting problem in GPU scheduling. The GPU has limited on-chip memory and the scheduler needs to consider this when sending commands to the GPU. Also, the memory transfer between the main memory and GPU itself can cause priority inversions to other applications using the GPU. These characteristics of GPUs open up many more challenges in effectively utilizing such processors in future real-time systems.

Bibliography

[1] AMD. Opteron 6000. http://www.amd.com/us/products/server/processors/6000-series-platform/pages/6000-series-platform.aspx. [Online; accessed 14-July-2011]. 1

[2] A. Amin, R. Ammar, and A. El Dessouly. Scheduling real time parallel structures on cluster computing with possible processor failures. *Computers and Communications, IEEE Symposium on*, 1:62–67, 2004. doi: http://doi.ieeecomputersociety.org/10.1109/ISCC.2004.1358382. 2.3

[3] J.H. Anderson, V. Bud, and U.C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 199 – 208, july 2005. doi: 10.1109/ECRTS.2005.6. 1, 2.1

[4] J.H. Anderson, J.M. Calandrino, and U.C. Devi. Real-time scheduling on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 179 – 190, april 2006. doi: 10.1109/RTAS.2006.35. 2.1

[5] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 243 –252, july 2008. doi: 10.1109/ECRTS.2008.9. 3.2.4

[6] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority

scheduling on multiprocessors are 50 In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 33 – 40, july 2003. doi: 10.1109/EMRTS.2003.1212725. 1.6, 2.1, 3.1

[7] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 322 –334, 0-0 2006. doi: 10.1109/RTCSA.2006.45. 2.1

[8] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 193 – 202, dec. 2001. doi: 10.1109/REAL.2001.990610. 2.1

[9] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Real-Time Systems Symposium, 2008*, pages 385 –394, 30 2008-dec. 3 2008. doi: 10.1109/RTSS.2008.44. 3.2.4

[10] Bjorn Andersson and Jan Jonsson. Preemptive multiprocessor scheduling anomalies. *Parallel and Distributed Processing Symposium, International*, 1:00–12, 2002. 1

[11] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. real-time scheduling: the deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991. 5.4

[12] AUTOSAR. Autosar - automotive open system architecture. `http://www.autosar.org`. [Online; accessed 18-July-2011]. 1.6

[13] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems Journal*, 3 (1):67–99, 1991. 6.4.1

[14] T.P. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760 – 768, aug. 2005. ISSN 1045-9219. doi: 10.1109/TPDS.2005.88. 2.1

[15] Moshe Bar. khttpd, a kernel-based web server. *Linux J.*, pages 58–59, August 2000. 7.4.3

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *Proceedings of the 19th ACM SOSP*, pages 164–177, October 2003. 2.5

[17] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996. ISSN 0178-4617. URL `http://dx.doi.org/10.1007/BF01940883`. 10.1007/BF01940883. 2.1

[18] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, July 2008. 2.4

[19] Sanjoy Baruah, Haohan Li, and Leen Stougie. Mixed-criticality scheduling: improved resource-augmentation results. *Proceedings of the ISCA International Conference on Computers and Their Applications (Honolulu, Hawaii)*, March 2010. 2.4

[20] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, April 2010. 2.4

[21] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *In Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990. 5.4.1

[22] S.K. Baruah and J.R. Haritsa. Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 46(9):1034–1039, 1997. 2.4

[23] S.K. Baruah and Shun-Shii Lin. Pfair scheduling of generalized pinwheel task systems. *Computers, IEEE Transactions on*, 47(7):812 –816, jul 1998. ISSN 0018-9340. doi: 10.1109/12.709381. 2.1

[24] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. An empirical comparison of global,

partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14 –24, 30 2010-dec. 3 2010. doi: 10.1109/RTSS. 2010.23. 1.6

[25] B.B.Brandenburg and J.H.Anderson. A comparison of the m-pcp, d-pcp, and fmlp on $litmus^{RT}$. *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 105–124, 2008. 2.2, 4.2.2

[26] B.B.Brandenburg, J.M.Calandrino, A.Block, H.Leontyev, and J.H.Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? *RTAS*, pages 342–353, 2008. 2.2

[27] A. Bertossi and L.V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems*, 7(3):229–245, 1994. 2.1

[28] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: http://doi.acm.org/10.1145/1278480. 1278667. URL `http://doi.acm.org/10.1145/1278480.1278667`. 1

[29] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. Litmusrt: A status report. *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123, November 2007. 2.5

[30] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *proc. of the 24th, IEEE RTSS*, 2003. 2.4

[31] R. Butenuth, W. Burke, and H.-U. HeiB. Cosy - an operating system for highly parallel computers. *Operating Systems Review*, 30:81–91, April 1996. 2.5

[32] Giorgio Buttazzo, Marco Spuri, and Fabrizio Sensini. Value vs deadline scheduling in overload conditions. In *Proceedings of the 16st, IEEE Real-Time Systems Symposium*,

1995. 2.4

[33] Giorgio Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic task model for adaptive rate control. *IEEE RTSS*, pages 286–295, 1998. 2.4

[34] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004. 2.3

[35] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 101 –110, dec. 2006. doi: 10.1109/RTSS.2006.10. 2.1

[36] S. Cho, S.-K. Lee, A. Han, and K.-J. Lin. Efficient real-time scheduling algorithms for multiprocessor systems. *IEICE Trans. Communications, E85-B(12)*, pages 2859–2867, Dec 2002. 2.4

[37] Sébastien Collette, Liliana Cucu, and Joel Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180 – 187, 2008. 2.3

[38] P. Dasgupta, R. J. LF, Blanc, M. Ahamad, and U. Ramachandran. The clouds distributed operating system. *IEEE Computers*, 24, 11:34–44, Nov 1991. 2.5

[39] R. Davis and A. Wellings. Dual priority scheduling. *In Proceedings of the 16th IEEE RTSS*, pages 100–109, Dec 1995. 2.4

[40] Dionisio de Niz and Raj Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems*, 2:196–208, 2006. 2.2

[41] Dionisio de Niz, Karthik Lakshmanan, and Raj Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proceedings of RTSS*, Washington, DC, USA, 2009. IEEE. 6.3.1, 6.3.2, 6.4.2, 6.4.3, 6.4.3

[42] M.L. Dertouzos and A.K Mok. Multiprocessor online scheduling of hard-real-time tasks.

IEEE Transactions on Software Engineering, 15:1497–1506, 1989. 2.3

[43] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *OPERATIONS RESEARCH*, 26(1):127–140, 1978. doi: 10.1287/opre.26.1.127. URL `http://or.journal.informs.org/cgi/content/abstract/26/1/127`. 2.1, 2.3

[44] Anand Eswaran and Raj Rajkumar. Energy-aware memory firewalling for qos-sensitive applications. *In Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005. 7.1

[45] R. Braden et. al. Resource reservation protocol (rsvp) - version i functional specification. *Internet Draft*, Nov 1996. 2.5

[46] expresslogic. Threadx - real-time operating system. `http://rtos.com/`. [Online; accessed 18-July-2011]. 1.6

[47] Xiang Feng. Towards real-time enabled microsoft windows. *Proceedings of the 5th International Conference On Embedded Software*, pages 142–146, 2005. 2.5

[48] Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *ECRTS*, pages 118–127. IEEE Computer Society, 2006. ISBN 0-7695-2619-5. doi: http://dx.doi.org/10.1109/ECRTS.2006.30. 5.4, 16, 5.4.1

[49] FreeScale. Embedded Multicore: An Introduction. `www.freescale.com/files/32bit/doc/ref_manual/EMBMCRM.pdf`, 2009. [Online; accessed 14-July-2011]. 1

[50] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *In Proceedings of the IEEE Real-Time Systems Symposium (December 2001), IEEE Computer*, pages 183–192. Society Press, 2001. 5.2.2

[51] M.K. Gardner and Liu J.W.S. Performance algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of the 11th ECRTS*, 1999. 2.4

[52] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11 – 13, may 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.160. 1

[53] Sourav Gosh and Ragunathan Rajkumar. Resource management of the os network subsystem. *In Proceedings of the IEEE International Symposium on Object-oriented Real-time distributed Computing*, 2002. 7.1

[54] Salvatore Greco. *Multiple Criteria Decision Analysis: State of the Art Surveys*. Springer's International Series on Operation Research Management Science, 2005. 6.2

[55] Nan Guan, M. Stigge, Wang Yi, , and Ge Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 165–174, April 2010. 2.1

[56] C.-C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. *Real Time Systems Symposium, 1989., Proceedings.*, pages 59 –67, dec. 1989. doi: 10.1109/REAL.1989.63557. 5.2.1

[57] C.J. Hou and K.G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *Trans. Computers*, 46(12), Dec 1997. 7.3

[58] Intel. Xeon 7500 Processor. `http://www.intel.com/products/server/processor/xeonE7/index.htm`. [Online; accessed 14-July-2011]. 1

[59] J.M.Lopez, J.L.Diaz, and D.F.Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. *Proceedings of ECRTS*, 2001. 6.5.3

[60] Nicholas T. Karonis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *In Journal of Parallel and Distributed Computing, Sp. Issue on Computational Grids*, May 2003. 2.5, 7.4.3

[61] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 441 –450, aug. 2007. doi: 10.1109/RTCSA.

2007.61. 2.1

[62] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23 –32, april 2009. doi: 10.1109/RTAS.2009.9. 2.1, 3.2.4

[63] Shinpei Kato and Yutaka Ishikawa. Gang edf scheduling of parallel task systems. In *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 459–468, Washington, DC, USA, 2009. IEEE Computer Society. 1.3.1, 2.3

[64] Ashok Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *J. Parallel Distrib. Comput.*, 43(1):37–45, 1997. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1997.1327. 2.3

[65] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: http://dx.doi.org/10.1109/MICRO.2010.51. URL `http://dx.doi.org/10.1109/MICRO.2010.51`. 8.2.1

[66] Michael O. Kolawole. *Radar systems, peak detection and tracking / Michael Kolawole*. Newnes, 2002. 6.1

[67] S. Lauzac, R. Melhem, and D. Mossé. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 188 –195, jun 1998. doi: 10.1109/EMWRTS.1998.685084. 2.1

[68] D. Lea. A java fork/join framework. *In Proceedings of the ACM 2000 Java Grande Conference*, page 3643, June 2000. 1.3.1, 2.3

[69] J. Lee and K. H. Park. Delayed locking technique for improving real-time performance of

embedded linux by prediction of timer interrupt. *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005. 4.3.2, 7.5.10

[70] Jupyung Lee and K.H.Park. The lock hold time prediction of non-preemptible sections in linux 2.6.19. *TR, KAIST*, 2007. 4.3.2, 7.5.10

[71] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166 –171, dec 1989. doi: 10.1109/REAL.1989.63567. 1.6, 2, 3.3.2

[72] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2*, page 237250, Dec 1982. 2.3

[73] Xuan Lin, Ying Lu, Jitender Deogun, and Steve Goddard. Real-time divisible load scheduling for cluster computing. *In 13th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 303–314, 2007. 2.3

[74] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321738.321743. URL `http://doi.acm.org/10.1145/321738.321743`. 3.2.2, 3.3.2, 5.2.1, 5.2.1, 5.2.2, 5.4.1, 6.4.2

[75] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. 6.3.1, 6.3.4

[76] J.W.S. Liu, K.-J. Lin, W.-K. Shih, A.C.-s. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58 –68, may 1991. ISSN 0018-9162. doi: 10.1109/2.76287. 2.4

[77] LLNL. Posix threads programming. `http://www.llnl.gov/computing/tutorials/pthreads/`. [Online; accessed 18-July-2011]. 7.5.2

[78] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28:39–68, 2004. ISSN 0922-

6443. URL `http://dx.doi.org/10.1023/B:TIME.0000033378.56741.14`. 10.1023/B:TIME.0000033378.56741.14. 2.1

[79] L.Sha, R.Rajkumar, and J.P.Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990. 4, 4.1.1, 4.1.2

[80] L. Lundberg. Analyzing fixed-priority global multiprocessor scheduling. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 145 – 153, 2002. doi: 10.1109/RTTAS.2002.1137389. 2.1

[81] Pedro Mejia-Alvarez, Rami Melhem, and Daniel Mossé. An incremental approach to scheduling during overloads in real-time systems. In *Proceedings of the 21st, IEEE RTSS*, 2000. 2.4

[82] C.W. Mercer, Y. Ishikawa, and H. Tokuda. Distributed hartstone: A distributed real-time benchmark suite. *In Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990. 7.5.6, 7.5.6

[83] C.W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1994. 7.1

[84] D. Mills. Network time protocol (version 3), specification, implementation and analysis. *Internet RFC1305*, Mar 1992. 7.4.6

[85] L. Ming. Scheduling of the inter-dependent messages in real-time communication. *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*, 1994. 2.2, 4.2.2

[86] I. Mizunuma, C. Shen, and M. Takegaki. Middleware for distributed industrial real-time systems on atm networks. *In 17th IEEE Real-Time Systems Symposium*, 1996. 2.5

[87] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-

aware memory channel partitioning. *SAFARI Technical Report, TR-SAFARI-2011-002, Carnegie Mellon University*, June 2011. 8.2.1

[88] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. *Proceedings of the 40th International Symposium on Microarchitecture (MICRO)*, pages 146–158, December 2007. 8.2.1

[89] S. Oikawa and R. Rajkumar. Linux/rk: A portable resource kernel in linux. *In Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, December 1998. 2.5

[90] S. Oikawa and R. Rajkumar. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*, pages 111 –120, 1999. doi: 10.1109/RTTAS.1999.777666. 3.2.4, 7.1

[91] Shuichi Oikawa and Raj Rajkumar. Linux/rk: A portable resource kernel in linux. *proc. of the 19th, IEEE RTSS*, 1998. 2.4

[92] OpenMP. Openmp specification for parallel programming. `http://openmp.org`. 1.3.1, 2.3, 2.5

[93] P.Gai, M.di Natale, G.Lipari, A.Ferrari, C.Gabellini, and P.Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. *RTAS*, pages 189–198, 2003. 2.2

[94] P.Holman and J.H.Anderson. Locking in pfair-scheduled multiprocessor systems. *RTSS*, pages 149–158, 2002. 2.2

[95] G. J. Popek and B. J. Walker. *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press, 1985. 2.5

[96] R. Rajkumar. Synchronization in real-time systems: A priority inheritance approach. *Kluwer Academic Publishers*, page 208, 1991. 2.5

[97] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multi-

processors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259 –269, dec 1988. doi: 10.1109/REAL.1988.51121. 3.2.4

[98] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *In Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997. 6.2

[99] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia operating systems. *In Proceedings of the SPIE conference on Multimedia Computing and Networking*, January 1998. 7.1

[100] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. *Readings in multimedia computing and networking*, pages 476–490, 2001. 2.4

[101] K. Ramamritham, J.A. Stankovic, and P.F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1: 184–194, 1990. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/71.80146. 2.3

[102] R.I.Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. In *Technical Report YCS217, Department of Computer Science, University of York*, 1993. 2.4

[103] R.Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *The Tenth International Conference on Distributed Computing Systems*, 1990. 4

[104] R.Rajkumar. *Dealing with Suspending Periodic Tasks*. IBM Thomas J. Watson Research Center, Yorktown Heights, 1991. 4.1.3, 4.2.2

[105] R.Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. 1.6, 2.2, 4.1.2

[106] R.Rajkumar, L.Sha, and J.P.Lehoczky. Real-time synchronization protocols for multipro-

cessors. *RTSS*, pages 259–269, 1988. 2, 4

[107] Saowanee Saewong and Ragunathan Rajkumar. Cooperative scheduling of multiple resources. *In Proceedings of the IEEE Real-Time Systems Symposium*, December 1999. 7.1

[108] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, Sep 1990. 6.3.1, 6.4.1, 6.4.1, 6.4.1

[109] Lui Sha, John P. Lehoczky, and Raj Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the IEEE Industrial Electronics Conference*, 1987. 2.4

[110] Chi-Sheng Shih, Phanindra Ganti, and Lui Sha. Schedulability and fairness for computation tasks in surveillance radar systems. In *Proceedings of the 10th RealTime and Embedded Computing Systems and Applications Conference*, 2004. 2.4

[111] A. Srinivasan, P. Holman, J.H. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10 pp., april 2003. doi: 10.1109/IPDPS.2003.1213226. 2.1

[112] J.A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, pages 62–72, May 1991. 2.5

[113] J.K. Strosnider, J.P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *Computers, IEEE Transactions on*, 44(1):73 –91, jan 1995. ISSN 0018-9340. doi: 10.1109/12.368008. 3.2.3

[114] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 38–45, May 1996. 2.5

[115] V Sundaram, A. Chandra, P. Goyal, P. Shenoy, J Sahni, and H Vin. Application performance in the qlinux multimedia operating system. *Proceedings of the Eighth ACM*

Conference on Multimedia, November 2000. 2.5

[116] A. Tannenbaum. Amoeba: A distributed operating system for the 1990s. *Computer*, pages 44–53, 1990. 2.5

[117] S.R. Thuel and J.P. Lehoczky. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 110–123, 1992. 2.4

[118] S.R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the Real-Time Systems Symposium*, pages 22–33, 1994. 2.4

[119] Tilera. Tilera 100-core processor. `http://www.tilera.com`. 1

[120] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40:117–134, 1994. 2.5

[121] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *Real-Time Systems Journal*, 4(2):145–166, May 1992. 7.3

[122] H. Tokuda and C.W Mercer. Arts: a distributed real-time kernel. *In SIGOPS Oper. Syst. Rev.*, 23(3), Jul 1989. 2.5

[123] T.P.Baker. A stack-based resource allocation policy for realtime processes. *RTSS*, pages 191–200, 1990. 2.2

[124] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, Jan. 2008. ISSN 0018-9200. doi: 10.1109/JSSC.2007.910957. 1

[125] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *In Proceedings of RTSS*, pages 239–243, 2007. 2.4

[126] V.F.Wolfe, L.C.DiPippo, R.Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston.

Real-time corba. *In Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, June 1997. 2.5

[127] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 80 –89, april 2008. ISSN 1080-1812. doi: 10.1109/RTAS.2008.6. 8.2.1

[128] S. H. Zanakis, A. Solomon, N. Wishart, and S. Dublish. Multi-attribute decision making: A simulation comparison of select methods. *European Journal of Operational Research*, 107:507–529, 1998. 6.2.2

[129] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 455 –463, aug. 2009. doi: 10.1109/RTCSA.2009.55. 8.2.1

[130] Dakai Zhu, Xuan Qi, Daniel Mossé, and Rami Melhem. An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof:–, 2011. ISSN 0743-7315. doi: DOI:10.1016/j.jpdc.2011.06.003. URL `http://www.sciencedirect.com/science/article/pii/S0743731511001304`. 2.1