SCHEDULING AND EXECUTION TIME ANALYSIS FOR MULTI-CORE ARCHITECTURE

Project Report submitted in partial fulfillment of the Degree of Bachelor of Technology

In

Computer Science

Under the Supervision of

Dr. Vivek Kumar Sehgal

(Associate Professor)

By

Prashant Kumar

Enrollment no. : 111317

To

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "**Scheduling and Execution time analysis for multi-core architecture**" submitted by **Prashant Kumar** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology ,Waknaghat ,Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Dr. Vivek Sehgal
(Associate Professor)

Acknowledgement

On the very outset of this report, I would like to extend my sincere & heartfelt obligation towards all the personages who have helped me in this endeavor. Without their active guidance, help, cooperation & encouragement, I would not have made headway in the project.

I would like to show my greatest appreciation to **Dr. Vivek Sehgal**. I feel motivated every time I get her encouragement. For her coherent guidance throughout the tenure of the project, I feel fortunate to be taught by **Dr. Vivek Sehgal**, who gave me her unwavering support. Besides being my mentor, she taught me that there is no substitute for hard work.

I express my gratitude and sincere thanks to **Prof. R.M.K. Sinha**, Dean, Department of Computer Science and Engineering. for allowing me to undertake this project.

I deeply express my sincere thanks to **Brig. (Retd.) S. P. Ghrera**, Head of Department, Department of Computer Science and Engineering, for encouraging and allowing me to present this project

I would also thank my parents **Mr. Hriday Narayan Singh** and **Mrs. Saroj Devi** for teaching me how to be successful and how to achieve my goal which helped in completing this project.

They have supported me in every step of my life.

Date: Prashant Kumar

Table of Contents

List of Figures

Abstract

Multi-core processors are already widespread in general-purpose computing systems with many no of manufacturers currently offering up to a ten to twelve cores per processor. With this the systems adopting such multi-core enabled processors gain increased computational capacity, improved parallelism, and higher performance with respect to power consumption. However, using multicore processors in real-time applications also introduces new challenges and opportunities for development of efficient scheduling algorithms. In this report on scheduling issue in multi-core processor, we study this problem of scheduling, characterize the design space, and develop an analytical system which will provide the efficiency of few of the effective algorithms available till today. Exploiting the nature of processor cores, the general principle adopted in this report is to statically partition tasks among processor cores, co-allocate multiple synchronizing tasks when possible, and introduce limited inter-core task migration when required and scheduling them when necessary for improving system utilization. We then analyze the overheads of inter-core task scheduling and synchronization and provide mechanisms to efficiently allocate synchronizing sequential tasks on multicores by co-locating such tasks on the basis of statics gathered over developed new system of scheduling. The results of this report contribute to a system that can efficiently utilize multi-core processors to predictably execute periodic real-time tasks with well-defined deadlines.

Chapter 1

Introduction

Processor manufacturers have widely adopted multi-core technologies to effectively utilize continuously increasing transistor density limitation. Multi-core technology is considered as a practical alternative to increase the processor clock frequency, which is limited by available instruction-level parallelism and leads to challenging power/thermal requirements. Commercial vendors such as Intel, AMD, and Free-Scale, already have offered various processor solutions with multiple cores on the same package of processor. Processors with up to a ten to twelve cores per package are already employed in production systems, while research prototypes with 80 cores-perchip have been successfully developed by Intel with new technology of bringing 10nm transistors on the cores. Tilera has also demonstrated a 100-core chip with shared caches, interconnects, and memory controllers. Researchers have even predicted that chips with hundreds of processing cores on the same package could become available in the future for general public as till today they are in experimental phase. Given the proliferation of multi-core processors in general-purpose computing, embedded and real-time applications such as **smart phones** are actively considering the use of such processors with 4 to 8 cores on a single package of processor. These application domains would benefit from the additional computational capacity made available at a lower power requirement which is considered as one of the key factor in their performance. However, effectively utilizing multi-core processors in traditional realtime applications requires new tools and techniques for programming, scheduling, synchronization, certification, and runtime support. In this report, 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 later chapters. The main motivation for this topic arises from the heavy computational requirements of many hard real-time applications such as automotive engine control, driver assistance and many more real-time application. These systems often execute control loops and other time-critical tasks, which require absolute guarantees on meeting

deadlines for task completion. The heavy computational demand motivates the need for employing multi-core processors, which could enable additional functionality such as smarter algorithms for 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

(ii) The potential availability of multiple levels of on-chip shared cache

(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, where a limited number of tasks are allowed to migrate across core boundaries. In this report, 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 multi-core systems.

1.1 Report Scope

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

1. Homogeneous Uniform Multi-core Processors, where each processor core has the same 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 report can be used for scheduling purpose of processes as because of time and resource limitation. We do not study servers or advanced computer processors 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 their deadlines.

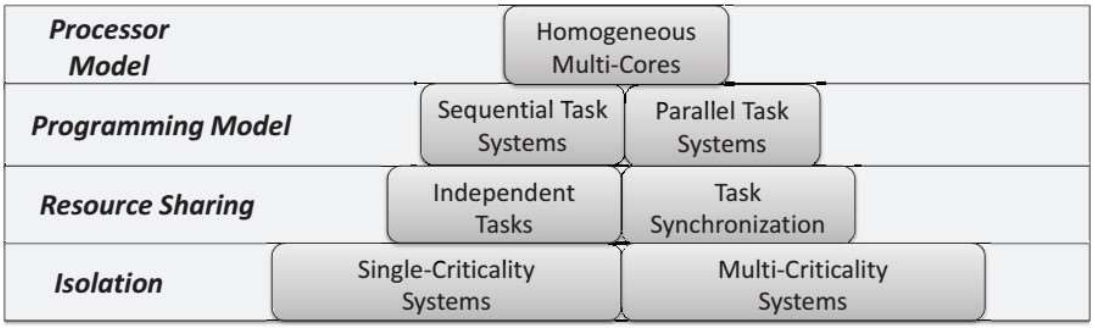


Figure 1.1: Report Scope

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. 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 and heave knowledge of kernel level implementation which will be completed in next part of the report. A few of the key design choices considered for 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 multicore embedded processors as well. Architectures with bounded interference would significantly reduce the pessimism in worst-case execution time analysis.

1.2 THE FUTURE OF SCHEDULING AND SYNCHRONIZATION

Transactional Memory: A *transaction* is a sequence of steps executed by a single thread. Transactions are "serializable", meaning transactions appear to execute sequentially, in a one-at-a-time order. Transactions are often (but not always) executed "speculatively." A speculative transaction that succeeds is said to "commit," and its effects become visible to other threads, while one that fails is said to "abort" (or cancel), and its effects are discarded. Direct hardware support for transactions will have a pervasive effect across the software stack, affecting how we implement and reason

about everything from low-level constructs like mutual exclusion locks, to concurrent data structures such as skip-lists or priority queues, to system-level constructs such as read-copy-update (RCU), all the way to run-time support for high-level language scheduling mechanisms. Although transactions can alleviate many of the well-known shortcomings of legacy scheduling constructs, we believe that such a pervasive change will present new challenges and opportunities very distinct from the familiar issues we face today.

Chapter 2

Background

Work related to this report falls in four categories:

(i) Multi-processor and multi-core scheduling

(ii) Multi-processor synchronization

(iii) Parallel task scheduling

We will now discuss the related work in first two of domains and describe the differences with few of the available approach till date.

2.1 Multi-processor and Multi-core Scheduling

The design space of the existing literature on multi-processor real-time scheduling algorithms is provided in figure 2.1. Multiprocessor scheduling schemes are classified into **global** and **partitioned** systems. It has been shown that each of these categories has its own advantages and disadvantages. Global scheduling schemes can better utilize the available processors. 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. On the other hand, partitioned 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. 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 multiprocessors. By dividing the task-set into heavy-weight and lightweight tasks, the RMUS algorithm achieves a utilization bound of 33% for fixed-priority global scheduling. These results have been improved with a higher bound of 37.5%. The global EDF scheduling schemes have been shown to possess a higher utilization bound of 50%. PFair scheduling algorithms based on the notion of proportionate progress can achieve the optimal utilization bound of 100%. Recent approaches 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 multicores, 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 and dynamic-priority algorithms based on 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. Partitioned dynamic-priority scheduling schemes with task splitting have been explored in this context. Fixed-priority scheduling with task-splitting support is relatively less analyzed in the literature. Recent results have provided task-splitting algorithms in the fixed-priority context. Optimal task-splitting algorithms have also been developed for tasks having the same period. Even though these have subsequently achieved a higher worst-case utilization bound, the average-case performance the provided 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 cacheaware approaches to real-time scheduling. We focus more on exploiting the shared caches to minimize the overhead of task splitting, rather than explicitly

choosing cachecollaborative 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 and work.

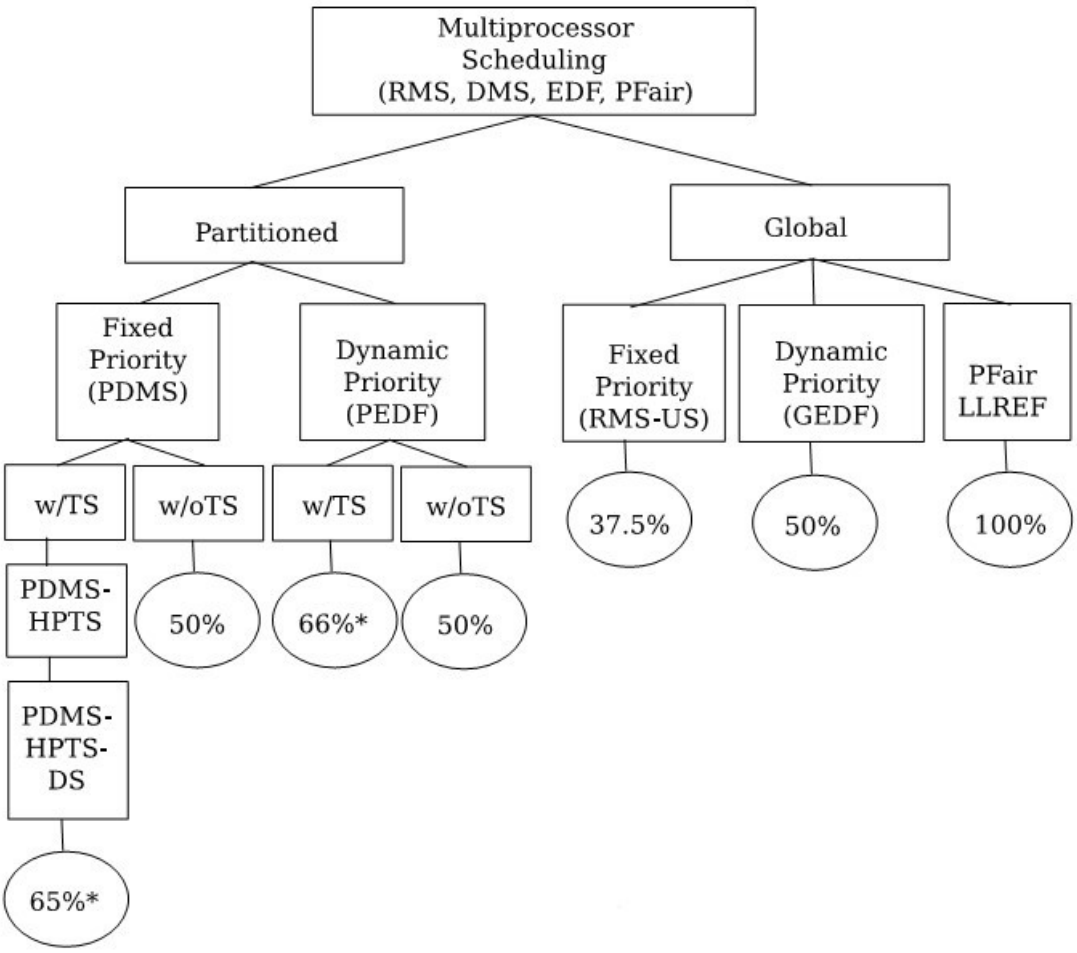


Figure 2.1 Multiprocessor scheduling algorithms

2.2 Multi-processor Synchronization

Traditional multiprocessor scheduling algorithms have dealt mostly with independent tasks having no interactions. Researchers like 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. 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 report, we are concerned with tasks that need to synchronize on (potentially) globally shared resources, which could be executing on different processor cores. Task synchronization for real-time systems application 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) 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). Synchronization schemes have also been developed for other related scheduling paradigms like PFair. Multiprocessor extensions to SRP have also been considered and performance comparisons have been done with MPCP. This dissertation adopts a provided 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 have investigated the performance differences between spin-based and suspension-based synchronization protocols. In these studies, their authors found that spinbased 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. With this new analysis, we found that the suspension-based proto-18 cols in fact behave better than spin under low preemption costs (less than 160µs per preemption) and longer critical sections (15 µs) than those studied.

Chapter 3

Synchronization of Sequential Tasks

In this chapter, we relax the assumption of independent tasks, which was made in above chapter. 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) have been extended to the multiprocessor scenario . 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 focus on a synchronization-aware partitioned fixed-priority scheduler to accommodate these inefficiencies. 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. 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.

3.1 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.

3.1.1 Global vs Local Synchronization

The Priority Ceiling Protocol (PCP) 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 3.1**. In this figure, there are three tasks, $\tau 1$ and $\tau 2$ running in processor P1 and $\tau 3$ running in processor P2. In addition, a resource is shared between tasks $\tau 2$ and $\tau 3$ using PCP. The figure depicts how $\tau 2$ locks the resource at time making $\tau 3$ wait for the lock up to time 51 when the lock is released by $\tau 2$. This waiting leaves processor P2 idle because the only task deployed there, $\tau 3$, is waiting for the lock (known as remote blocking). Furthermore, during the time $\tau 2$ holds the lock it suffers multiple preemptions from the higher priority task $\tau 1$. As a consequence, $\tau 3$ 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 $\tau 2$ and $\tau 3$ are deployed together, say in processor P2.

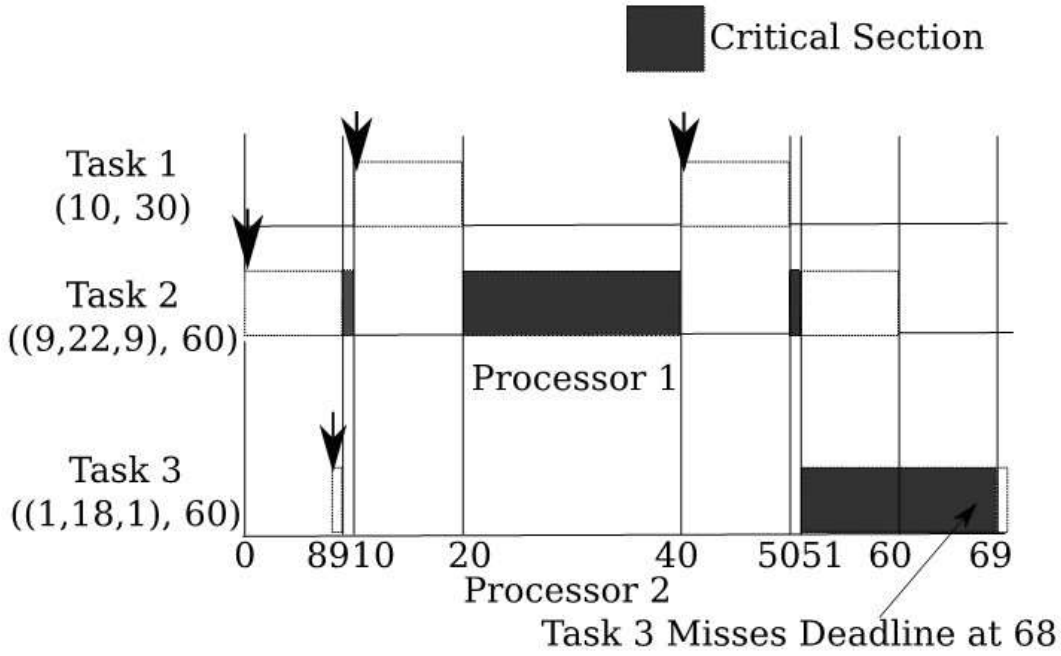


FIGURE 3.1 Global vs Local Synchronization

The key aspect of the example in Figure 3.1 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 mentioned 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.

3.1.2 Multiprocessor Priority Ceiling Protocol

We shall analyze and characterize the behavior of different execution control policies (ECPs) in Section 3.2. 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 MG. Under MPCP, when any job J acquires MG, it will execute the gcs corresponding to MG at a priority of $\pi G + \pi'$, where πG 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.

1) Jobs use assigned priorities unless within critical sections.

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

3) A job J within a global critical section (gcs) guarded by a global mutex MG 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 MG. MG can be granted to J by means of an atomic transaction on shared memory, if MG is not held by another job.

6) If a request for a global mutex MG cannot be granted, the job J is added to a prioritized queue on MG. 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 MG, the highest priority job JH

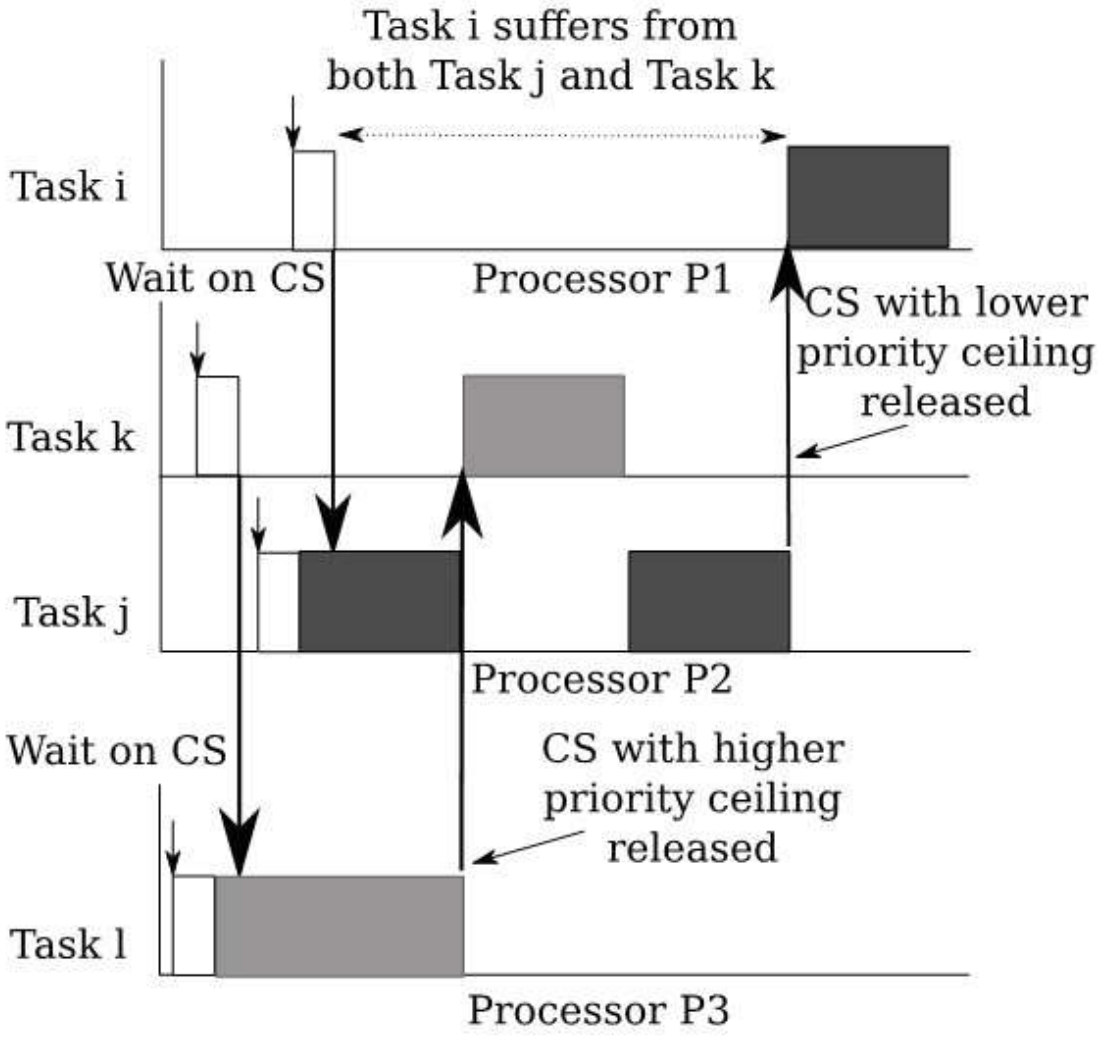


Figure 3.2 Ceiling protocol

waiting for MG is signaled and becomes eligible for execution at JH's host processor at its gcs priority. If no jobs are suspended on MG, 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.

3.1.3 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. 3.2. In this figure, Task τi in P1 could be suspended on τj in P2. Task τk on P2 could be suspended on another task τl on another processor P3. The priority ceiling of the mutex shared between τk and τl could be higher than the priority-ceiling of the mutex shared between τi and τj. In this scenario, the release of the critical section by τl would give control to the task τk, which preempts the task τj executing its critical section. The task τi waiting on τj therefore suffers from the interference due to the release of a mutex by τl.

Back-To-Back Execution of Suspending Tasks

A phenomenon that arises when tasks suspend themselves is that of "back-to-back execution". Consider the example shown in Fig. 3.3. There are three tasks τ1: ((2, 2, 0), 8), τ2: (4, 8), τ3 :((1, 2, 2), 64). Task τ1 and τ2 are assigned to processor P1.

Task τ3 is assigned to processor P2.It is easy to verify that τ1 and τ3 are schedulable. However, an anomalous scheduling behavior happens with respect to task τ2. It should be schedulable if τ1 follows a periodic release behavior, since it expects at most one preemption from a task with its same period of 8. When τ2 is released at time instant 3, however, it faces back-to-back execution due to the remote synchronization effect of τ1 and this leads to τ2 suffering the interference of a second arrival of τ1 leading to a deadline miss. This back-to-back preemption arises due to the jitter in the blocking time of task τ1 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

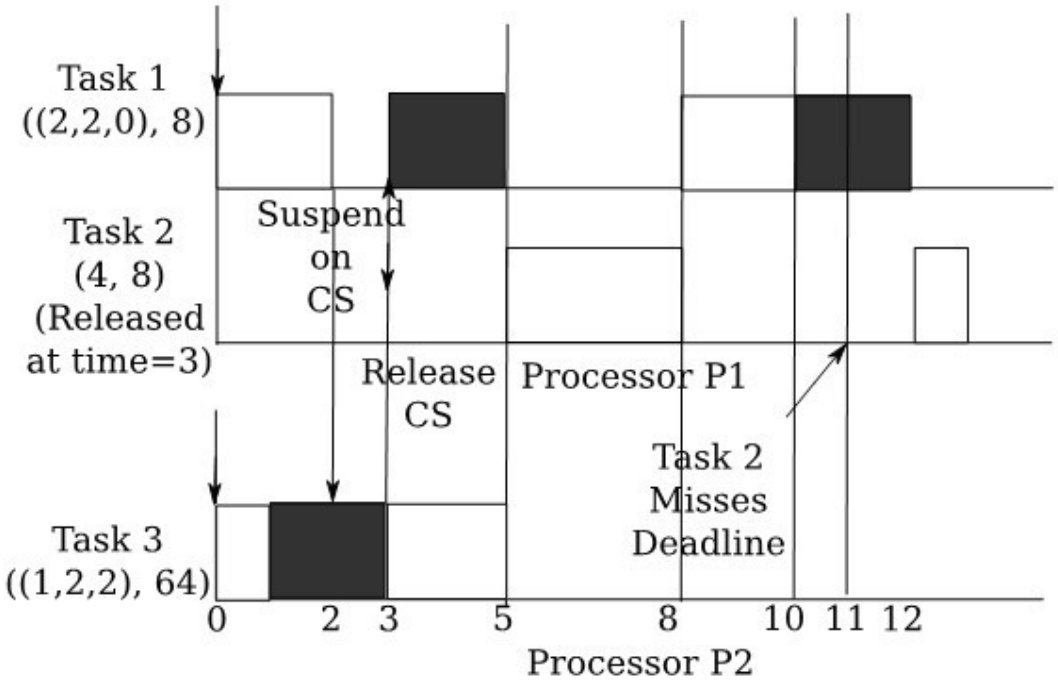


Figure 3.3 Suspended task execution

scenario shown in Fig. 3.4. Whenever task τ2 suspends, task τ3 can get a chance to execute, and it can request a lock on the global critical section shared with τ1. When τ1 releases the global critical section, τ3 preempts τ2 due to its higher priority ceiling and interferes with the normal execution of τ2 twice. In the worst case, every normal execution-segment (of duration $C_{i,k}1 \leq k \leq n_i$) of a task τi can be preempted at most once by each of the lower-priority tasks τj ($j > i$) executing their global critical sections released from remote processors.

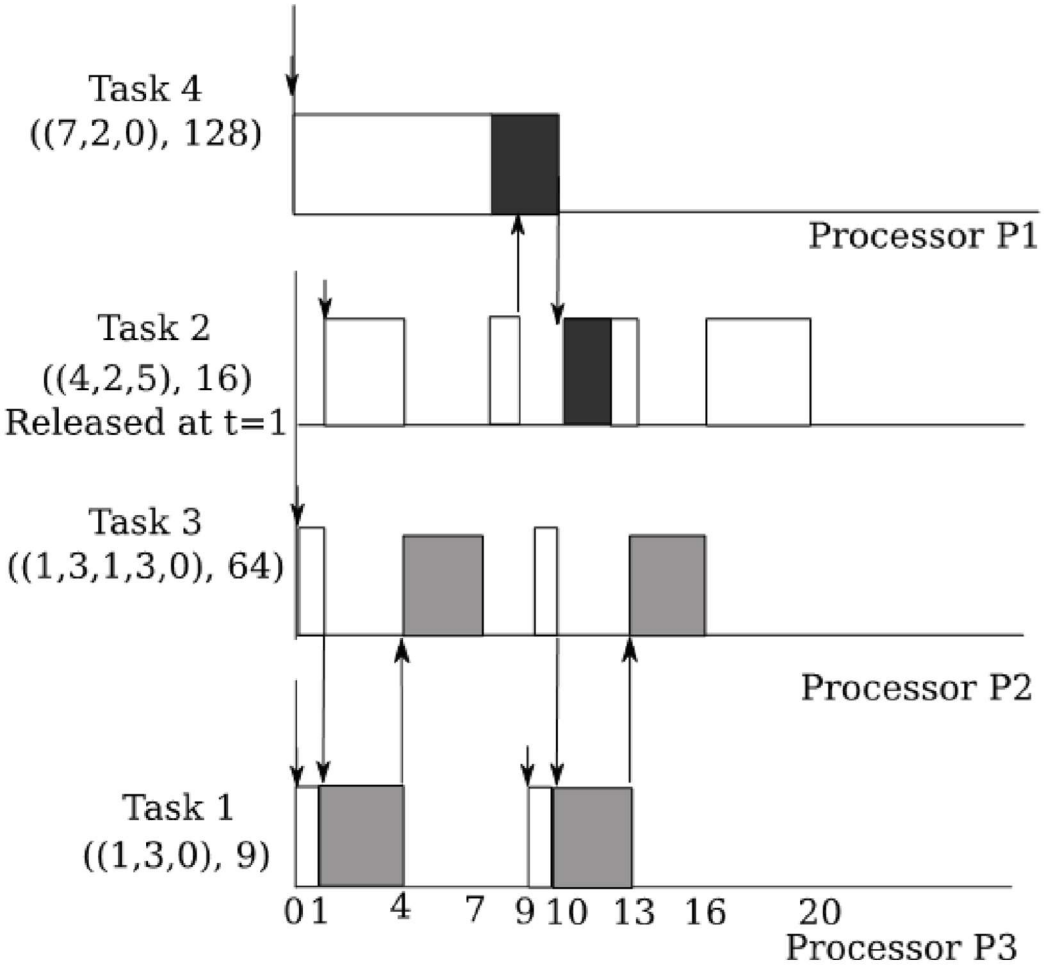


Figure 3.4 Suspended task execution

3.2 Coordination Among Scheduling, Allocation, and Synchronization

Our goal is to introduce a 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 the synchronization-aware task allocation strategy. We then analyze different execution control policies under the multi-processor priority ceiling protocol.

3.2.1 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 the binpacking 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. 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).

3.2.2 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** can also be applied for minimizing the scheduling penalty arising from synchronization. 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 Br(i,j) for the j'th critical section acquired by the task τi(the r in Br(i,j) denotes remote blocking as opposed to local blocking.

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-toback execution can result in additional interference from each higher priority task (τh). 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. 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 , 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

3.3 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 task-set. 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.

3.3.1 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 binpacking 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.

3.3.2 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 3.5 depicts the results of conducted experiments with increasing critical section sizes. 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.

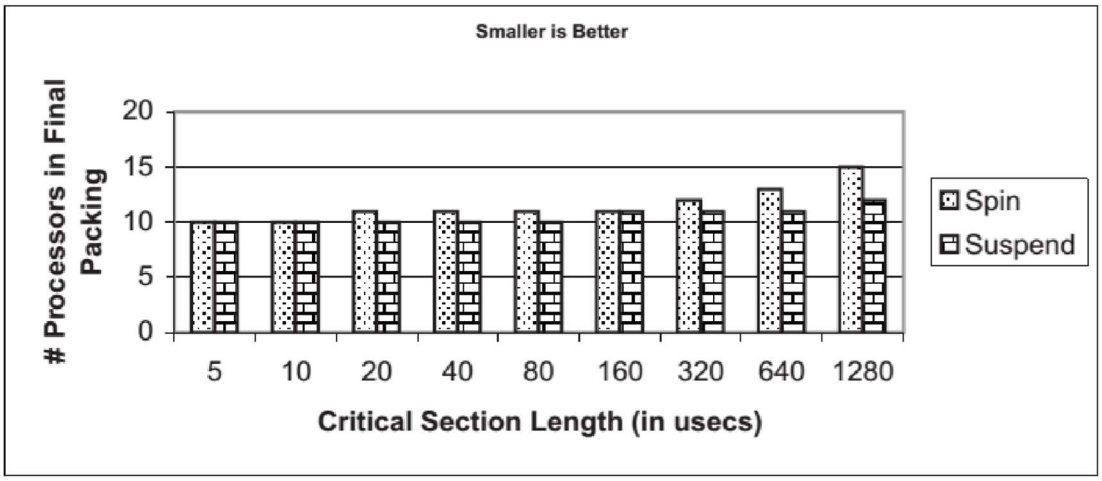


Figure 3.5 Comparison among synchronization schemes

Next, with increased 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 3.6 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. For smaller critical section lengths, the previous experiments already indicate that the performance difference is negligible, and this was verified. In Figure 3.6, 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.

3.3.3 Synchronization-Aware Task Allocation

Explained 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 task set. As a result, given algorithm does not always pay off. However, when remote blocking penalties play a major role, then our synchronization-aware packer yields significant benefits.

3.3.4 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 Pi is maximally utilized when τ' is added. Let the highest-priority task on Pi 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 Pi. 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 Pi. 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.

Chapter 4

Mixed-Criticality Systems Evaluation

4.1 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
(ii) achieve high schedulable utilization.

Such multi-objective optimization problems have been studied by trade-off approaches such as *Multiple Criteria Decision Analysis* and others more specific to resource allocation like the *QoS Resource Allocation Model*. 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

(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 task-sets, and develop an encoding of the *system workload*.

4.1.1 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 task-set described 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:

$$w = \mathrm{X}\{w_i W_g\}_{g=1}^{k}$$

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 $_k$ through the maximum overloading of all lower-criticality levels $\{l\} \forall l > g$ (at most $\mathrm{X}\, 2^{k-l} = _{l=g+1} (2^{k-g} - 1)$ 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. 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.

4.1.2 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 ascalar 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 X$d_{r,c}$). Under non-anomalous scheduling algorithms, 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 *v* (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 v* 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.

4.1.3 Illustration

Consider the set of four tasks. *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 v.

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

$w_1 = 3 =< 1,1 >$

$w_2 = 2 =< 1,0 >$

$w_3 = 3 =< 0,1 >$

$w_4 = 0 =< 0,0 >$

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.

4.2 Zero-Slack Scheduling

4.2.1 Criticality Inversion in Uniprocessors

Traditional uniprocessor scheduling algorithms such as Rate-Monotonic Scheduling (RMS) and Earliest-Deadline First (EDF) 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 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 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.

4.2.2 Scheduling Guarantee for ZS

We now define the scheduling guarantee of zero-slack scheduling. ZS performs admission control, and if admitted [1], 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.

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

Task-sets with k criticality levels schedulable under ZS are thus guaranteed to have a ductility:

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. An interested reader is referred to for a discussion on the generalized ZS algorithm and associated properties.

4.2.3 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 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 4.1: Zero-Slack-RM Scheduled Task set alignments always happen.

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

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. 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.

4.2.4 A Zero-Slack-RM Scheduling Example

Let us use an example to illustrate the characteristics of the zero-slack-RM scheduler. Tabl 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 4.1 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. 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

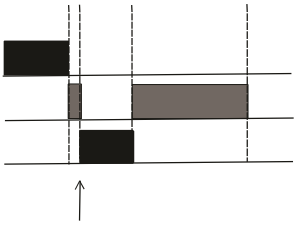


Figure 4.1: Critical Zone of *Task 1*

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 4.1, but they are unfortunately not necessarily aligned with each other.

Properties of The Zero-Slack-RM Scheduler

Theorem 1. *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 2. *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.

4.3 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

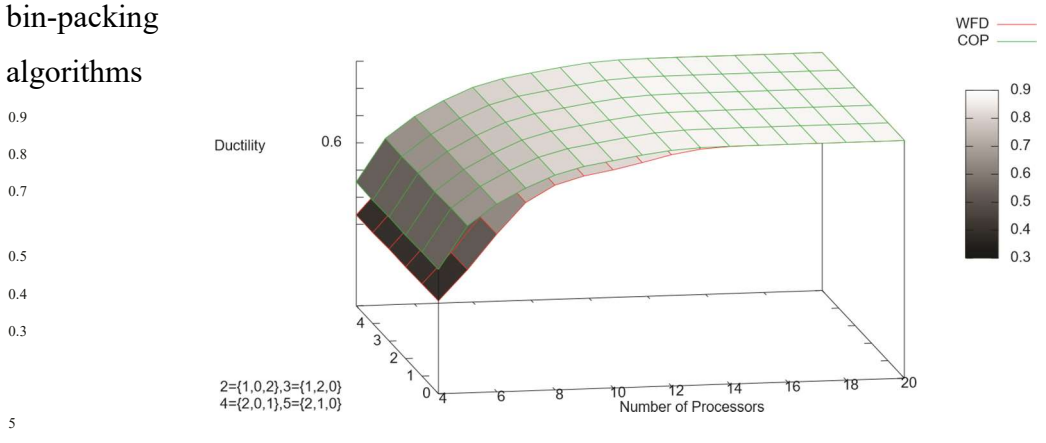


Figure 4.2: Surface of Average Performance (Harmonic Tasks)

for (non mixed-criticality) multiprocessor systems are typically evaluated exclusively along the dimension of normal schedulability. For any given taskset, the performance of different binpacking 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 4.2 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-

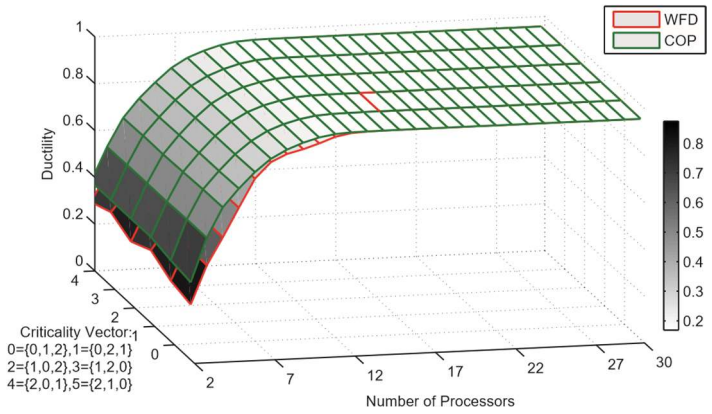


Figure 4.3: Surface of Average Performance (Task Periods: Uniform [10,100])

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 4.3 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

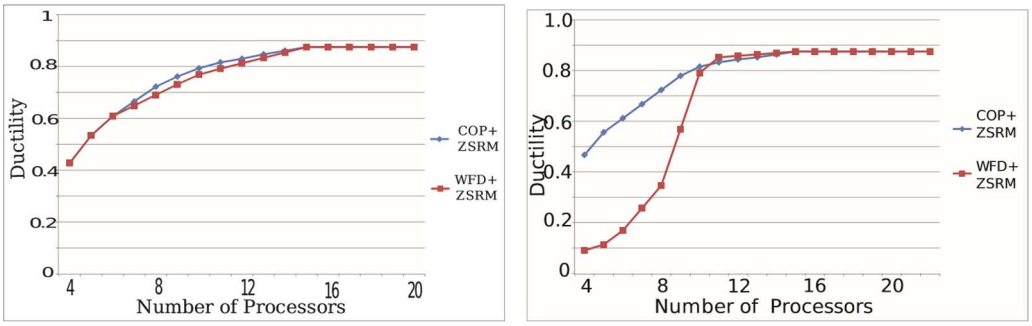
fashion between $\frac{1}{6}T_i$ and $\frac{1}{2}T_i$, and the normal computation time C_i of each task was chosen in an uniformly random fashion between $\frac{1}{12}C_i^o$and

$\frac{1}{2}C_i^o$. The obtained ductility values for COP and WFD at different criticality vectors and varying number of available processors is shown in Figure 4.3. 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 4.2. 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 4.2: Task Types

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

Table 4.2: average-case



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

Figure 4.4: Comparison at different criticality vectors

performance of COP as shown in Figure 4.3 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 4.2. 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 4.3: 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	$\frac{29}{49}$ Near Friendly

4.4 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 5

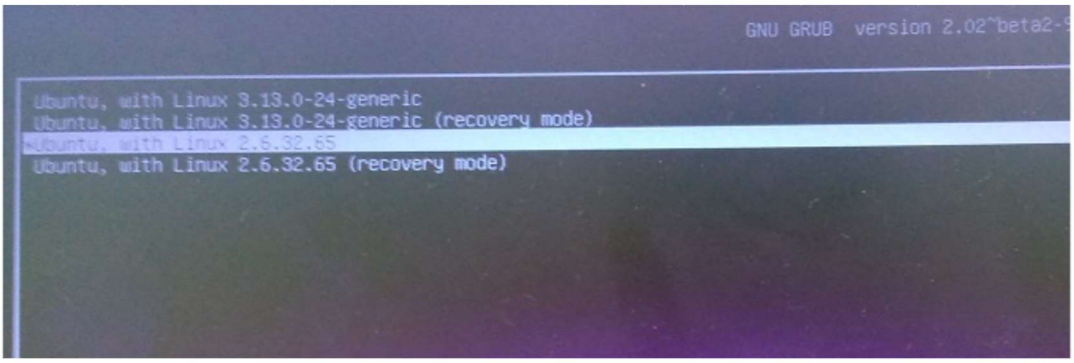
RT SYSTEM CODE SNAPSHOTS

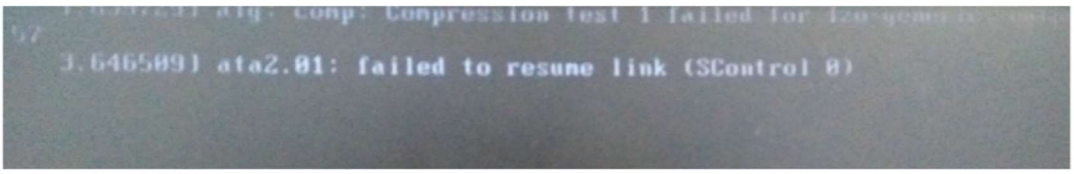
5.1 CODE SNAPSHOTS

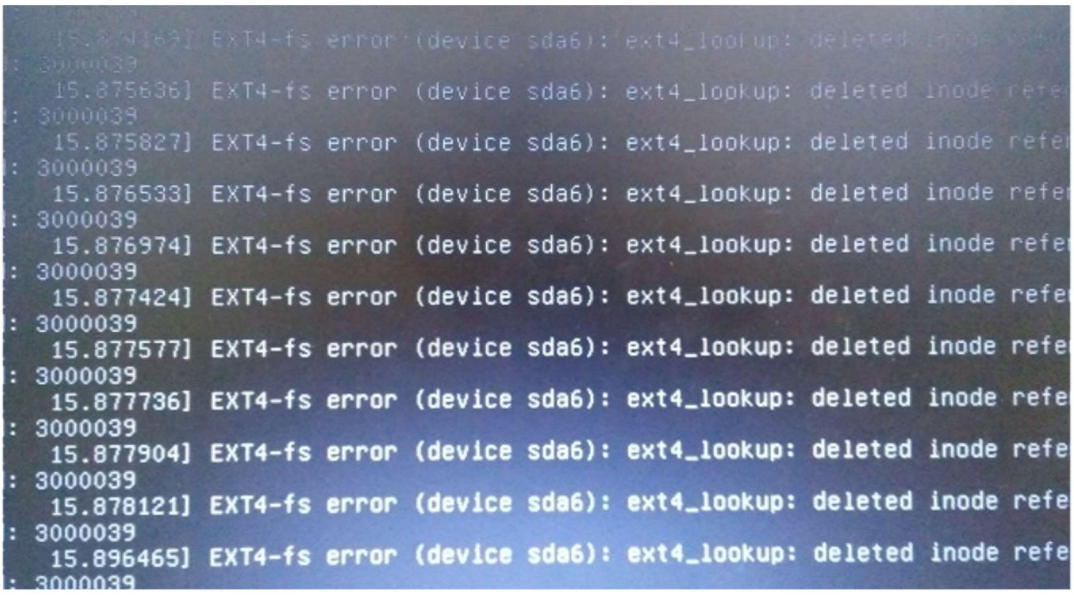
…….ABOVE CODE DEFAULT……

```
static void enqueue_rt_entity(struct sched_rt_entity *rt_se, bool head);

static void sched_rt_rq_dequeue(struct rt_rq *rt_rq)
{
        struct sched_rt_entity *rt_se = rt_rq->rt_se;

        if (rt_se && on_rt_rq(rt_se))
                dequeue_rt_entity(rt_se);
}
static void dequeue_task_rt(struct rq *rq, struct task_struct *p, int sleep)', bool head)
{
        struct sched_rt_entity *rt_se = &p->rt;

        update_curr_rt(rq);
        dequeue_rt_entity(rt_se);

        dequeue_pushable_task(rq, p);
}

if (unlikely(rt_task(rq->curr)) &&
             (rq->curr->rt.nr_cpus_allowed < 2 ||
              rq->curr->prio < p->prio) &&
             (p->rt.nr_cpus_allowed > 1)) {
                 int cpu = find_lowest_rq(p);

                 return (cpu == -1) ? task_cpu(p) : cpu;
         }
```

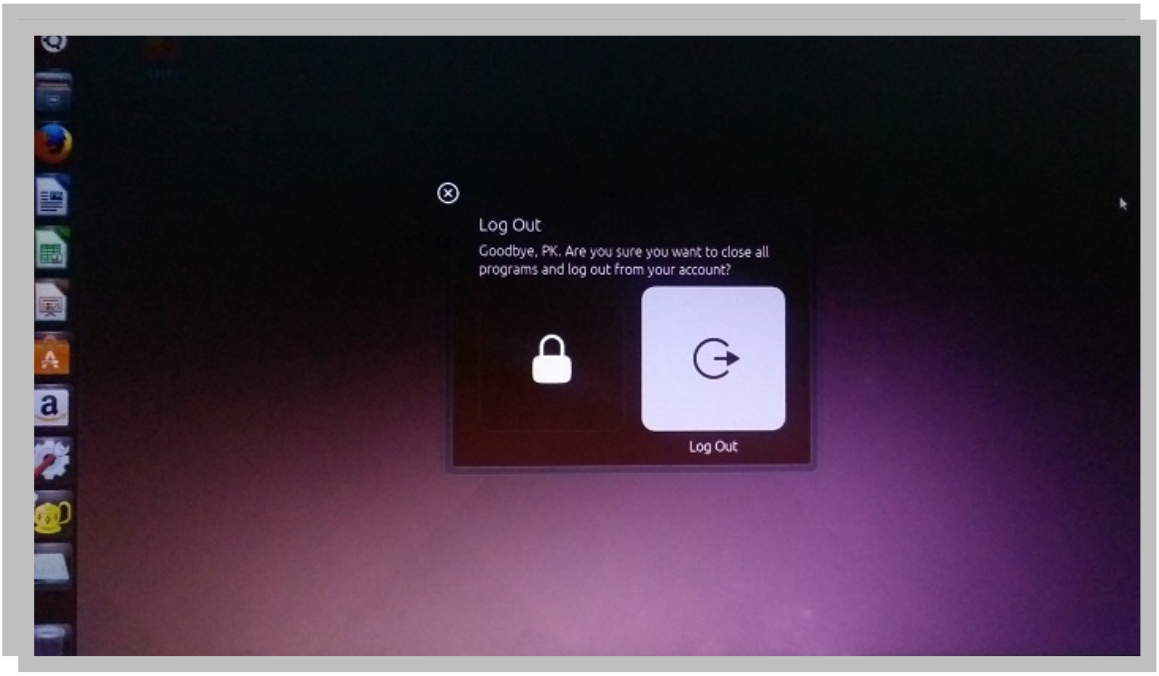
5.2 Snapshot's of Running System


GNU GRUB version 2.02~beta2-
Ubuntu, with Linux 3.13.0-24-generic
Ubuntu, with Linux 3.13.0-24-generic (recovery mode)
Ubuntu, with Linux 2.6.32.65
Ubuntu, with Linux 2.6.32.65 (recovery mode)


3.646509] ata2.01: failed to resume link (SControl 0)


15.875636] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.875827] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.876533] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.876974] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.877424] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.877577] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.877736] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.877904] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.878121] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039
15.896465] EXT4-fs error (device sda6): ext4_lookup: deleted inode refe
: 3000039


Ubuntu Desktop
tasks
About This Computer
Ubuntu Help
System Settings...
Lock
Guest Session
PK
Log Out...
Shut Down...


Log Out
Goodbye, PK. Are you sure you want to close all programs and log out from your account?
Log Out

Chapter 6

Conclusions and Future Work

In this report, we have studied the problem of scheduling and synchronizing real-time periodic tasks on multi-core processors, using fixed-priority scheduling and static offline task allocation. This report presents key advancements with respect to utilization bounds and resource augmentation bounds for multi-core processors.

6.1 Conclusions

The major contributions of this report are summarized in the following categories:

- Scheduling Independent Sequential Tasks on Multi-core Processors
- Multi-core Task Synchronization

More details of these individual contributions follow.

6.1.1 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

worstcase utilization bound of 65%. The main benefit of the introduced algorithm is its semipartitioned 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.

6.1.2 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 report, we made 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 explained 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. The experimental results shows 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.

6.2 Future Work

Multi-core processors are relatively new developments in the arena of real-time and embedded systems. There are many possible avenues for future work with regards to this report and the provided approach.

Transactional Memory is also one of the field to study upon in order to enhance the performance of multicore-processors.

References

1. Chang, Che-Wei; Chen, Jian-Jia; Kuo, Tei-Wei; Falk, H., "Real-Time Task Scheduling on Island-Based Multi-Core Platforms," Parallel and Distributed Systems, IEEE Transactions on , vol.PP, no.99, pp.1,1,doi: 10.1109/TPDS.2013.2297308
2. Winter, J.A.; Albonesi, D.H., "Scheduling algorithms for npredictably heterogeneous CMP architectures," Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on , vol., no., pp.42,51, 24-27 June 2008
   doi: 10.1109/DSN.2008.4630069
3. http://www.sei.cmu.edu/cyber-physical/research/timing-verification/multicore-scheduling.cfm
4. Parallel Computer Architecture A Hardware / Software Approach , David E. Culler, Jaswinder Pal Singh with Anoop Gupta, Morgan Kaufmann .
5. Rahman, M.M., "Process synchronization in multiprocessor and multi-core processor," Informatics, Electronics & Vision (ICIEV), 2012 International Conference on , vol., no., pp.554,559, 18-19 May 2012
6. Stoif, C.; Schoeberl, M.; Liccardi, B.; Haase, J., "Hardware synchronization for embedded multi-core processors," Circuits and Systems (ISCAS), 2011 IEEE International Symposium on , vol., no., pp.2557,2560, 15-18 May 2011
7. Joseph, A.; Dhanwada, N.R., "Process Synchronization in Multi-core Systems Using On-Chip Memories," VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on , vol., no., pp.210,215, 5-9 Jan. 2014
8. Lakshmanan, K.; Kato, S.; Rajkumar, R., "Scheduling Parallel Real-Time Tasks on Multi-core Processors," *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st* , vol., no., pp.259,268, Nov. 30 2010-Dec. 3 2010

9. Vaidya, V.G.; Sah, S.; Ranadive, P., "Optimal task scheduler for multi-core processor," *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on* , vol.1, no., pp.V1-1,V1-4, 3-5 Oct. 2010

10. http://i.stanford.edu/pub/cstr/reports/cs/tr/99/1624/CS-TR-99-1624.pdf

11. http://infolab.stanford.edu/pub/cstr/reports/csl/tr/98/759/CSL-TR-98-759.pdf