INTEGRATING PRAGMATIC CONSTRAINTS AND BEHAVIORS INTO REAL-TIME SCHEDULING THEORY

Bipasa Chattopadhyay

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2015

Approved by:

Sanjoy K. Baruah

James H. Anderson

Kevin Jeffay

Montek Singh

Nathan W. Fisher

ABSTRACT

BIPASA CHATTOPADHYAY : Integrating Pragmatic Constraints and Behaviors into Real-Time Scheduling Theory
(Under the direction of Sanjoy K. Baruah)

Scheduling theory has been studied and developed extensively in prior research. In some existing scheduling theory results, the focus is primarily on demonstrating interesting theoretical properties, thus these results are not always cognizant of pragmatic constraints. We seek to determine how existing scheduling theory can be improved with respect to pragmatic constraints and behaviors.

The goal of this research is to study and design scheduling algorithms for scheduling real-time workload under constraints and behaviors found in real-time systems. Based on our study we derive a scheduling algorithm for partitioning a collection of real-time tasks in a manner that is cognizant of *multiple resource constraints*. We apply the above scheduling algorithm for partitioning *mixed-criticality* tasks.

In real-time systems the scheduling algorithm must schedule workload such that all timing constraints are met; we verify this using *schedulability tests*. We describe schedulability tests for each of the scheduling algorithms that we derive. We also propose a new schedulability test for an existing scheduling algorithm that is commonly used in real-time systems research for scheduling tasks with *limited-preemptivity*.

Finally, we propose a scheduling algorithm and schedulability test for scheduling real-time workload on processors that allow *dynamic overclocking*.

Dedicated to my family and friends.

ACKNOWLEDGEMENTS

I would like to thank the following people for their continued help and support through out my graduate studies.

Sanjoy Baruah, my advisor, has been instrumental in my effectiveness as a graduate student. In my graduate career what I needed the most was to develop the ability to think rigorously about algorithms and I have benefited immensely from Sanjoy Baruah's guidance.

My advising committee: James Anderson, Kevin Jeffay, Montek Singh, and Nathan Fisher have provided helpful advice and insightful ideas in the process of completing the work done in this dissertation.

All members of the Real-time Systems group for discussions both technical and otherwise. Some of the members with whom I have interacted with and learnt from the most are: Björn Brandenburg, Andrea Bastoni, Cong Liu, Haohan Li, Jeremy Erickson, Glenn Elliott, Mac Mollison, Zhishan Guo, and Bryan Ward.

Jodie Turnbull, the student services manager at our department, has helped me in every step of my graduation. She is very approachable and patient.

The members of the Graduate Women in Computer Science (GWiCS) club for their enthusiasm and support. Some of my most memorable events with them are when we participated in Pearl hacks (March 2014) as a team, and when we organized the Women in Computing Research Workshop (February 2015).

This research has been supported by NSF grants CNS 1016954, CNS 1115284, and CNS 1218693, and CNS 1239135; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

In real-time systems logical correctness and temporal correctness are equally important. Some examples of real-time systems are safety-critical computer systems such as those in aircrafts, automobiles, nuclear reactors, and railway switching systems. In all the above examples the computation that is performed by the system has to be logically correct, that is, the system needs to compute the correct result (logical correctness in required in any general computer system), and the computation needs to be temporally correct, that is, the computation must complete within a certain time frame. The lack of temporal correctness in such systems could lead to life-threatening events.

There are several aspects that need to be considered in order to implement real-time systems. In real-time scheduling theory two types of algorithms, which are crucial for implementing real-time systems, are studied extensively. First, a scheduling algorithm controls how any given workload is scheduled to run on a computing platform. Second, a schedulability test algorithm determines whether the schedule generated by the scheduling algorithm ensures that all timing constraints are met. Every scheduling algorithm should have an associated schedulability test.

In this dissertation we study real-time scheduling theory and incorporate pragmatic constraints and behaviors found in real-time systems. The pragmatic constraints and behaviors that we consider are motivated in the following section.

1.1 Motivation

The advantages of multicore technologies, and the increase in functionality of real-time systems has caused an increasing trend towards the use of multicore CPUs and multiprocessor platforms for implementing real-time systems. One approach for implementing real-time systems on multiprocessor platforms is to statically assign real-time workload to processors such that only the workload

that is assigned to a processor can execute on that processor. This is the well known ***partitioned scheduling*** approach. While partitioning it is necessary to ensure that sufficient amounts of all the resources required by the workload are available on a processor. Some key resources include computing capacity, local (per-core) memory, and network bandwidth. We study and evaluate partitioning scheduling approaches on platforms in which an arbitrary (but fixed) number of different types of resources are available in limited quantities upon each processor.

Multiprocessor platforms are also used to implement ***mixed-criticality*** real-time systems. In such systems, workload that may be of different degrees of importance or criticality are implemented upon a common platform. For example, the safety critical workload that *must* meet their timing guarantees, and the non-safety critical workload that may cause suboptimal behavior if they do not meet their timing guarantees, are implemented upon a common platform. Workloads with different criticalities are validated to different levels of assurance. In case of the safety critical workload, we need high confidence that the timing guarantees are met. Thus, pessimistic assumptions (i.e inflated values) are used to denote the computing capacity required by the workload. In the case of the non-safety critical workload less pessimistic assumptions are used to denote the computing capacity required by the workload. However, if we provision the system to operate under the pessimistic assumptions of the safety-critical workload, then we waste a lot of the computing capacity available on the processors, because during run-time the safety critical workload may use much less computing capacity than the pessimistic estimate. The question then arises as to how can we better utilize the available computing capacity? One way to do this is to design scheduling algorithms for mixed criticality systems that schedule all the workload under less pessimistic assumptions, while ensuring that only the safety critical workload can be scheduled under pessimistic assumptions if and when needed. Scheduling in this manner effectively utilizes the available computing capacity, which further enables more software functionality to be implemented on the same hardware. We extend our knowledge of partitioned scheduling to derive a partitioned scheduling approach for mixed criticality real-time systems.

An important design choice that arises upon scheduling on uniprocessor and multiprocessor platforms alike is whether preemptions are enabled (fully-preemptive or simply preemptive scheduling) or disabled (non-preemptive scheduling) during task execution. Both fully-preemptive and non-preemptive scheduling have pros and cons. For example, fully-preemptive scheduling provides better *schedulability*. However, in the case of fully-preemptive scheduling, the *overheads* incurred at run-time tend to be larger. Further, in fully-preemptive scheduling access to shared resources need to be arbitrated using non-trivial synchronization protocols, whereas in non-preemptive scheduling the synchronization protocols are simpler to implement. An alternative to fully-preemptive and non-preemptive scheduling is a restricted model of preemptive scheduling referred to as ***limited-preemptive scheduling***. Limited-preemptive scheduling is an approach to incorporate the positive aspects of both fully-preemptive and non-preemptive scheduling. We study scheduling under limited-preemptive scheduling.

On a computing platform the frequency of the processor is an important property of the computing platform. The frequency rating provided by the processor manufacturer is usually conservative, and a processor can be operated at a frequency higher than the specified frequency rating; this is called overclocking. Overclocking a processor improves its performance, for example some workloads may be able to satisfy their timing guarantees only with the help of overclocking. However, uncontrolled overclocking may overheat the processor, and may finally damage the processor. In ***dynamic overclocking*** a processor can overclock only if the conditions are favorable. For example, the *Turbo Boost technology by Intel* (Rotem et al., 2012), enables overclocking given a suitable workload and suitable operating conditions. We design a system model that justifiably reflects the dynamic overclocking behavior allowed on a processor and study how real-time workloads can be scheduled on such processors.

1.2 Thesis Statement

In my research we study existing real-time scheduling theory that has been developed within the real-time systems community. We also study existing scheduling theory that has been developed in

the "traditional" scheduling (Operations Research/Theoretical Computer Science) community; some of this theory is directly applicable to real-time systems, some is not, and some needs significant adaption. Based on the analysis, we derive new scheduling algorithms and compare them on the basis of theoretical metrics, and schedulability experiments. One of the theoretical metrics we use to evaluate algorithms is the *resource augmentation* bound (Kalyanasundaram and Pruhs, 2000), which compares the performance of an algorithm with that of a hypothetical optimal one, under the assumption that the algorithm under discussion has access to *more resources* than an optimal algorithm. This leads me to my thesis statement which is as follows:

A careful analysis of existing scheduling theory and evaluation of scheduling algorithms using theoretical metrics, such as resource augmentation bound and worst-case run-time complexity, and evaluation using schedulability experiments aid in the design and implementation of effective real-time scheduling algorithms that are cognizant of pragmatic constraints and behaviors.

1.3 Contributions

We derive two partitioned scheduling approaches and compare both the approaches using theoretical metrics. Based on our analysis we identify the partitioned scheduling approach that is preferable for implementing partitioned scheduling on processors that have an arbitrary (but fixed) number of different types of limited resources. We derive a heuristic improvement for the preferred partitioned scheduling approach, and evaluate the heuristic improvement based on schedulability experiments.

We then derive a partitioned scheduling approach for a mixed-criticality system. We also derive the resource augmentation bound, run-time complexity, and pragmatic improvements for the partitioned scheduling approach. We use schedulability experiments to evaluate the pragmatic improvements.

From our study of prior work on limited-preemption scheduling, we found that there already exist schedulability tests for some of the well known partitioned scheduling approaches with limited preemptions. We did not however, encounter a *demand-based* schedulability test under limited-

preemption scheduling for a scheduling approach that is not based on partitioned scheduling but is widely used in real-time systems research. We therefore derive a new schedulability test and perform schedulability experiments to emphasize the different scenarios in which our schedulability test can be used.

Finally, we study and identify a system model that justifiably reflects the dynamic overclocking behavior allowed on a processor. Further, we derive an *offline* scheduling algorithm and a schedulability test for scheduling real-time workload on processors that allow this behavior.

Overview. In Chapter 2, I provide some background on real-time systems. I then describe the work on partitioned scheduling in Chapter 3, mixed-criticality scheduling in Chapter 4, limited-preemption scheduling in Chapter 5, and dynamic overclocking in Chapter 6. I present a summary of the work accomplished in this dissertation in Chapter 7.

CHAPTER 2: BACKGROUND

In real-time systems it is necessary to ensure that a scheduling algorithm is able to schedule a given workload on a computing platform such that all the timing constraints are met. In order to make such guarantees about timing constraints we need to first, identify the relevant characteristics of the workload, the computational platform, and the scheduling algorithm that affect the timing constraints of the system (Liu, 2000). We can then build a model for the workload and the computational platform that justifiably abstracts away any information that is not relevant, or can be simplified, from the perspective of ensuring timing constraints. Once we have derived the workload and platform model we can describe a scheduling algorithm that determines how to schedule the workload on the platform. (We describe only those characteristics of a scheduling algorithm that are necessary to validate the timing constraints of the workload.) Given a scheduling algorithm, we can use a schedulability test to determine if the timing constraints are indeed met.

We first describe a simple workload model that we call the one-shot job model.

2.1 One-shot job model

A real-time job, referred to as a job for brevity, is denoted as J_i, where the subscript i is used to identify one job from another when referring to a set of jobs. We characterize a job $J_i = (a_i, c_i, d_i)$, by three parameters: an arrival time a_i, a worst-case execution time (WCET) c_i, and an absolute deadline d_i. Figure 2.1 depicts each of these parameters for two jobs J_1 and J_2.

The arrival time a_i of a job J_i is the time at which a job arrives. A job is scheduled correctly if it can execute for up to c_i time units in the interval $[a_i, d_i]$, that is in the interval between its arrival time a_i and its deadline d_i. In Figure 2.1, jobs J_1 and J_2 are scheduled correctly.

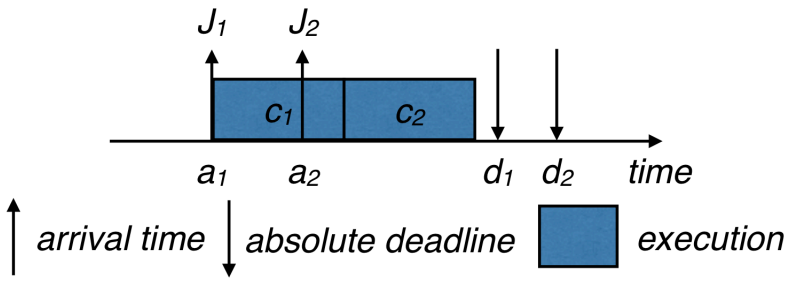


Figure 2.1: One-shot job model

In this simple model all jobs are independent. Thus, the execution of one job is not dependent on the execution of another job. Also, all jobs are fully-preemptive. A scheduling algorithm determines which job runs on a processor at any given time.

2.2 Liu and Layland Task Model

An implicit-deadline sporadic task (Liu and Layland, 1973) also known as *Liu and Layland* task, is characterized by an ordered pair of parameters: a worst-case execution time (WCET) and a minimum inter-arrival separation (that is, for historical reasons, also called the period of the task). Let $\tau_i = (C_i, T_i)$, denote an implicit-deadline sporadic task with WCET C_i and period T_i. Such a task generates a potentially infinite sequence of jobs, with the first job arriving at any time and subsequent jobs arriving at least T_i time units apart. Each job has an execution requirement no greater than C_i; this must be met by its absolute deadline that occurs T_i time units after the job's arrival. Figure 2.2 shows a possible arrival sequence for an implicit-deadline sporadic task τ_i.

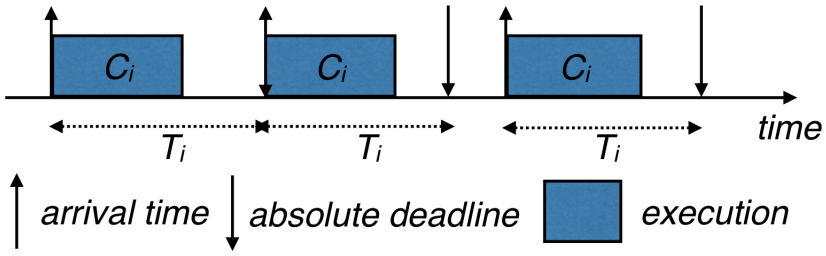


Figure 2.2: Arrival sequence of an implicit-deadline task $\tau_i = (C_i, T_i)$. Note that T_i is the minimum inter-arrival separation between two consecutive jobs.

We use the term *utilization* to denote the ratio of the WCET parameter of a task to its period. The utilization of task τ_i, denoted as u_i is equal to C_i/T_i. An implicit-deadline sporadic task system or task set τ, comprises a finite collection of sporadic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and a task is denoted by τ_i (i = 1 to n).

By changing certain characteristics of the implicit-deadline sporadic task model we can obtain other task models. For example, in certain cases we may a priori know that subsequent jobs of a task arrive exactly T_i time units apart. Such a task model is referred to as a periodic task model.

Further, an additional parameter D_i can be used to represent the relative deadline of a task τ_i. In an implicit-deadline sporadic task $D_i = T_i$, therefore this additional parameter is omitted. However, the relationship between a task's relative deadline D_i and its period T_i may be such that, $D_i \leq T_i$. In this case the sporadic task is said to have a constrained deadline (Baruah et al., 1990; Mok, 1983). Each constrained-deadline sporadic task τ_i, is represented by three parameters: (C_i, D_i, T_i), where $D_i \leq T_i$. The execution requirement of a job of such a task must be met by its absolute deadline, which occurs D_i time units after the job's arrival.

It may also be the case that tasks may have deadline greater than period, also referred to as arbitrary deadlines. Yet another additional parameter a_i, can be used to represent the arrival time of the first job of each task τ_i in a task system. Such a task system is referred to as a concrete task system.

In Chapter 3 we extend the implicit-deadline sporadic task model to incorporate an additional parameter that represents the memory requirement of each task. In Chapter 4 we describe a mixed-criticality implicit-deadline sporadic task model. In Chapter 5 we extend the constrained-deadline sporadic task model to represent the *preemptive* and *non-preemptive* execution requirement of each task. These model extensions are described in their respective chapters for better readability. In Chapter 6 we use the one-shot job model described in Section 2.1.

2.3 Computing Platform model

An important characteristic of a computing platform, from the perspective of scheduling jobs or tasks, is the computing capacity of the platform. It is very common in real-time systems research to consider that a uniprocessor platform has a computing capacity, or speed of 1. This implies that the utilization $u_i = C_i/T_i$ of a task τ_i, which is the ratio of the task's worst-case execution time C_i and period T_i, is essential equal to the portion of the computing capacity available on a uniprocessor that needs to be exclusively used by this task, in the worst-case, for ensuring that each job of the task meets its deadline.

A multiprocessor platform can be classified as one of the following platform models, depending upon the relationship between the computing capacities of different processors available on the platform:

- Identical: all the processors are identical, in the sense that they all have the same computing capacity or speed.
- Uniform: each processor is characterized by its own computing capacity, with the interpretation that a job executing on a processor of speed s for t time units completes $s \times t$ units of execution.
- Unrelated: each processor P_j, has an execution rate $r_{i,j}$ associated with each job-processor ordered pair (J_i, P_j), with the interpretation that job J_i completes $(r_{i,j} \times t)$ units of execution by executing on processor P_j for t time units.

In most of this work we use the identical multiprocessor platform model, with m identical processors each with computing capacity equal to 1. In Chapter 3 we incorporate resource constraints, in addition to limited computing capacity, into the identical multiprocessor model. In Chapters 4, and 5 we use the identical multiprocessor model as is. In Chapter 6 we use a uniprocessor model that we describe in Section 6.1.

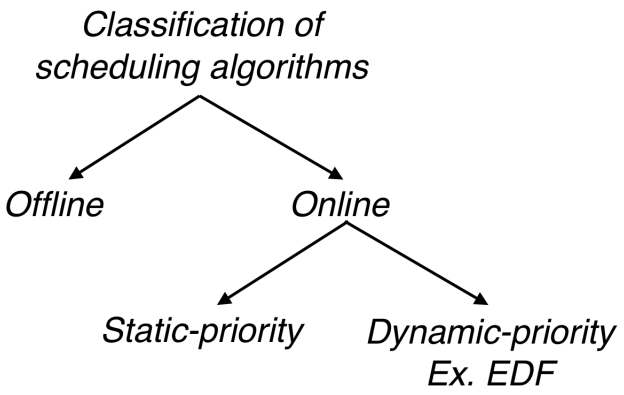


Figure 2.3: Classification of scheduling algorithms.

2.4 Scheduling Algorithms and Schedulability tests

The scheduling algorithm, sometimes called a scheduler, decides which job should execute on a processor at any given time. Scheduling algorithms are generally required to be simple and fast because they execute alongside other jobs.

There is a broad classification of scheduling algorithms based on whether the schedule is generated prior to run-time or at run-time. In static or *offline* scheduling, the schedule is generated prior to run-time. Offline scheduling often requires the exact knowledge of the arrival time of all the jobs that need to be scheduled, or it makes pessimistic assumptions about the arrival times of jobs. In dynamic or *online* scheduling the schedule is generated at run-time. In online scheduling, the decision to schedule a job is made after the job has arrived. Therefore, it is not necessary to know the exact arrival time of jobs.

In online *dynamic-priority* scheduling, the temporal behavior of all jobs that are ready to execute is taken into account to determine which job should be prioritized over others for execution. For example, in the Earliest Deadline First (EDF) scheduling algorithm the job of some task τ_i with the earliest absolute deadline is given the highest priority and is scheduled to execute on a processor. Subsequent jobs of the same task τ_i may have different priorities depending upon their arrival time and the absolute deadline of all jobs that are ready to execute. In contrast, in *static-priority scheduling* priorities are assigned to tasks such that all jobs of some task τ_i are always prioritized

over all jobs of some task τ_j. Both dynamic- and static- priority scheduling have pros and cons. In static-priority scheduling a safety-critical task can be prioritized over a non-safety critical task. Thus, a job of a safety-critical task will always have a higher priority. However, dynamic-priority scheduling is often able to better utilize the computing capacity available on a computing platform. We focus on dynamic-priority scheduling.

A task set τ is deemed *schedulable* by a scheduling algorithm A if algorithm A can schedule all jobs generated by τ such that all jobs complete execution by their deadline, and all jobs adhere to the specifications of the task model. A task set τ is deemed *feasible* if it is schedulable in accordance with some scheduling algorithm A. Note that feasibility is not defined with respect to any specific scheduling algorithm. A scheduling algorithm is considered *optimal* if it can schedule any feasible task set.

A real-time scheduling algorithm should be able to a priori guarantee if a given task set τ is schedulable. A schedulability test indicates whether a specific algorithm can correctly schedule a given task set. Since schedulability tests are performed prior to run-time they do not necessarily have to be efficient. However, they should be computationally tractable. There is class of tests called feasibility tests that indicate whether some algorithm can correctly schedule a given task set. We do not study feasibility tests in this work.

A *necessary* schedulability test for algorithm A guarantees that if a task set does not satisfy the schedulability test, then the task set is not schedulable by algorithm A. A *sufficient* schedulability test for algorithm A guarantees that if a task set is schedulable by algorithm A, then the task set satisfies the schedulability test.

A schedulability test for algorithm A, can be necessary and sufficient. Such a schedulability test is also referred to as an *exact* schedulability test. An exact schedulability test and a sufficient schedulability test, both guarantee that if a task set satisfies the schedulability test, then it is schedulable by algorithm A. An exact schedulability test also guarantees that if a task set does not satisfy the schedulability test, then it is not schedulable by algorithm A. However, if a task set does not satisfy a sufficient schedulability test, then it may or not be schedulable by algorithm

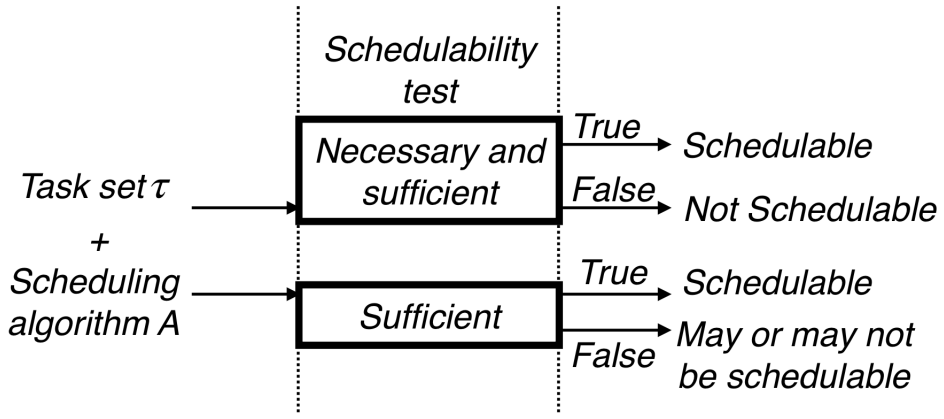


Figure 2.4: Necessary and/or sufficient schedulability test.

A. The distinction between a necessary and sufficient (exact) schedulability test, and a sufficient schedulability test is shown in Figure 2.4

2.4.1 EDF scheduling

Since all the scheduling approaches described in this paper are derived from the EDF scheduling algorithm, we provide a brief description of the EDF scheduling algorithm. In EDF scheduling, when a job arrives, it is queued in a priority queue that is ordered by shortest time remaining to absolute deadline. The job at the front of the priority queue is chosen for execution and removed from the priority queue. Thus, the job with the earliest absolute deadline is chosen for execution. When the job completes, the next job that is in front of the priority queue is chosen for execution, and so on. Note that a job is *preempted* and re-queued on the priority queue only if a job with an earlier absolute deadline arrives before it completes execution.

In the above description of the EDF scheduling algorithm we assume that preemptions are allowed. This is called preemptive EDF scheduling. There is a variant of the EDF scheduling algorithm called non-preemptive EDF. In this variant preemptions are not allowed. (A detailed discussion on the pros and cons of allowing preemptions is presented in Chapter 5.) When we refer to EDF scheduling we are referring to preemptive EDF scheduling unless stated otherwise.

The EDF scheduling algorithm is optimal for scheduling a set of one-shot jobs on a uniprocessor (Liu, 2000). The EDF scheduling algorithm is also optimal for scheduling a collection of tasks on a uniprocessor (Liu and Layland, 1973). A specific result is stated below.

Theorem 2.1. *(Liu and Layland, 1973). An implicit-deadline sporadic task system τ can be scheduled under EDF on a uniprocessor if and only if the total utilization U, of all tasks τ_i in the task system τ, sums up to at most one:*

$$U = \sum_{i=1}^{n} u_i \leq 1. \tag{2.1}$$

□

Note that if $U > 1$ then the task system is not feasible on a uniprocessor. Thus, to prove the optimality of EDF it is sufficient to show that if $U \leq 1$ then the task system is schedulable on a uniprocessor under EDF. The proof of the above theorem is able to show that. From Theorem 2.1, we can also claim that Equation 2.1, is a necessary and sufficient schedulability test to verify if a task system τ, is schedulable on a uniprocessor under EDF.

It has been shown in (Jeffay et al., 1991) that non-preemptive EDF is optimal for scheduling implicit-deadline sporadic task systems on a uniprocessor. A necessary and sufficient schedulability is also described in (Jeffay et al., 1991).

2.4.2 Partitioned vs. Global scheduling

Due to multiple reasons including cost, energy-efficiency, and thermal properties, the use of multicore/multiprocessor platforms has become widespread. On such platforms there are two main scheduling approaches: partitioned and global scheduling.

In partitioned scheduling each task is statically assigned to a processor. Every job of a task executes on the processor to which the task is assigned. In global scheduling a job of a task may execute on any processor. A job may also *migrate* before it completes execution and execute on a different processor; this is called intra-job migration.

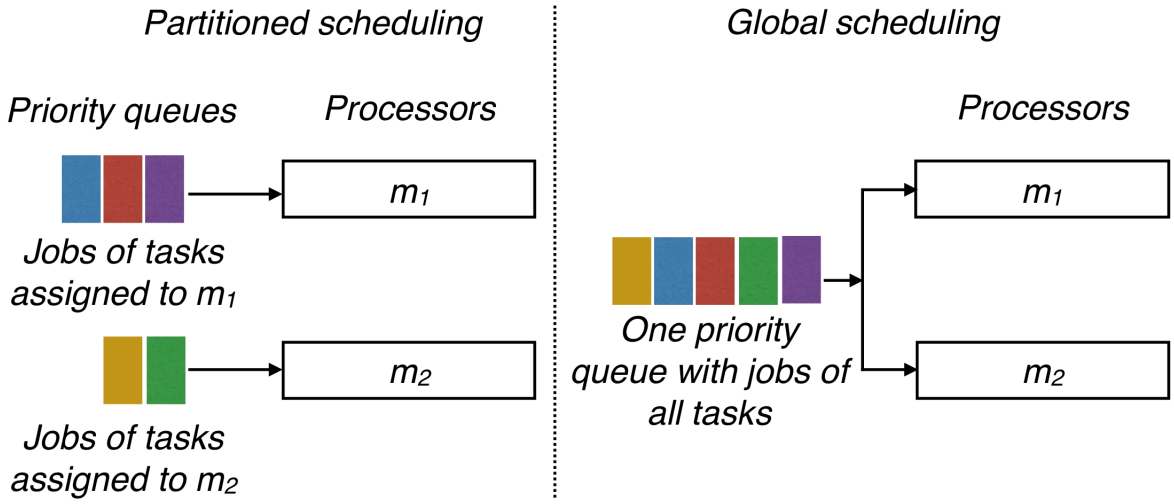


Figure 2.5: Partitioned vs. Global scheduling

Suppose that the computing platform consists of m processors. In partitioned scheduling we need to first solve a partitioning problem to assign tasks onto m processors; this is related to the bin-packing problem and is NP-hard in the strong sense. Thus, an optimal solution to the partitioning problem is intractable. This leads to utilization loss in partitioned scheduling. However, once the tasks are assigned to the processors the problem reduces to multiple uniprocessor scheduling problems. If EDF is used to schedule the tasks on each processor, then it is referred to as partitioned EDF scheduling. From Theorem 2.1 we know that EDF scheduling is optimal for scheduling implicit-deadline sporadic tasks on uniprocessors.

In global scheduling the top m highest priority jobs are scheduled on the processors. Thus, in global EDF scheduling the m jobs with the earliest deadlines are scheduled on the m processors. In *hard real-time* scheduling, all jobs must complete execution by their deadline. Under this constraint, global EDF scheduling incurs utilization loss due to pessimism in any schedulability test. However, in the case of *soft real-time* scheduling, global EDF scheduling has no utilization loss. In soft real-time scheduling the execution of a job completes within a deadline tardiness bound. We study *hard real-time* scheduling.

We now compare the *run-time overheads* incurred in partitioned and global scheduling. Following are examples of some run-time overheads incurred in a real-time system (Bastoni et al., 2010)-

- Release overhead- time needed to service the interrupt routine that is responsible for releasing jobs at correct times.
- Scheduling overhead- time spent while selecting the next job to execute and re-queuing the previously-scheduled job.
- Context switching overhead- time spent in switching the execution stack and processor registers.
- Preemption and migration overhead- when a job gets preempted and starts executing at a later time on the same (different) processor there may be loss of cache affinity. Any cost incurred due to the loss of cache affinity is called a preemption (migration) overhead.

Run-time overheads affect the timing behavior of tasks. One way to incorporate the delays incurred due to run-time overheads is to inflate the WCET of tasks (Bastoni et al., 2010). It has been shown in (Bastoni et al., 2010), that in the worst-case (a fully utilized system) the preemption and migration overhead under global EDF scheduling and partitioned EDF scheduling are comparable. Release and scheduling overheads are usually larger in the case of global EDF scheduling when compared to partitioned EDF scheduling. This is because, in partitioned EDF scheduling each processor has its own priority queue, and jobs only access the priority queue of the processor they are assigned to. Whereas, in global EDF scheduling there is one global priority queue, and there is contention to access the global priority queue every time any job is released or preempted. This increases the delay incurred when a job is released or preempted. Thus, if overheads are incorporated by inflating the WCET parameter of tasks, the WCET parameter of a task τ_i would be lesser if it were being scheduled under partitioned EDF scheduling as apposed to global EDF scheduling.

CHAPTER 3: PARTITIONED SCHEDULING

We study the partitioned Earliest Deadline First (EDF) scheduling of implicit-deadline sporadic task systems (Liu and Layland, 1973) on multiprocessor platforms. Earlier work on this problem (see, e.g., (Oh and Baker, 1998; Lopez et al., 2004)) focused exclusively on processors in which only one resource – computing capacity – is available in limited amounts. The partitioning algorithms described in earlier works (Oh and Baker, 1998; Lopez et al., 2004) can guarantee that the cumulative computing capacity of all the tasks assigned to a processor does not exceed the total computing capacity of the processor. However additional resource constraints (e.g., the amount of memory available on each processor) were not considered.

Some more recent work (e.g., (Baruah and Fisher, 2004; Fisher et al., 2005)) based on approaches such as dynamic programming (Baruah and Fisher, 2004) and integer linear programming (Fisher et al., 2005) have considered processors with limited memory in addition to limited computing capacity. Each of the tasks were characterized by a memory requirement, in addition to characterizations of its computational requirements (which was modelled according to the *implicit-deadline sporadic task* model (Liu and Layland, 1973)). In this chapter we study the partitioning problem on platforms in which *an arbitrary (but fixed) number of different types of resources may be available in limited quantities upon each processor*, and the partitioning algorithm must therefore be cognizant of all these limitations when determining the assignment of tasks to processors. This problem is known to be highly intractable. To get an idea of its computational complexity it is worth noting that the simpler problem of partitioning tasks onto multiprocessor platforms, where computing capacity is the only limiting resource on each processor, is NP-hard in the strong sense. Therefore, an algorithm that achieves optimal resource utilization even in the case of one limiting resource per processor, is expected to have an inefficient implementation.

3.1 System Model

The ideas presented in this chapter are able to deal with any fixed number of different types of resources. However, for the sake of simplicity we will restrict much of the discussion to systems in which there are two constraining resources – computation capacity and local memory – on each processor.

We consider a task system model in which all tasks are implicit-deadline sporadic tasks (Liu and Layland, 1973) also known as *Liu and Layland* tasks. A task system τ consists of n tasks: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and a task is denoted by τ_i (i = 1 to n). Each task τ_i is characterized by its

- *Computation requirement*, u_i. Every implicit-deadline task is characterized by a worst-case execution requirement C_i, and a minimum inter-arrival separation T_i. We denote the *utilization* u_i of a task, which by definition is C_i/T_i, as its computation requirement.
- *Memory requirement*, v_i. The memory requirement of a task is the fraction of local memory a task requires for its exclusive use. For example, local memory may be used to store the executable code of a task on a processor.

We make no assumptions about the relationship between the computation requirement u_i, and the memory requirement v_i, for a task τ_i. In particular, we do not require that tasks with modest computation requirements have small v_i, and those with large computation requirements have large v_i. (Such restrictions would not allow us to model, e.g., a relatively simple task that is extremely computation-intensive because it repeatedly samples external input at a rapid rate, or a task with large code-size, comprised of much conditional code, that is invoked very infrequently and hence does not place a large computation demand on the processor.)

The multiprocessor platform is comprised of m identical processors denoted $\pi_1, \pi_2, \ldots, \pi_m$ on which we are to partition a task system τ of n sporadic tasks. We know that preemptive EDF is optimal on uniprocessors (Liu and Layland, 1973), and that a necessary and sufficient condition for a collection of implicit-deadline sporadic tasks to be EDF-schedulable on a uniprocessor is that the

sum of the computation requirements of all the tasks on the processor not exceed the computing capacity of the processor (as stated in Theorem 2.1). A correct partitioning of τ on the m processors is therefore one that ensures that

1. the sum of the computation requirements (the u_i parameters) of all the tasks assigned to each processor does not exceed 1, and

2. the sum of the memory requirements (the v_i parameters) of all the tasks assigned to each processor does not exceed 1.

3.2 Context, and related work

A partitioning problem is inherently a decision problem, either a task system is schedulable or not. Partitioning problems are NP-hard in the strong sense, therefore an optimal partitioning algorithm is intractable. To deal with the intractability, we consider designing approximate partitioning algorithms. In order to be able to quantitatively discuss the effectiveness of different approximation algorithms it is useful to define the corresponding *optimization* problem. For partitioned scheduling one such optimization problem, the one we use in this chapter, asks: given a task system and a platform, *what is the minimum multiplicative factor by which the resources available on each processor in the platform must be augmented, in order for the task system's schedulability to be determined in polynomial-time*? The minimum multiplicative factor by which the resources need to be augmented is called the *resource augmentation bound* of the algorithm.

There is a classification of NP-hard optimization problems according to the difficulty of obtaining approximate solutions to these problems in polynomial-time. In particular, an NP-hard minimization problem is said to be in the class **APX** (for *APproXimable*) if there is a constant c such that some polynomial-time algorithm can obtain a solution to any problem instance that is no more than c times the cost of the (optimal) minimum-cost solution. APX problems for which there exists polynomial-time algorithms that can obtain a solution to any problem instance that is no more than c times the cost of the (optimal) minimum-cost solution for all $c > 1$ are said to be in the

class **PTAS** (for *Polynomial-Time Approximation Schemes*). Here, c is called the approximation bound of the algorithm. (We use the terms approximation bound and resource augmentation bound interchangeably.) It is good to be able to show that an NP-hard optimization problem is in the class APX, even better to be able to show that it is in the class PTAS.

PTAS. A problem related to the partitioning problem, described in (Hochbaum and Shmoys, 1987), that deals with the scheduling of non-preemptive jobs with the objective of *minimizing makespan* is shown to have a resource augmentation bound of $1+\varepsilon$, for any $\varepsilon > 0$. Therefore, the minimizing makespan problem is in the class PTAS. It can be shown that both the partitioning problem and minimizing makespan problem are in fact equivalent. Therefore, for a single limited resource (computing capacity), the partitioning problem for implicit-deadline sporadic task systems is in the class PTAS. In (Chattopadhyay and Baruah, 2011) we derived a lookup table based PTAS partitioning approach for a single limited resource. In this approach we do the expensive computation related to partitioning just once and store the results in a lookup table. For any subsequent partitioning the worst-case run-time complexity is associated with the run-time incurred in looking up the lookup table.

It can be shown by reduction to the vector scheduling problem (Chekuri and Khanna, 2004), that for two limited resources (computing capacity and local memory), the partitioning problem is in the class PTAS. We can use the results in (Chekuri and Khanna, 2004) to extend the partitioning algorithm we proposed in (Chattopadhyay and Baruah, 2011), and derive a PTAS partitioning algorithm for two limited resources. We evaluated this partitioning algorithm and observed that the number of entries in the lookup table were prohibitively large. As a result, the worst-case run-time complexity of the algorithm for any subsequent partitioning was also very large.

APX. Various heuristics for task partitioning on processors with limited computing capacity have been studied and evaluated. In (Lopez et al., 2004), heuristics such as *First-Fit*, *Best-Fit*, *Worst-Fit*, *First-Fit-Decreasing* etc., that have very efficient implementations have been compared on the basis of their *sufficient schedulability conditions* (for a description of these heuristics please see (Lopez

et al., 2004)). Paraphrasing and simplifying slightly, the main result from (Lopez et al., 2004) can be stated as follows: any implicit-deadline sporadic task system satisfying the condition

$$\sum_{\text{all } i} u_i \leq m - (m-1) \times \min\left(\frac{1}{2}, \max_{\text{all } i}\{u_i\}\right) \tag{3.1}$$

is successfully partitioned by the First-Fit-Decreasing (FFD) heuristic upon a platform consisting of m unit-capacity processors. In multiprocessor platforms with limited computing capacity First-Fit has an approximation bound of $(2 - \frac{1}{m})$ (Fisher, 2007, P.176). Thus, the First-Fit heuristic is in the class APX.

Two partitioning algorithms, based on constructing and approximately solving integer linear programs (ILPs), are presented in (Fisher et al., 2005). One algorithm constructs an ILP from the specification of the task system and then solves a non-integer relaxation to this ILP. This algorithm has a resource augmentation bound of 3 for processors with two limited resources, hence it is not a PTAS. However, it can be implemented far more efficiently than our partitioning algorithm based on vector scheduling. This algorithm is only applicable for partitioning task systems in which all the resource requirements of a task are below a certain threshold. The second algorithm presented in (Fisher et al., 2005) has no such restriction. If a task system consists only of tasks with computation and memory requirements < 0.5, the second algorithm simply calls the first algorithm to partition the task system. (The exact resource augmentation bound of the second algorithm is not known.) However, unlike the first algorithm, the second algorithm does not always have an efficient run time complexity. For assigning certain tasks the second algorithm needs to solve a pure ILP. Solving a pure ILP for the partitioning problem is NP-hard and the solution may not be obtained efficiently.

Heuristics such as First-Fit, Best-Fit, Worst-Fit, First-Fit-Decreasing etc. for the multidimensional bin packing problem, which is related to the partitioning problem, have been described and analyzed in (Kou and Markowsky, 1977). The analysis in (Kou and Markowsky, 1977) shows that the First-Fit algorithm has an approximation bound no greater than $d+1$ where d is an arbitrary,

but fixed, number of dimensions that an item can occupy. This approximation bound is with respect to the number of bins needed to pack the items and it is shown to be tight in (Kou and Markowsky, 1977).

The above result for the bin packing problem can be applied to the partitioning problem. Thus, First-Fit for the task partitioning problem can be shown to have an approximation bound $d+1$ with respect to the number of processors. Here d is an arbitrary, but fixed, number of limited resources available on each processor. For a partitioning algorithm the approximation bound with respect to the number of processors indicates how many additional processors may be needed to ensure the successful partitioning of a task system that can be partitioned by an optimal algorithm. The resource augmentation bound, on the other hand, indicates by how much the resources on the existing processors may need to be inflated to ensure the successful partitioning of a task system that can be partitioned by an optimal algorithm. In this chapter we compare partitioning algorithms on the basis of their resource augmentation bound.

3.3 PTAS Partitioning

In this section we describe the PTAS partitioning approach proposed in (Chattopadhyay and Baruah, 2011) for platforms in which computing capacity is the only limiting resource. (We later discuss why this approach may not be suitable for partitioning on processors with more than one limited resource.) The main idea behind our approach is to construct a *lookup table* (LUT) for any identical multiprocessor platform upon which we intend to execute implicit-deadline sporadic tasks under partitioned EDF. Whenever a task system is to be partitioned upon this platform, the table that we construct may be used to determine the assignment of tasks to the processors.

The lookup table for a particular platform is constructed without knowledge of the task systems that will later be partitioned upon the platform. We do not therefore know, during table construction time, the exact characteristics of the tasks that will be assigned to the processors of the platform. Instead, the table is constructed assuming that the utilizations of all the tasks have values from within a fixed set of distinct values V. When this lookup table is later used to actually perform partitioning

of a given task system τ, each task in τ may need to have its WCET parameter *inflated* so that the resulting task utilization is indeed one of these distinct values in V. (The *sustainability* (Baruah and Burns, 2006) property of preemptive uniprocessor EDF ensures that if the tasks with the inflated WCET's are successfully scheduled, then so are the original tasks.) The challenge is to choose the distinct utilization values in V in a clever manner, so that the amount of such inflation that is needed is bounded.

We will see that larger the number of distinct utilization values we are permitted to have in the set V, the smaller the amount of inflation that is needed. Hence, an important design decision must be made prior to table-construction time: ***How large a table will we construct?*** This is expressed in terms of choosing a value for a parameter ε for the procedure that constructs the lookup table. Informally speaking, smaller the value of ε, smaller the degree of rounding up that is needed and closer to optimal our subsequent task-assignment procedure will be. However, the size of the lookup table and the time required to do a lookup also depend on the value of ε: the smaller the value, the larger the table-size.

3.3.1 Overview: Constructing a lookup table

We now describe the construction of the lookup table for a given multiprocessor platform. Note that this table is constructed only once for a given platform. Let us suppose that the multiprocessor platform consists of m unit-speed processors. The steps involved in constructing the lookup table for this platform are as follows:

1. Choose a value for the parameter ε, which determines the degree of accuracy.

2. Based on the value chosen for ε, determine the utilization values that are to be included in the set V. (To make explicit the dependence of this set of utilization values upon the value chosen for ε, we will henceforth denote this set as $V(\varepsilon)$). Recall that during the process of actually partitioning implicit-deadline sporadic task systems upon this platform, we will round up the actual utilizations of tasks to equal one of the utilization values in $V(\varepsilon)$.

3. Determine the combinations of tasks with utilizations in $V(\varepsilon)$ that can be scheduled together on a single processor.

4. Use these single-processor combinations to determine the combinations of tasks with utilizations in $V(\varepsilon)$ that can be scheduled on m processors.

Each of these steps is discussed in greater detail, in Sections 3.3.2-3.3.5. Throughout the discussion, we will illustrate the construction of the lookup table by means of an example. Let us suppose that the platform in this running example consists of 4 unit-speed processors (i.e., $m = 4$).

3.3.2 Choosing ε

As stated in Section 3.3.1 above, the procedure for computing the lookup table must be provided a parameter ε, which is a positive real number. A design decision must now be made in the form of choosing a value for ε.

We will see later (Theorem 3.2) that the performance guarantee that is made by our partitioning algorithm is as follows: any task system that can be partitioned upon m unit-speed processors by an optimal partitioning algorithm will be partitioned by our algorithm on m processors each of speed $(1+\varepsilon)$. Hence, in choosing a value for ε we are in effect reducing the guaranteed utilization bound of each processor to equal $1/(1+\varepsilon)$ times the actual utilization; so the decision in choosing a value for ε essentially becomes: what fraction of the processor capacity are we willing to sacrifice, [1] in order to be able to do task partitioning more efficiently? For instance, if we were willing to tolerate a loss of up to 10% of processor utilization, ε would need to satisfy the condition:

$$\frac{1}{1+\varepsilon} \geq 0.9$$
$$\Leftrightarrow \quad \varepsilon \leq \frac{1}{0.9} - 1$$

[1] We note that this "sacrifice" is only in terms of *worst-case guarantees*, as formalized in Theorem 3.2. It is quite possible that some of this sacrificed capacity can in fact be used during the partitioning of particular task systems.

$$\Leftrightarrow \quad \varepsilon \leq \frac{1}{9}.$$

As stated in Section 3.3.1, the size of the lookup table, and the time required to do a lookup, also depend on the value of ε: smaller the value, larger the table-size. Hence, ε is assigned the largest value consistent with the desired overall system utilization. In the example above, ε would in fact be assigned the value $1/9$.

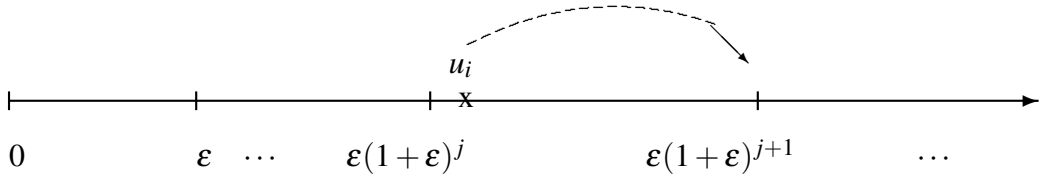
Example 3.1. For our running example, let us choose the value 0.3 for the parameter ε. (For an actual platform we would typically choose a far smaller value, but this larger value is more useful for purposes of illustration here: for small values of ε, the sizes of the intermediate data structures are too large to be illustrative from a pedantic perspective.) With $\varepsilon \leftarrow 0.3$, we are guaranteeing to achieve the same performance as an optimal algorithm would, on a platform consisting of the same number of processors each of speed equal to $1/(1+0.3)$, or ≈ 0.77, of the speeds of the processors available to our algorithm that is, we are "sacrificing," in the worst case, a bit less than a quarter of the platform's computing capacity. (However, recall the point made in footnote 1, concerning the pessimism in this worst-case bound on the fraction of capacity that is sacrificed. This point is illustrated for our running example in Example 3.5, where the lookup table that we eventually construct is used to successfully partition a task system with utilization ≈ 3.6, in excess of the upper bound of $4 \cdot (1/1.3) \approx 3.077$.) $\diamond$

3.3.3 Determining utilization values

Once we have settled on a value for ε we use this value to determine which utilization values to include in the set $V(\varepsilon)$ of distinct utilization values that will be represented in the lookup table we construct. In choosing the members of $V(\varepsilon)$, the objective is to minimize the amount by which the utilizations of the tasks to be partitioned must be *inflated* in order to become equal to one of the values in $V(\varepsilon)$.

The choice we make is to have $V(\varepsilon)$ be equal to the set of all real numbers of the form $\varepsilon \cdot (1+\varepsilon)^k$, for all non-negative integers k (up to the upper limit of one). Why are these particular

values chosen? Recall that when the table is used to perform task partitioning, the actual task utilizations (which may take on any value) will be rounded up to the nearest value present in the set $V(\varepsilon)$. Suppose that an actual utilization u_i is just a bit greater than one of the values present in V, say, $\varepsilon(1+\varepsilon)^j$ — this is depicted in the figure below by an "x.".



This utilization will be rounded up to $\varepsilon(1+\varepsilon)^{j+1}$. The fraction by which this utilization has been inflated is therefore

$$\frac{\varepsilon(1+\varepsilon)^{j+1}}{u_i} \;<\; \frac{\varepsilon(1+\varepsilon)^{j+1}}{\varepsilon(1+\varepsilon)^j} \;=\; (1+\varepsilon)\,.$$

Thus if each task's utilization were to be inflated by this maximal factor, it follows that any collection of tasks with total utilization $\leq 1/(1+\varepsilon)$ would have inflated utilization ≤ 1, and would hence be determined, based on our lookup table, to fit on a single processor[2].

Let us now determine $|V(\varepsilon)|$, the number of elements in the set $V(\varepsilon)$. We wish to include each positive real number ≤ 1 that is of the form $\varepsilon(1+\varepsilon)^j$ for non-negative j. Since

$$\begin{aligned}
&\varepsilon(1+\varepsilon)^j \leq 1 \\
&\quad\Leftrightarrow\quad (1+\varepsilon)^j \leq (1/\varepsilon) \\
&\quad\Leftrightarrow\quad j\log(1+\varepsilon) \leq \log(1/\varepsilon) \\
&\quad\Leftrightarrow\quad j \leq \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)}\,,
\end{aligned}$$

[2]Note that this argument does not hold for actual utilizations — the u_i in the figure — less than $\varepsilon/(1+\varepsilon)$. If a task with utilization u_i arbitrarily close to zero ($u_i \to 0^+$) were to have its utilization rounded up to $\varepsilon/(1+\varepsilon)$, the inflation factor would be $(\varepsilon/(1+\varepsilon)) \div u_i$, which approaches ∞ as $u_i \to 0$. We will see that the task-assignment procedure of Section 3.3.6 handles tasks with utilization $< \varepsilon/(1+\varepsilon)$ differently.

we conclude that

$$|V(\varepsilon)| = \left\lfloor \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)} \right\rfloor + 1. \tag{3.2}$$

Example 3.2. For our example ($\varepsilon = 0.3$), it may be verified that $\frac{\log(1/\varepsilon)}{\log(1+\varepsilon)} = \frac{\log(1/0.3)}{\log 1.3} \approx 4.589$. Hence by Equation 3.2 there are $(\lfloor 4.589 \rfloor + 1) = 5$ elements in $V(0.3)$. We therefore have 5 distinct utilization values to consider: for $j = 0, 1, 2, 3$, and 4. (The value of 1.114 for $j = 5$ is too large, as are the values for all $j > 5$.) These elements are computed as follows:

j	**util. value** $(0.3 \cdot (1.3)^j)$
0	0.3000
1	0.3900
2	0.5070
3	0.6591
4	0.8568
5	1.114

Hence

$$V(0.3) = \left\{0.3000, 0.3900, 0.5070, 0.6591, 0.8568\right\}.$$

◇

3.3.4 Determining legal single-processor configurations

In Section 3.3.3 above, we have determined the distinct utilization values in the set $V(\varepsilon)$. We now seek to determine all the different ways in which a single processor can be packed with tasks of these utilizations. We refer to these as *single-processor configurations*.

For reasons of efficiency in storage (and subsequent lookup), we seek only the maximal configurations of this kind: a single-processor configuration is said to be a *maximal* one if no additional task (also with utilization $\in V(\varepsilon)$) can be added without the sum of the utilizations exceeding the capacity of the processor:

Definition 3.1. Single-proc. configuration: For a given value of ε, a single-processor configuration is a $|V(\varepsilon)|$-tuple

$$\langle x_1, x_2, \ldots, x_{|V(\varepsilon)|} \rangle$$

of non-negative integers, satisfying the constraint that

$$\Big(\sum_{i=1}^{|V(\varepsilon)|} (x_i \cdot \varepsilon \cdot (1+\varepsilon)^{i-1}) \Big) \leq 1. \tag{3.3}$$

The single-processor configuration $\langle x_1, x_2, \ldots, x_{|V(\varepsilon)|} \rangle$ is ***maximal*** if

$$\Big(\sum_{i=1}^{|V(\varepsilon)|} (x_i \cdot \varepsilon \cdot (1+\varepsilon)^{i-1}) \Big) > (1-\varepsilon) \tag{3.4}$$

(thereby implying that no task can be added to this single-processor configuration without exceeding the processor's capacity). □

Our objective is to determine a list $L_1(\varepsilon)$ of all possible maximal single-processor configurations for the selected value of ε (here, the subscript "1" denotes the number of processors. In section 3.3.5 below, we will describe how we construct the list $L_m(\varepsilon)$ for m processors).

Since there are only finitely many distinct utilization values in $V(\varepsilon)$ (the exact number is as determined by Equation 3.2), all the elements of $L_1(\varepsilon)$ can in principle be determined by exhaustive enumeration; simply try all $|V(\varepsilon)|$-tuples with the i'th component no larger than $\big(1/(\varepsilon \cdot (1+\varepsilon)^{i-1})\big)$, adding the ones that satisfy Inequalities 3.3 and 3.4 to $L_1(\varepsilon)$. Such a procedure has run-time exponential in $(1/\varepsilon)$: the smaller the value of ε, the greater the run-time of the procedure (and the length of $L_1(\varepsilon)$ — the number of maximal single-processor configurations found). Although this can be quite high for small ε, we point out that:

- This run-time is incurred only once. After the list $L_1(\varepsilon)$ has been constructed, it can be stored and repeatedly reused for doing task partitioning.

- Although the size of $L_1(\varepsilon)$ is indeed exponential in $(1/\varepsilon)$ (to be specific $|L_1(\varepsilon)| = \mathscr{O}(|V(\varepsilon)|^{\frac{1}{\varepsilon}})$), our experiments reveal that this size is quite reasonable in practice for values of ε that are

Config. ID	0.3000	0.3900	0.5070	0.6591	0.8568
1	3	0	0	0	0
2	2	1	0	0	0
3	1	0	1	0	0
4	1	0	0	1	0
5	0	2	0	0	0
6	0	1	1	0	0
7	0	0	0	0	1

Table 3.1: All the maximal single-processor configurations for the example.

not too small. We have computed these lists for various values of ε. For instance, choosing $\varepsilon = \frac{1}{9}$ (which is equivalent to sacrificing at most 10% of each processor's capacity) yields a list of 9604 maximal single-processor configurations. Given current memory costs, look-up tables of sizes far larger than this are quite viable — consider the lookup tables used in, e.g., floating-point co-processors for speeding up the computation of operations such as `sin`, `cos`, `log`, etc., which are often tens of megabytes large.

- Several simple and straightforward counting and programming techniques can be used to optimize the computation of the list $L_1(\varepsilon)$. For example, we use a dynamic programming approach in our computation of the list $L_1(\varepsilon)$. We start with a maximal configuration that consists of the maximum number of only the first component in the set $V(\varepsilon)$, which is ε. (An example of such a configuration is the single-processor configuration with ID. 1 in Table 3.1 with $\varepsilon = 0.3$). We then systematically compute the next configuration by decreasing the value of the first component, and maximizing the value of the next component until we have reached the last component. We repeat the above for every component in the set $V(\varepsilon)$. A configuration is added to the list $L_1(\varepsilon)$ if it satisfies Inequalities 3.3 and 3.4.

Example 3.3. For our running example with $\varepsilon = 0.3$, it turns out that there are just seven maximal single-processor configurations. These maximal single-processor configurations are shown in Table 3.1. The numbers in the headings for columns 2-6 are the 5 distinct utilization values in $V(0.3)$, that we determined in Section 3.3.3 above. Each row corresponds to a different maximal

single-processor configuration. It may be verified that the sum of the utilizations in each configuration (i) satisfies Inequality 3.3 (is no larger than 1.0), and (ii) satisfies Inequality 3.4 (is at least 0.7, i.e., adding a task with even the smallest utilization would exceed the processor's capacity). Consider, for example, the single-processor configuration with ID. 2, the sum of the utilizations is $2 \cdot 0.3000 + 1 \cdot 0.3900$, or 0.9900. For the single-processor configuration with ID. 4, the sum of the utilizations is $1 \cdot 0.3000 + 1 \cdot 0.6591$, or 0.9591. $\diamond$

3.3.5 Determining legal multi-processor configurations

We can use the maximal single-processor configurations determined above to determine maximal configurations for a collection of m processors. Intuitively, each such maximal multiprocessor configuration will represent a different manner in which m processors can be maximally packed with tasks having utilizations in $V(\varepsilon)$.

Definition 3.2. Multiprocessor configuration: For given m and ε, a multiprocessor configuration is an ordered pair of a $|V(\varepsilon)|$-tuple

$$\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|} \rangle$$

of non-negative integers, and an m-tuple

$$\langle z_1, z_2, \ldots, z_m \rangle$$

of positive integers $\leq |L_1(\varepsilon)|$. The z_j's denote configuration ID's of single-processor configurations (as previously computed, and stored in $L_1(\varepsilon)$); $\langle z_1, z_2, \ldots, z_m \rangle$ thus denotes the m-processor configuration obtained by configuring the j'th processor according to the single-processor configuration represented by ID z_j in $L_1(\varepsilon)$, for $1 \leq j \leq m$.

The tuples $\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|} \rangle$ and $\langle z_1, z_2, \ldots, z_m \rangle$ must satisfy the constraint that for each i, $1 \leq i \leq |V(\varepsilon)|$, the i'th component of the tuples in $L_1(\varepsilon)$ with ID's $\in \langle z_1, z_2, \ldots, z_m \rangle$ sums to exactly y_i.

A multiprocessor configuration $\left(\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|}\rangle, \langle z_1, z_2, \ldots, z_m\rangle\right)$ is ***maximal*** if there is no other multiprocessor configuration $\left(\langle y'_1, y'_2, \ldots, y'_{|V(\varepsilon)|}\rangle, \langle z'_1, z'_2, \ldots, z'_m\rangle\right)$ such that $y'_i \geq y_i$ for all i, $1 \leq i \leq |V(\varepsilon)|$. □

Let $L_m(\varepsilon)$ denote the list of all maximal multiprocessor configurations. $L_m(\varepsilon)$ can in principle be determined using exhaustive enumeration; simply consider all m-combinations of the single-processor configurations computed and stored in $L_1(\varepsilon)$. While the worst-case run-time could be as large as $|L_1(\varepsilon)|^m$ and thus once again exponential in ε and m, this step, like the computation of $L_1(\varepsilon)$, also needs to be performed only once for a given multiprocessor platform. As was the case with computing $L_1(\varepsilon)$, all manner of counting techniques and programming heuristics may be employed to reduce the run-time in practice. We list a few such optimizations below.

- One obvious such heuristic that can speed up the computation of $L_m(\varepsilon)$ quite significantly is to iteratively compute $L_j(\varepsilon)$ from $L_{j-1}(\varepsilon)$ and $L_1(\varepsilon)$, for $j = 2, 3, \ldots, m$. In such an iterative approach, only the maximal multiprocessor configurations are retained in each intermediate $L_j(\varepsilon)$. Since the number of maximal multiprocessor configurations in $L_j(\varepsilon)$ is typically far smaller than the worst-case bound of $|L_1(\varepsilon)|^j$, such an iterative procedure is observed to be far more efficient than determining $L_m(\varepsilon)$ directly from $L_1(\varepsilon)$ by considering all m-combinations of maximal single-processor combinations.

- For large values of m, further savings in run-time can be achieved by only determining $L_j(\varepsilon)$ for values of j that are exact powers of two and $\leq m$ (i.e. for $j = 1, 2, 4, \ldots, 2^{\lfloor \log m \rfloor}$). Each such $L_j(\varepsilon)$ can be determined by considering 2-combinations of configurations in $L_{j/2}(\varepsilon)$. Once these have all been determined, $L_m(\varepsilon)$ can be determined by considering the combinations of the $L_j(\varepsilon)$'s corresponding to the j's that are powers of two summing to m. It is evident that there are at most $\log m$ such j's.

After it has been computed, $L_m(\varepsilon)$ is stored in a lookup table that is provided along with the m-processor platform, and is used (in a manner discussed in Section 3.3.6 below) for partitioning specific task systems upon the platform.

0.3	0.39	0.507	0.6591	0.8568	**Single-proc. ID's**
3	2	1	2	0	[4 4 5 3]
3	4	2	0	0	[6 6 5 1]
0	3	3	0	1	[6 6 6 7]
4	1	1	1	1	[7 4 3 2]
4	0	1	3	0	[4 4 4 3]

Table 3.2: Some example maximal 4-processor configurations.

Example 3.4. For our example 4-processor platform with $\varepsilon = 0.3$, it turns out that there are 140 maximal multiprocessor configurations. Although this is too many to enumerate in this document, we depict a few maximal multiprocessor configurations in Table 3.2 in the format that they will appear in the lookup table. The numbers in the headings for columns 1-5 are the 5 distinct utilizations; the sixth column lists the 4 maximal single-processor configurations (named according to the configuration ID's of Table 3.1) that give rise to this particular maximal multiprocessor configuration.

$\diamond$

3.3.6 Task assignment

The lookup table of maximal multiprocessor configurations needs to be determined once. Once this lookup table has been obtained, we can use it repeatedly to determine whether any implicit-deadline sporadic task system can be partitioned on this platform. We now describe the partitioning algorithm for doing so.

Let τ denote a collection of n implicit-deadline sporadic tasks to be partitioned among the (unit-capacity) processors in the m-processor platform. Let u_i denote the utilization of the i'th task in τ. (The task system τ, to be partitioned is completely specified by specifying the utilizations of the n tasks in it.) Our task assignment algorithm is depicted in Figure 3.1. It operates in two phases.

1. In the first phase (Steps 1 and 2 in the pseudo-code), it attempts to assign all tasks with utilization $\geq \varepsilon/(1+\varepsilon)$. The lookup table constructed as described in Section 3.3.1 is used during this phase.

2. Once this phase has been completed, tasks with utilization $< \varepsilon/(1+\varepsilon)$ are considered during the second phase (Steps 3 and 4 in the pseudo-code). In essence, the algorithm attempts to accommodate these small-utilization tasks in the remaining capacity that is left over in the individual processors after phase 1 is completed.

We will first show that the partitioning algorithm is ***sound***.

Theorem 3.1. *If the partitioning algorithm of Figure 3.1 succeeds in assigning all the tasks in τ, then the tasks that are assigned to each processor can be scheduled on that processor to meet all deadlines by uniprocessor EDF.*

Proof. During the first phase (Steps 1 and 2 in the pseudo-code), the algorithm assigns tasks to processors such that the sum of the inflated utilizations of all the tasks on each processor does not exceed the capacity of the processor. Hence, the sum of the original (i.e., non-inflated) utilizations of tasks assigned to any particular processor does not exceed the capacity of the processor. This property is preserved during the second phase of the algorithm (Steps 3 and 4 in the pseudo-code), since a task is only added to a processor during this phase if the sum of the utilizations after doing so will not exceed the processor's capacity. Hence if the task-assignment algorithm succeeds in assigning all the tasks to processors, then the sum of the utilizations of the tasks assigned to any particular processor is no larger than one. It follows from the optimality of EDF on uniprocessor platforms (Liu and Layland, 1973; Dertouzos, 1974) that each processor is consequently successfully scheduled by EDF. □

And what if the algorithm *fails* to assign all the tasks in τ to the processors? In that case, we will now show that no algorithm, not even an optimal one, could have partitioned τ upon an m-processor platform comprised of processors of slightly smaller computing capacity:

Task system τ, consisting of n implicit-deadline tasks with utilizations $u_1, u_2, \ldots, u_n$, is to be partitioned among m unit-speed processors.

1. For each task with utilization $\geq \varepsilon/(1+\varepsilon)$, *round up* its utilization (if necessary) so that it is equal to $\varepsilon \times (1+\varepsilon)^k$ for some non-negative integer k. (Observe that such rounding up inflates the utilization of a task by at most a factor $(1+\varepsilon)$: the ratio of the rounded-up utilization to the original utilization of any task is $\leq (1+\varepsilon)$.)

   Now all the tasks with (original) utilization $\geq \varepsilon/(1+\varepsilon)$ have their utilizations equal to one of the distinct values that were considered during the table-generation step. Let k_i denote the number of tasks with modified utilization equal to $\varepsilon \times (1+\varepsilon)^{i-1}$, for each i, $1 \leq i \leq |V(\varepsilon)|$.

2. Determine whether this collection of modified-utilization tasks can be accommodated in one of the maximal m-processor configurations that had been identified during the pre-processing phase. That is, determine whether there is a maximal multiprocessor configuration

   $$\Big(\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|}\rangle, \langle z_1, z_2, \ldots, z_m\rangle\Big)$$

   in $L_m(\varepsilon)$, satisfying the condition that $y_i \geq k_i$ for each i, $1 \leq i \leq |V(\varepsilon)|$.

   - If the answer here is "no," then report ***failure***, we are unable to partition τ among the m processors.
   - If the answer is "yes," however, then ***a viable partitioning has been found*** for the tasks with (original) utilization $\geq \varepsilon/(1+\varepsilon)$. Assign these tasks according to the maximal m-processor configuration.

3. It remains to assign the tasks with utilization $< \varepsilon/(1+\varepsilon)$. Assign each such task to any processor upon which it will "fit" i.e., any processor on which the sum of the (original — i.e., unmodified) utilizations of the tasks assigned to the processor would not exceed one if this task were assigned to that processor.

4. If all the tasks with utilization $< \varepsilon/(1+\varepsilon)$ are assigned to processors in this manner, then ***a viable partitioning has been found*** for all the tasks. However, if some task cannot be assigned in this manner, then report ***failure***, we are unable to partition τ among the m processors.

Figure 3.1: Outline of Algorithm PTAS-PARTITION

Theorem 3.2. *If the partitioning algorithm of Figure 3.1 fails to partition the tasks in τ, then no algorithm can partition τ on a platform of m processors each of computing capacity $1/(1+\varepsilon)$.*

Proof. The partitioning algorithm of Figure 3.1 may declare failure at two points, one of which is in phase one and the other is in phase two. We consider each possible point of failure separately.

1. Suppose that the algorithm reports failure during phase one while attempting to assign only the tasks with utilization $\geq \varepsilon/(1+\varepsilon)$ (Step 2 in the pseudo-code). Since each such task has its utilization inflated by a factor $\leq (1+\varepsilon)$, it must be the case that all such (original — i.e., unmodified-utilization) tasks cannot be scheduled by an optimal algorithm on a platform comprised of m processors each of computing capacity $1/(1+\varepsilon)$. In other words, even just the tasks in τ with unmodified utilizations $\geq \varepsilon/(1+\varepsilon)$ cannot be partitioned among m processors of computing capacity $1/(1+\varepsilon)$ each, and consequently all of τ clearly cannot be partitioned on such a platform.

2. Suppose that the algorithm reports failure during phase two, while attempting to assign the tasks with utilization $< \varepsilon/(1+\varepsilon)$ (Step 4 in the pseudo-code). This would imply that while some task with utilization $< \varepsilon/(1+\varepsilon)$ remains unallocated, the sum of the utilizations of the tasks already assigned to each processor is $> (1-\varepsilon/(1+\varepsilon))$. Therefore the total utilization of τ exceeds $m \times (1-\varepsilon/(1+\varepsilon)) = m(1/(1+\varepsilon))$, and τ cannot consequently be feasible on m processors of computing capacity $(1/(1+\varepsilon))$ each.

The theorem follows. □

Example 3.5. Returning to our example ($m = 4$ processors, $\varepsilon = 0.3$), let us consider a task system τ comprised of tasks with the following utilizations (listed here in non-decreasing order):

$$\frac{1}{5}, \frac{1}{5}, \frac{1}{3}, \frac{7}{20}, \frac{9}{25}, \frac{2}{5}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}.$$

Noting that $\varepsilon/(1+\varepsilon) = 0.3/1.3 \approx 0.2308$, we observe that the first two tasks have utilization $< \varepsilon/(1+\varepsilon)$ and are hence not to be considered during the first steps of the partitioning algorithm.

We round up the remaining utilizations:

$$\frac{1}{5}, \frac{1}{5}, 0.3900, 0.3900, 0.3900, 0.5070, 0.5070, 0.5070, 0.8568.$$

Using Table 3.2, we notice that the rounded-up utilizations here correspond to the configuration listed on the third row: 0 3 3 0 1, obtained from the single-processor configurations [6 6 6 7] of Table 3.1.

Accordingly, we assign the tasks with utilization $\geq \varepsilon/(1+\varepsilon)$ to the 4 processors as specified in configurations 6, 6, 6, and 7 respectively:

6: 1/3, 2/5. (Remaining capacity $= 1 - 0.7333 = 0.2667$)

6: 7/20, 1/2. (Remaining capacity $= 1 - 0.85 = 0.15$)

6: 9/25, 1/2. (Remaining capacity $= 1 - 0.86 = 0.14$)

7: 3/4. (Remaining capacity $= 1 - 0.7500 = 0.25$)

It remains to assign the two tasks with utilization $< \varepsilon/(1+\varepsilon)$: the ones with utilization $1/5$ each. These tasks can be accommodated in the first and last processors yielding the following mapping:

6: 1/3, 2/5, 1/5. (Remaining capacity = 0.0667)

6: 7/20, 1/2. (Remaining capacity = 0.15)

6: 9/25, 1/2. (Remaining capacity = 0.14)

7: 3/4, 1.5. (Remaining capacity = 0.05)

$\diamond$

3.3.7 Run-time complexity

The run-time complexity of Algorithm PTAS-PARTITION is dominated by Step 2, in Figure 3.1. In this step the algorithm performs a lookup in the lookup table to determine if there is a maximal

multiprocessor configuration that can be used for partitioning tasks with utilization greater than $\varepsilon/(1+\varepsilon)$. The time to lookup the lookup table depends upon how the lookup table is implemented, however in the worst-case it is $\mathcal{O}(|L_m(\varepsilon)|)$, where $L_m(\varepsilon)$ is the set of all the maximal multiprocessor configurations stored in the lookup table. (We have shown how to compute the maximal multiprocessor configurations $L_m(\varepsilon)$ in Section 3.3.5.)

For a single limited resource the number of multiprocessor configurations is exponential with respect to m and ε, which are constant parameters with respect to our problem. From Example 3.4, we know that for $m = 4$, $\varepsilon = 0.3$, there are $L_4(0.3) = 140$ maximal multiprocessors configurations. For a smaller value, $\varepsilon = 0.2$, there are $L_4(0.2) = 12980$ maximal multiprocessor configurations. Note that the primary cause of this difference in the number of maximal multiprocessor configurations is due to the difference in the number of elements in the set $V(\varepsilon)$ as computed in Section 3.3.3. From Example 3.2, we know that for $\varepsilon = 0.3$, $|V(0.3)| = 5$, whereas when $\varepsilon = 0.2$, $|V(0.2)| = 9$; the number of elements differ by 4.

For more than one limited resource it can be shown by a reduction to the vector scheduling problem that the number of multiprocessor configurations in the lookup table is exponential with respect to m, ε, and d, where d is an arbitrary, but fixed, number of limited resources on the multiprocessor platform. For $d > 1$, the elements in the set $V(\varepsilon)$ is a function of ε and d; we denote this as $V(\varepsilon, d)$. It can be shown that the value of $|V(\varepsilon, d)|$ is strictly greater than $|V(\varepsilon)|^d$, where $V(\varepsilon)$ is as computed in Section 3.3.3. This results in an "explosion" in the size of the lookup table when $d > 1$, which in turn significantly impacts the run-time complexity of looking up the lookup table.

In the following section we present the APX partitioning algorithm described in (Chattopadhyay and Baruah, 2012). This algorithm has a larger (less desirable) resource augmentation bound when compared to the resource augmentation bound of Algorithm PTAS-PARTITION. Thus with respect to resource augmentation bound, Algorithm PTAS-PARTITION dominates. However, the algorithm described in the following section has much better run-time complexity, and can be easily extended to incorporate multiple resource constraints.

3.4 APX Partitioning

Our APX partitioning algorithm (Chattopadhyay and Baruah, 2012), is a generalization of the work presented in (Lopez et al., 2004), in the sense that instead of assuming that computing capacity is the only limited resource, we assume that an arbitrary, but fixed, number of resources are available in limited quantities upon each processor. We first present our algorithm assuming that two resources, computing capacity and local memory, are limited. In Section 3.4.6, we show how our algorithm can be extended to incorporate an arbitrary, but fixed, number of limited resources on each processor.

3.4.1 Partitioning algorithm

Given a task system τ of n implicit-deadline sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$ we want to partition the tasks onto m identical processors $\pi_1, \pi_2, \ldots, \pi_m$ that have unit-capacity and unit-memory. We present an approximate, but efficient algorithm to solve the problem. Figure 3.2 gives a pseudo-code representation of our algorithm. The algorithm assumes that the collection of tasks $\tau_1, \tau_2, \ldots, \tau_n$ is in any given order and then attempts to assign the tasks onto one of the m processors. We now explain how a task τ_i is assigned to a processor.

First, let us suppose that tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$ have been successfully assigned. For any processor π_k, let $\tau(\pi_k)$ denote the tasks among $\tau_1, \tau_2, \ldots, \tau_{i-1}$ that have already been assigned to it. Task τ_i is assigned to a processor π_k only if the following two conditions are satisfied:

$$\left(1 - \sum_{\tau_j \in \tau(\pi_k)} u_j\right) \geq u_i \text{ (Condition 2)} \tag{3.5}$$

and

$$\left(1 - \sum_{\tau_j \in \tau(\pi_k)} v_j\right) \geq v_i \text{ (Condition 3)} \ . \tag{3.6}$$

```
APX-PARTITION(τ, m)
    ▷ The collection of sporadic tasks τ = {τ_1, ..., τ_n} is to be partitioned on
      m identical, unit-capacity and unit-memory processors denoted π_1, ..., π_m.
      τ(π_k) denotes the tasks assigned to processor π_k; initially, τ(π_k) ← ∅ for all
      k.
1   for i ← 1 to n
    ▷ i ranges over the tasks
2           for k ← 1 to m
            ▷ k ranges over the processors
3                   if τ_i satisfies Conditions 3.5-3.6
                    on processor π_k then
                            ▷ assign τ_i to π_k;
4                           τ(π_k) ← τ(π_k) ⋃ {τ_i}
5                           break;
6           end (of inner for loop)
7           if (k > m) return PARTITIONING FAILED
8   end (of outer for loop)
9   return PARTITIONING SUCCEEDED
```

Figure 3.2: Pseudo-code for Algorithm APX-PARTITION.

If no such π_k exists, then the algorithm declares failure: it is unable to partition τ upon the m-processor platform.

The following lemma asserts that, in assigning a task τ_i to a processor π_k our partitioning algorithm does not adversely affect the schedulability of the tasks previously assigned to the processors.

Lemma 3.1. *If the tasks previously assigned to each processor were EDF-schedulable on that processor and our algorithm assigns task τ_i to processor π_k, then the tasks assigned to each processor (including processor π_k) remain EDF-schedulable on that processor.*

Proof. Observe that the EDF-schedulability of the processors other than processor π_k is not affected by the assignment of task τ_i to processor π_k. It remains to demonstrate that, if the tasks assigned to

π_k were EDF-schedulable on π_k prior to the assignment of τ_i and Conditions 3.5-3.6 are satisfied, then the tasks on π_k remain EDF-schedulable after adding τ_i. To see that this is true, observe that

- Condition 3.5 ensures that there is sufficient computing capacity to accommodate task τ_i on processor π_k, and
- Condition 3.6 ensures that there is sufficient local memory to accommodate task τ_i on processor π_k.

□

The correctness of the partitioning algorithm can now be established by repeated applications of Lemma 3.1.

Theorem 3.3. *If our partitioning algorithm returns* PARTITIONING SUCCEEDED *on task system* τ*, then the resulting partitioning is EDF-schedulable.*

Proof. Observe that the algorithm returns PARTITIONING SUCCEEDED if and only if it has successfully assigned each task in τ to some processor.

Prior to the assignment of task τ_1, each processor is trivially EDF-schedulable. It follows from Lemma 3.1 that all processors remain EDF-schedulable after each task assignment as well. Hence, all processors are EDF-schedulable once all tasks in τ have been assigned.

□

3.4.2 Run-time complexity

Algorithm APX-PARTITION can maintain, for each processor, the cumulative computation and memory requirements of all the tasks that have been assigned to each processor thus far. For each task τ_i and each processor π_k, Conditions 3.5 and 3.6 can then be evaluated in constant time. Therefore the i'th task can be assigned in $\mathcal{O}(m)$ time. For n tasks this yields an overall run-time of $\mathcal{O}(n \times m)$, which is linear in the product of the number of tasks and the number of processors.

3.4.3 Resource augmentation bound

Algorithm APX-PARTITION is an approximation algorithm that seeks to solve the partitioning problem in polynomial time. In order to quantitatively discuss the effectiveness of Algorithm APX-PARTITION we derive a *sufficient* schedulability condition for Algorithm APX-PARTITION in Theorem 3.4 below, and use this schedulability condition to derive the resource augmentation bound of Algorithm APX-PARTITION in Theorem 3.5.

In the context of the two resource partitioning problems the resource augmentation bound can be conceptualized as follows: if an optimal algorithm can schedule a task system onto m processors then an approximation algorithm is guaranteed to schedule the task system onto m processors if the resources are inflated by a factor equal to the resource augmentation bound. Different approximation algorithms can be compared on the basis of their resource augmentation bounds. The smaller the resource augmentation bound the better -closer to optimal- the algorithm.

We would like to stress that *the properties described in Theorems 3.4–3.5 are not intended to be used as a schedulability tests to determine whether Algorithm* APX-PARTITION *would successfully schedule a given sporadic task system* – since the algorithm itself runs efficiently in polynomial time the "best" (i.e., most accurate) polynomial-time sufficient schedulability test for determining whether a particular task system is successfully scheduled by it is to actually run Algorithm APX-PARTITION and check whether it returns PARTITIONING SUCCEEDED. Rather, these properties are intended to provide a quantitative measure of how effective Algorithm APX-PARTITION is *vis a vis* the performance of an optimal scheduler.

First some definitions: for a given task system $\tau = \{\tau_1, \ldots, \tau_n\}$, let us define the following notation:

$$u_{\max}(\tau) \stackrel{\text{def}}{=} \max_{i=1}^{n} (u_i) \tag{3.7}$$

$$u_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{j=1}^{n} u_j \tag{3.8}$$

$$v_{\max}(\tau) \stackrel{\text{def}}{=} \max_{i=1}^{n} (v_i) \tag{3.9}$$

$$v_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{j=1}^{n} v_j. \tag{3.10}$$

Intuitively, $u_{\max}(\tau)$ represents the maximum computation requirement of any *individual* task, and $u_{\text{sum}}(\tau)$ represents the total computation requirement of all the tasks in the task system. Similarly, $v_{\max}(\tau)$ represents the maximum memory requirement of any individual task, and $v_{\text{sum}}(\tau)$ represents the total memory requirement of all the tasks in the task system.

Lemma 3.2 follows immediately.

Lemma 3.2. *If task system τ is feasible (under either the partitioned or the global scheduling paradigm) on an identical multiprocessor platform consisting of m processors each of computing capacity ξ and available memory ξ, it must be the case that*

$$\xi \geq \max(u_{max}(\tau), v_{max}(\tau)) \ ,$$

and

$$m \cdot \xi \geq \max(u_{sum}(\tau), v_{sum}(\tau)) \ .$$

Proof. Observe that

1. No individual task's computation requirement may exceed the computing capacity of a processor, i.e., it must be the case that $u_i \leq \xi$.

2. No individual task's memory requirement may exceed the amount of memory available on each processor, i.e., it must be the case that $v_i \leq \xi$.

Taken over all tasks in τ, these observations together yield the first condition.

In the second condition, the requirement that $m \cdot \xi \geq u_{\text{sum}}(\tau)$ simply reflects the requirement that the cumulative computation requirement of all the tasks in τ not exceed the computing capacity of the platform. Similarly, the requirement that $m \cdot \xi \geq v_{\text{sum}}(\tau)$ reflects the requirement that the total memory required by all the tasks in τ not exceed the memory available on the platform. □

Lemma 3.2 above specifies necessary conditions for our partitioning algorithm to successfully partition a sporadic task system; Theorem 3.4 below specifies a *sufficient* condition. But first, a technical lemma that will be used in the proof of Theorem 3.4.

Lemma 3.3. *Suppose that Algorithm* APX-PARTITION *is attempting to schedule task system τ on a platform consisting of unit-capacity and unit-memory processors.*

A: *If $u_{sum}(\tau) \leq 1$, then Condition 3.5 is always satisfied.*

B: *If $v_{sum}(\tau) \leq 1$, then Condition 3.6 is always satisfied.*

Proof. The proof of A is straightforward, since violating Condition 3.5 requires that $(u_i + \sum_{\tau_j \in \tau(\pi_k)} u_j)$ exceed 1. Similarly, the proof of B follows from the observation that violating Condition 3.6 requires that $(v_i + \sum_{\tau_j \in \tau(\pi_k)} v_j)$ exceed 1.

□

Thus, any implicit-deadline sporadic task system satisfying all of $u_{\text{sum}}(\tau) \leq 1$ and $v_{\text{sum}}(\tau) \leq 1$ is successfully scheduled by our algorithm. We will describe, in Theorem 3.4, what happens when one or more of these conditions are not satisfied; Lemmas 3.4-3.5 below derive technical results that are used in proving Theorem 3.4.

Lemma 3.4. *Suppose $u_{sum}(\tau) > 1$. Let the tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$, be successfully mapped by the partitioning algorithm onto the available processors. When the partitioning algorithm is attempting to assign τ_i, if Condition 3.5 fails on m_1 processors then the following inequality must hold:*

$$m_1 < \frac{u_{sum}(\tau) - u_i}{1 - u_i}. \tag{3.11}$$

Proof. Since none of the m_1 processors satisfy Condition 3.5 for task τ_i, it must be the case that there is not enough remaining computing capacity on each such processor to accommodate the computation requirement of task τ_i. Therefore, strictly more than $(1 - u_i)$ of the computing capacity of each such processor has been consumed by the tasks already assigned to these processors.

Summing over all m_1 processors and noting that the tasks already assigned $(\tau_1, \tau_2, \ldots, \tau_{i-1})$ to these processors is a subset of the tasks in τ, we obtain the following:

$$\begin{aligned}
& (1-u_i)m_1 < \sum_{j=1}^{i-1} u_j \\
\Rightarrow \quad & (1-u_i)m_1 + u_i < \sum_{j=1}^{i} u_j \\
\equiv \quad & (1-u_i)m_1 + u_i < \sum_{j=1}^{n} u_j \ (\textstyle\sum_{j=1}^{i} u_j \leq \sum_{j=1}^{n} u_j) \\
\equiv \quad & m_1 < \frac{u_{\text{sum}}(\tau) - u_i}{1-u_i} \ (\textstyle\sum_{j=1}^{n} u_j = u_{\text{sum}}(\tau)) \ ,
\end{aligned}$$

which is as asserted by the lemma.

Note that according to Lemma 3.3 (Part A), if $u_{\text{sum}}(\tau) \leq 1$ then Condition 3.5 is always satisfied. In this lemma we consider that Condition 3.5 fails on m_1 processors when attempting to map task τ_i. Therefore, by Lemma 3.3 (Part A), the value of $u_{\text{sum}}(\tau)$ should be greater than 1, which is as stated in the Lemma. For $u_{\text{sum}}(\tau) > 1$, if $u_i = u_{\text{max}}(\tau)$, i.e. if task τ_i has the maximum computation requirement in τ, then we obtain the following upper bound on the value of m_1:

$$m_1 < \frac{u_{\text{sum}}(\tau) - u_{\text{max}}(\tau)}{1 - u_{\text{max}}(\tau)}. \tag{3.12}$$

□

Lemma 3.5. *Suppose $v_{sum}(\tau) > 1$. Let the tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$, be successfully mapped by the partitioning algorithm onto the available processors. When the partitioning algorithm is attempting to assign τ_i, if Condition 3.6 fails on m_2 processors then the following inequality must hold:*

$$m_2 < \frac{v_{sum}(\tau) - v_i}{1 - v_i}. \tag{3.13}$$

Proof. The proof is similar to the proof for Lemma 3.4. Also, just like in Lemma 3.4 we can show that for $v_{\text{sum}}(\tau) > 1$, if $v_i = v_{\max}(\tau)$, i.e. if task τ_i has the maximum memory requirement in τ, then we obtain the following upper bound on the value of m_2:

$$m_2 < \frac{v_{\text{sum}}(\tau) - v_{\max}(\tau)}{1 - v_{\max}(\tau)}. \tag{3.14}$$

□

We now present a sufficient schedulability condition for Algorithm APX-PARTITION which is applicable when $u_{\text{sum}}(\tau) > 1$ and $v_{\text{sum}}(\tau) > 1$:

Theorem 3.4. *A sporadic task system τ such that $u_{sum}(\tau) > 1$ and $v_{sum}(\tau) > 1$ is successfully scheduled by our algorithm on m unit-capacity and unit-memory processors, for any*

$$m \geq \left(\frac{u_{sum}(\tau) - u_{max}(\tau)}{1 - u_{max}(\tau)} + \frac{v_{sum}(\tau) - v_{max}(\tau)}{1 - v_{max}(\tau)} \right). \tag{3.15}$$

Proof. Our proof is by contradiction – we will assume that our algorithm fails to partition task system τ on m processors, and prove that in order for this to be possible m must violate Inequality 3.15 above.

Let us suppose that our partitioning algorithm fails to obtain a partition for τ on m unit-capacity processors. In particular, let us suppose that task τ_i cannot be mapped on to any processor. Let m_1 and m_2 denote (as in Lemmas 3.4-3.5 above) the number of processors on which Conditions 3.5 and 3.6 have failed respectively when we attempted to assign τ_i to some processor. It is necessary that:

$$m_1 + m_2 \geq m$$

(By Lemmas 3.4 and 3.5 respectively)

$$\left(\frac{u_{\text{sum}}(\tau) - u_{\max}(\tau)}{1 - u_{\max}(\tau)} + \frac{v_{\text{sum}}(\tau) - v_{\max}(\tau)}{1 - v_{\max}(\tau)} \right) > m_1 + m_2$$

$\Rightarrow$

$$\left(\frac{u_{\text{sum}}(\tau)-u_{\max}(\tau)}{1-u_{\max}(\tau)}+\frac{v_{\text{sum}}(\tau)-v_{\max}(\tau)}{1-v_{\max}(\tau)}\right) \quad > m. \tag{3.16}$$

Taking the contrapositive, it follows that the negation of Equation 3.16 is sufficient to ensure that our partitioning algorithm will successfully partition τ on m unit-capacity and unit-memory processors, as is claimed by the theorem.

□

Using Theorem 3.4 above, we now present resource-augmentation characterizations of our partitioning algorithm when it is used for partitioning implicit-deadline sporadic task systems.

Theorem 3.5. *Algorithm* APX-PARTITION *makes the following performance guarantee: if an implicit-deadline sporadic task system is feasible on m identical processors each of a particular computing capacity and memory, then Algorithm* APX-PARTITION *will successfully partition this task system upon a platform comprised of m processors that each have $(3-\frac{2}{m})$ times the computing capacity and memory as the original.*

Proof. Let us assume that $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is feasible on m processors each of computing capacity and memory equal to ξ. Since τ is feasible on m ξ-speed processors, it follows from Lemma 3.2 that the tasks in τ satisfy the following properties:

$$u_{\max}(\tau) \leq \xi,\ v_{\max}(\tau) \leq \xi,$$

and

$$u_{\text{sum}}(\tau) \leq m \cdot \xi,\ v_{\text{sum}}(\tau) \leq m \cdot \xi.$$

Suppose once again that τ is successfully scheduled by Algorithm APX-PARTITION on m unit-capacity and unit-memory processors.

Now we have four possibilities:

Case 1 $u_{\text{sum}}(\tau) \leq 1$, $v_{\text{sum}}(\tau) \leq 1$: According to Lemma 3.3 (Part A) and (Part B), Conditions 3.5 and 3.6 are always satisfied. For this case Algorithm APX-PARTITION is an optimal algorithm and the resource augmentation bound is 1.

Case 2 $u_{\text{sum}}(\tau) \leq 1$, $v_{\text{sum}}(\tau) > 1$: According to Lemma 3.3 (Part A), Condition 3.5 is always satisfied. Therefore the problem reduces to a problem in which there is only one limited resource. In this case we know from (Fisher, 2007, P.176) that the resource augmentation bound is $2 - \frac{1}{m}$.

Case 3 $u_{\text{sum}}(\tau) > 1$, $v_{\text{sum}}(\tau) \leq 1$: Same as above.

Case 4 $u_{\text{sum}}(\tau) > 1$, $v_{\text{sum}}(\tau) > 1$:

For this case we know from Theorem 3.4 that the task system is schedulable if:

$$m \geq \left(\frac{u_{\text{sum}}(\tau) - u_{\max}(\tau)}{1 - u_{\max}(\tau)} + \frac{v_{\text{sum}}(\tau) - v_{\max}(\tau)}{1 - v_{\max}(\tau)} \right).$$

We obtain an upper bound on the RHS of the above equation when:

- $u_{\text{sum}}(\tau) = m \cdot \xi$ and $v_{\text{sum}}(\tau) = m \cdot \xi$.
- $u_{\max}(\tau) = \xi$ and $v_{\max}(\tau) = \xi$.

Therefore, the task system is schedulable if:

$$\begin{aligned} m &\geq \frac{m\xi - \xi}{1 - \xi} + \frac{m\xi - \xi}{1 - \xi} \\ &\equiv m \geq \frac{\xi}{1 - \xi}(2m - 2) \\ &\equiv m - m\xi \geq (2m - 2)\xi \\ &\equiv \frac{1}{\xi} \geq \left(3 - \frac{2}{m}\right), \end{aligned}$$

which is as claimed in the statement of the theorem.

□

3.4.4 Heuristic improvements

The description of Algorithm APX-PARTITION, and the derivation of its resource augmentation bound make no assumptions about the order in which the tasks are considered for placement on the processors. In implementing Algorithm APX-PARTITION however, we may want to consider the tasks according to some ordering that enhances the likelihood that a given task system will be successfully partitioned. (For instance, since it is intuitively speaking more difficult to place a task that has larger computation and memory requirements, it may be better to consider such tasks for placement earlier when more computing capacity and memory are available on the processors.)

For any two tasks τ_i and τ_j, let us say that $\tau_i \trianglerighteq \tau_j$ if $u_i \geq u_j$ and $v_i \geq v_j$. A straightforward extension of the "decreasing" concept in First-Fit-Decreasing yields the following rule for ordering the tasks:

- If $\tau_i \trianglerighteq \tau_j$ then consider τ_i before considering τ_j (ties broken arbitrarily).

When we have tasks τ_i and τ_j such that neither $\tau_i \trianglerighteq \tau_j$ nor $\tau_j \trianglerighteq \tau_i$ hold (i.e., if $(u_i > u_j$ and $v_i < v_j)$ or $(u_i < u_j$ and $v_i > v_j)$), there are several possible generalizations to the FFD rule that we can come up with. Inspired by the proof of Theorem 3.4 consider tasks in *decreasing* order of f_i, where f_i is defined as follows:

$$f_i \stackrel{\text{def}}{=} \left(\frac{u_{\text{sum}}(\tau) - u_i}{1 - u_i} + \frac{v_{\text{sum}}(\tau) - v_i}{1 - v_i} \right). \tag{3.17}$$

Note that if there are no memory constraints (all the v_i's are zero), then ordering as per decreasing f_i reduces to FFD for partitioning tasks onto processors with limited computing capacity.

3.4.5 Experimental evaluation

We experimentally evaluate whether our heuristic in Section 3.4.4 for re-ordering tasks improves the schedulability of Algorithm APX-PARTITION. In order to do so, we randomly generated task sets and reordered the tasks in each task set according to the heuristic. We then determined if the

generated task sets and the reordered task sets could be partitioned by Algorithm APX-PARTITION. We measured the percentage of task sets that could be successfully partitioned in both cases.

The task sets were generated as follows. Each task set comprised n tasks. The UUnifast-Discard algorithm described in (Davis and Burns, 2009) was used to generate n values of computation requirement $\{u_1 \ldots u_n\}$ such that $\sum_{i=1}^{n} u_i = M$, and n values of memory requirement $\{v_1 \ldots, v_n\}$ such that $\sum_{i=1}^{n} v_i = M$, for some value of $M \leq m$. (The UUnifast-Discard algorithm generates values at random from a uniform distribution over the range $[0, 1]$). Each task τ_i in the task set had its computation requirement set equal to u_i, and memory requirement set equal to v_i, as computed above.

For $m = \{2, 4, 6, 8\}$ processors and for each M (total computation and memory requirement) starting from $M = m/2$ and incremented in steps of $m * 0.05$ until $M = m$, 1000 task sets were generated. We determined the percentage of the task sets that were schedulable under Algorithm APX-PARTITION without re-ordering and with re-ordering as per our heuristic. From the resulting graphs we made the following observations.

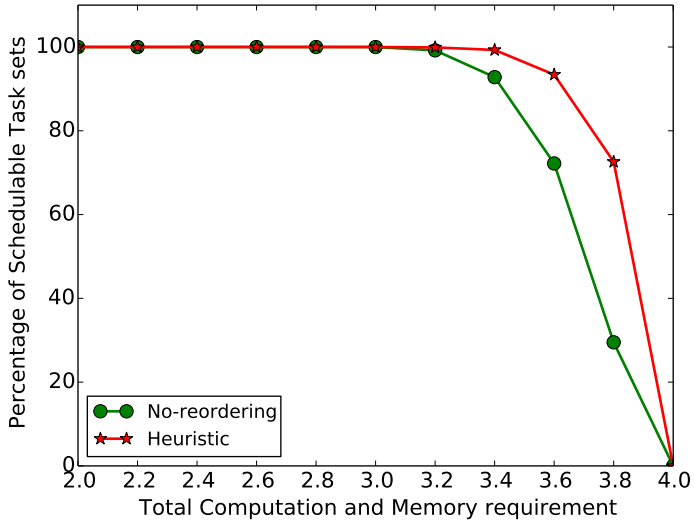


Figure 3.3: Evaluating partitioning heuristic: $m = 4, n = 40$

Observation 1. In Figure 3.3, we observe that for $m = 4$ processors and $n = 40$ tasks, more task sets are successfully partitioned by Algorithm APX-PARTITION after the tasks have been re-ordered as per our heuristic.

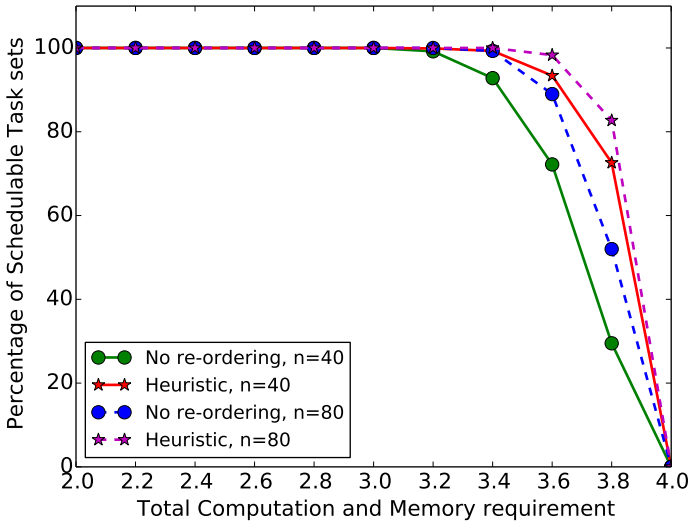


Figure 3.4: Evaluating partitioning heuristics: $m = 4, n = 40/80$

Observation 2. In Figure 3.4, we interpose the graph for $m = 4$ processors and $n = 40$ tasks, with the graph for $m = 4$ processors and $n = 80$ tasks. We observe that for both partitioning with no re-ordering and with our heuristics, we get better schedulability when the number of tasks in the task set is larger. This is because for larger number of tasks the computation and memory requirement of each task is smaller. Thus, the resource requirement of each task is smaller with respect to the size of the processor, and intuitively this increases schedulability.

These observations are consistent for experiments with different values of m.

3.4.6 Extending to > 2 distinct resource types

So far, we have restricted our attention to the partitioning of implicit-deadline sporadic task systems upon platforms in which each processor has limited amounts of two resources- computing capacity and local memory. Our results are easily generalized to platforms in which there are multiple resources on each processor (in addition to computing capacity), each available in limited

quantities. In order for these generalizations to hold it is required that each such additional resource be allocated in the specified amount "permanently" to each task throughout the duration of the run-time of the system.

More formally, suppose that there are ℓ kinds of resources (in addition to computing capacity). We characterize each sporadic task τ_i by

- Its *computation requirement*, using the traditional implicit-deadline sporadic-tasks model: $\tau_i = C_i, T_i$
- Its *resource requirements* $v_i[1], v_i[2], \ldots, v_i[\ell]$, with $v_i[p]$ denoting the fraction of the p'th resource that is locally available on each processor that must be reserved for the exclusive use of this task.

In determining whether such a task τ_i "fits" on processor π_k, Algorithm APX-PARTITION must ensure that π_k has enough of each of the ℓ resources; Condition 3.6 is therefore replaced by the more general condition:

$$\forall p \ : 1 \leq p \leq \ell : \quad \left(1 - \sum_{\tau_j \in \tau(\pi_k)} v_j[p]\right) \geq v_i[p]. \tag{3.18}$$

It can be shown that with this generalization to Condition 3.6, the resource augmentation bound of Theorem 3.5 becomes

$$\left(2 + \ell - \frac{1+\ell}{m}\right).$$

Observe that for $\ell \leftarrow 1$ we have, once again, the system model discussed in Section 3.1, and the bound above becomes exactly the bound of Theorem 3.5.

3.5 Conclusion

As embedded devices such as smart-phones increasingly come to be implemented upon multi-core and multiprocessor platforms, it becomes increasingly important that platform resources be

efficiently managed. However, much prior scheduling-theoretic research on obtaining multiprocessor implementations of real-time systems has focused almost exclusively on processor computing capacity, to the exclusion of other resources such as memory that are also available in scarce quantities on individual processors or cores. A few results have been obtained (for example, (Chekuri and Khanna, 2004) and (Baruah and Fisher, 2004; Fisher et al., 2005)) concerning the scheduling of systems in which the usage of multiple resources must be simultaneously optimized.

In this chapter we further explore the issue of resource allocation and scheduling on platforms that require the simultaneous management of multiple resources. We have described two partitioning algorithms, and derived their resource augmentation bound and run-time complexity. One is a PTAS and the other is a APX partitioning algorithm. Even though the PTAS partitioning algorithm has a better resource augmentation bound, we claim that the APX partitioning algorithm is more effective for partitioning a collection of implicit-deadline sporadic tasks on memory-constrained multiprocessor platforms. Unlike the PTAS partitioning algorithm, the APX partitioning algorithm is easy to implement and has a more desirable run-time computational complexity - linear in the product of the number of tasks in the task system and the number of processors. We have described a heuristic improvement for the APX partitioning algorithm, and performed schedulability experiments to show that our heuristic does in fact increase schedulability. We have also indicated how the APX partitioning algorithm can be extended to deal with additional resource constraints.

CHAPTER 4: MIXED CRITICALITY

In mixed-criticality (henceforth referred to as MC) systems, multiple functionalities that may be of different degrees of importance or criticalities are implemented upon a common platform. As more functionalities with different degrees of criticality are implemented on a common multiprocessor platform, mixed-criticality systems are becoming more complex, less uniform and predictable, and show greater variation in their performance. Certification of such systems is crucial to their successful deployment. In order to certify a system as being correct, the certification authority (CA) mandates certain assumptions about the worst-case behavior of the system during run-time. These assumptions are typically far more conservative than the assumptions that the system designer would use during the process of designing, implementing, and testing the system if subsequent certification was not required. (For instance, the worst-case execution time estimate used by the CA to characterize a complex piece of code is likely to be more pessimistic (i.e., larger) than the WCET estimate used by the system designer.) While the CA is only concerned with the correctness of the safety-critical part of the system the system designer is responsible for ensuring that the entire system is correct, including the non-critical parts.

The scheduling problem then becomes one of coming up with a single scheduling strategy that meets two separate goals, (i) certification of the high criticality jobs under more pessimistic assumptions, and (ii) schedulability of *all* the jobs (including the low criticality ones) under the system designer's less pessimistic assumptions.

A large body of recent research has addressed mixed-criticality scheduling for certifiability (see, e.g., (Vestal, 2007; Dorin et al., 2010; Lakshmanan et al., 2011; Park and Kim, 2011; Guan et al., 2011; Baruah et al., 2011; Baruah and Fohler, 2011; Tamas-Selicean and Pop, 2011; Huang et al., 2012; Herman et al., 2012; Baruah et al., 2012; Li and Baruah, 2012; Pathan, 2012)– this list

is by no means exhaustive). From among these papers, (Baruah et al., 2012) has results most closely related to our work on partitioned scheduling of mixed criticality systems.

In the following discussion we first describe the mixed criticality task model we will use in this chapter, and then present a variant of the EDF scheduling algorithm, Algorithm EDF-VD described in (Baruah et al., 2012) for scheduling mixed-criticality tasks on a uniprocessor. Finally, we present a mixed-criticality partitioning algorithm, which is a modification of Algorithm APX-PARTITION described in Section 3.4.1, for partitioning mixed-criticality tasks on m identical processors each having a computing capacity of 1. The tasks assigned to each processor are scheduled as per Algorithm EDF-VD. The mixed-criticality partitioning algorithm was first presented in our work (Baruah et al., 2014).

4.1 System Model

As with the task model described in Section 2.2 we will model a MC task system τ, as consisting of a finite specified collection of MC sporadic tasks each of which generate a potentially infinite sequence of MC jobs.

MC jobs. Each job is characterized by a 5-tuple of parameters:
$J_i = (a_i, d_i, \chi_i, c_i(\text{LO}), c_i(\text{HI}))$, where

- $a_i \in R^+$ is the release time.
- $d_i \in R^+$ is the deadline. We assume that $d_i \geq a_i$.
- $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes the criticality of the job. A HI-criticality job (a J_i with $\chi_i = \text{HI}$) is one that is subject to certification under the CA's pessimistic assumptions, whereas a LO-criticality job (a J_i with $\chi_i = \text{LO}$) is one that needs to be schedulable under the system designer's less pessimistic assumptions.
- $c_i(\text{LO})$ specifies the worst case execution time (WCET) estimate of J_i that is used by the system designer (i.e., the WCET estimate at the LO-criticality level).

- $c_i(\text{HI})$ specifies the worst case execution time (WCET) estimate of J_i that is used by the CA (i.e., the WCET estimate at the HI-criticality level). We assume that
  - $c_i(\text{HI}) \geq c_i(\text{LO})$ (i.e., the WCET estimate used by the system designer is never more pessimistic than the one used by the CA), and
  - $c_i(\text{HI}) = c_i(\text{LO})$ if $\chi_i = \text{LO}$ (i.e., a LO-criticality job is aborted if it executes for more than its LO-criticality WCET estimate[1]).

The MC job model has the following semantics. Job J_i is released at time a_i, has a deadline at d_i, and needs to execute for some amount of time γ_i. However, *the value of γ_i is not known beforehand, but only becomes revealed by actually executing the job until it* signals *that it has completed execution*. If J_i signals completion without exceeding $c_i(\text{LO})$ units of execution, we say that it has exhibited LO-criticality behavior. If it signals completion after executing for more than $c_i(\text{LO})$ but no more than $c_i(\text{HI})$ units of execution, we say that it has exhibited HI-criticality behavior. If it does not signal completion upon having executed for $c_i(\text{HI})$ units, we say that its behavior is erroneous.

MC implicit-deadline sporadic tasks. Each implicit-deadline sporadic task in the MC model is characterized by a 4-tuple of parameters: $\tau_k = (\chi_k, C_k(\text{LO}), C_k(\text{HI}), T_k)$, with the following interpretation. Task τ_k generates a potentially infinite sequence of MC jobs, with successive jobs being released at least T_k time units apart. Each such job has a deadline that is T_k time units after its release. The criticality of each such job is χ_k, and it has LO-criticality and HI-criticality WCET's of $C_k(\text{LO})$ and $C_k(\text{HI})$ respectively. A MC **sporadic task system** is specified as a finite collection of such sporadic tasks.

Utilizations. The utilization or computing requirement of a implicit-deadline sporadic task described in Section 2.2 denotes the ratio of its WCET to its period; the utilization of a task system denotes the sum of the utilizations of all the tasks in the system. We now define analogous concepts

[1]We assume that the run-time system provides support for ensuring that jobs do not execute for more than a specified amount; see, e.g., (Baruah et al., 2011) for a discussion of this issue.

	χ_k	$C_k(\text{LO})$	$C_k(\text{HI})$	T_k
τ_1	LO	2	2	6
τ_2	HI	1	2	10
τ_3	HI	2	10	20

Table 4.1: An example mixed-criticality implicit-deadline sporadic task system.

for mixed-criticality sporadic task systems. Let $U_k(\text{LO})$ and $U_k(\text{HI})$ denote the LO-criticality and HI-criticality utilizations of task τ_k such that:

$$U_k(\text{LO}) := C_k(\text{LO})/T_k \text{ and } U_k(\text{HI}) := C_k(\text{HI})/T_k.$$

Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a MC implicit-deadline sporadic task system. For each of x and y in $\{\text{LO}, \text{HI}\}$, we define a utilization parameter as follows:

$$U_x^y(\tau) = \sum_{\tau_i \in \tau \wedge \chi_i = x} U_i(y).$$

Thus for example, $U_{\text{HI}}^{\text{LO}}(\tau)$ denotes the sum of the utilizations of the HI-criticality tasks in τ, under the assumption that each job of each task executes for no more than its LO-criticality WCET.

Example 4.1. Consider the task system depicted in Table 4.1. For this task system,

$$\begin{aligned} U_{\text{LO}}^{\text{LO}}(\tau) &= 2/6 = 0.33 \\ U_{\text{LO}}^{\text{HI}}(\tau) &= 2/6 = 0.33 \\ U_{\text{HI}}^{\text{LO}}(\tau) &= 1/10 + 2/20 = 0.2 \\ U_{\text{HI}}^{\text{HI}}(\tau) &= 2/10 + 10/20 = 0.7. \end{aligned}$$

$\diamond$

Scheduling MC implicit-deadline sporadic task systems. A particular MC implicit-deadline sporadic task system may generate different instances of jobs during different runs. Furthermore,

during any given run each job comprising the instance may exhibit LO-criticality, HI-criticality, or erroneous behavior. We define an algorithm for scheduling MC implicit-deadline sporadic task system τ to be *correct* if it is able to schedule every instance generated by τ such that:

- If all jobs exhibit LO-criticality behavior, then all jobs receive enough execution between their release time and deadline to be able to signal completion, and
- If *any* job exhibits HI-criticality behavior, then all HI-criticality jobs receive enough execution between their release time and deadline to be able to signal completion.

Note that if any job exhibits HI-criticality behavior, we do not require any LO-criticality jobs (including those that may have arrived before this happened) to complete by their deadlines. This is an implication of the requirements of certification: informally speaking, the system designer fully expects that all jobs will exhibit LO-criticality behavior, and hence is only concerned that they behave as desired under these circumstances. The CA on the other hand, allows for the possibility that some jobs may exhibit HI-criticality behavior and requires that all HI-criticality jobs nevertheless meet their deadlines.

4.2 EDF for Mixed Criticality systems

Algorithm EDF-VD (for ***E**arliest **D**eadline **F**irst with **V**irtual **D**eadlines*) of (Baruah et al., 2012) is derived from preemptive EDF for scheduling mixed-criticality implicit-deadline sporadic task systems upon a uniprocessor. We now provide a high-level description of EDF-VD.

Let $\tau = \{\tau_1, \ldots, \tau_n\}$ denote the MC implicit-deadline sporadic task system that is to be scheduled on a processor with unit computing capacity. EDF-VD's approach to scheduling τ can be thought of as a three-phased one:

1. An initial pre-processing phase occurs prior to run-time.

2. During run-time, jobs are initially dispatched in the expectation that the behavior of the system is going to be a LO-criticality one; no job will execute for more than its LO-criticality WCET.

3. If some job does execute beyond its LO-criticality WCET without signaling that it has completed execution, the dispatching algorithm is modified accordingly and the algorithm enters its optional third phase.

We now discuss each of the three phases.

During *pre-processing*, a schedulability test is performed to determine whether τ can be guaranteed to be successfully scheduled. If it is determined that τ can be successfully scheduled, then an additional parameter, called a *modified period* denoted $\hat{T}_i$, is computed for each HI-criticality task $\tau_i \in \tau$. It is always the case that $\hat{T}_i \leq T_i$.

Initial run-time dispatching is done according to EDF. Since EDF is defined for regular rather than mixed-criticality task systems, we map the mixed-criticality tasks in τ to regular implicit-deadline sporadic tasks as follows: each LO-criticality task $\tau_k = (\chi_k, C_k(\text{LO}), C_k(\text{HI}), T_k)$ is mapped to a regular task $(C_k(\text{LO}), T_k)$, while each HI-criticality task $\tau_k = (\chi_k, C_k(\text{LO}), C_k(\text{HI}), T_k)$ is mapped to a regular task $(C_k(\text{LO}), \hat{T}_k)$, where the $\hat{T}_k$'s are the modified periods computed during the pre-processing phase.

If some job does execute beyond its LO-criticality WCET without signaling that it has completed execution, we enter the *third phase* of the algorithm, and the following changes occur.

1. All currently-active LO-criticality jobs are immediately discarded; henceforth, no LO-criticality job will receive any execution.

2. Subsequent run-time scheduling of the HI-criticality tasks (including their jobs that are currently active) are done according to EDF. This is done by mapping each HI-criticality MC task $\tau_k = (\chi_k, C_k(\text{LO}), C_k(\text{HI}), T_k)$ in τ to a regular task $(C_k(\text{HI}), T_k)$.

EDF-VD Properties. The following properties of Algorithm EDF-VD, derived in (Baruah et al., 2012), are used in deriving the partitioning algorithm described next.

Theorem 4.1. *(Baruah et al., 2012). Any mixed-criticality implicit-deadline sporadic task system τ satisfying the property*

$$\max\left(U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau), U_{\text{HI}}^{\text{HI}}(\tau)\right) \leq 3/4$$

is successfully scheduled by EDF-VD on a unit-speed processor.

Theorem 4.1 asserts that any MC implicit-deadline sporadic task system for which both the LO-criticality utilization $(U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau))$ and the HI-criticality utilization $(U_{\text{HI}}^{\text{HI}}(\tau))$ are $\leq 3/4$ is successfully scheduled by EDF-VD on a processor with unit computing capacity.

Since utilization not exceeding processor speed is a necessary condition for schedulability, a resource augmentation bound, also called speedup bound when it refers to augmenting processor speed or computing capacity, of $4/3$ for EDF-VD immediately follows:

Corollary 4.1. *(Baruah et al., 2012). EDF-VD has a speedup bound no greater than* $4/3$.

Theorem 4.1 guarantees schedulability when both LO-criticality and HI-criticality utilizations are bounded by $3/4$. A more general test for when one of the utilizations is greater than $3/4$, and the other less than $3/4$ has also been derived.

Theorem 4.2. *(Baruah et al., 2012). Any mixed-criticality implicit-deadline sporadic task system τ satisfying the property*

$$U_{\text{LO}}^{\text{LO}}(\tau) \leq \frac{1 - U_{\text{HI}}^{\text{HI}}(\tau)}{1 - \left(U_{\text{HI}}^{\text{HI}}(\tau) - U_{\text{HI}}^{\text{LO}}(\tau)\right)}$$

is successfully scheduled by EDF-VD on a unit-speed processor.

Both Theorems 4.1 and 4.2 are sufficient schedulability conditions.

4.3 Algorithm MC-partition

We start with an overview of our partitioning algorithm. It proceeds in two phases:

1. During the first phase each HI-criticality task is assigned to some processor while ensuring that the cumulative HI-criticality utilization assigned to each processor does not exceed $3/4$.

2. During the second phase each LO-criticality task is assigned to some processor while ensuring that the cumulative LO-criticality utilization assigned to each processor also does not exceed $3/4$.

Observe that by Theorem 4.1, such an assignment procedure ensures that each processor remains schedulable by EDF-VD. The algorithm reports failure if it fails to successfully assign every task.

The details are as follows. Let τ denote the MC implicit-deadline sporadic task system that is to be partitioned amongst m processors. Let us assume that there are n tasks in τ, of which n_1 are HI-criticality tasks. Without loss of generality, assume that $\{\tau_1, \tau_2, \ldots, \tau_{n_1}\}$ are the HI-criticality tasks, and $\{\tau_{n_1+1}, \ldots, \tau_n\}$ the LO-criticality ones. Let $\{\pi_1, \pi_2, \ldots, \pi_m\}$ denote the m processors. Figure 4.1 gives a pseudo-code representation of Algorithm MC-PARTITION.

Let us suppose that tasks $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ have been successfully assigned. We now explain how task τ_i is assigned to a processor.

For any processor π_k, let $\tau(\pi_k)$ denote the tasks from amongst $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ that have already been assigned to it. Algorithm MC-PARTITION assigns the task τ_i to any processor π_k satisfying the following condition. If $i \leq n_1$ (i.e., if τ_i is a HI-criticality task) then:

$$\left(U_i(\text{HI}) + \sum_{\tau_j \in \tau(\pi_k)} U_j(\text{HI}) \right) \leq \frac{3}{4}, \tag{4.1}$$

else (i.e., $i > n_1$ and τ_i is hence a LO-criticality task)

$$\left(U_i(\text{LO}) + \sum_{\tau_j \in \tau(\pi_k)} U_j(\text{LO}) \right) \leq \frac{3}{4}. \tag{4.2}$$

If no such π_k exists, then Algorithm MC-PARTITION declares failure: it is unable to partition τ upon the m-processor platform.

```
MC-PARTITION(τ, m)
    ▷ τ = {τ_1, ..., τ_n} is to be partitioned on m identical, unit-
      capacity processors denoted {π_1, ..., π_m}. Tasks {τ_1, ..., τ_{n_1}}
      are HI-criticality tasks; tasks {τ_{n_1+1}, ..., τ_n} are LO-criticality
      tasks. The set of tasks assigned to processor π_k is denoted as
      τ(π_k); initially, τ(π_k) ← ∅ for all k.
 1  for i ← 1 to n_1 ▷ Phase 1: HI-criticality tasks
 2          for k ← 1 to m
 3                  if τ_i satisfies Condition 4.1 on processor π_k
                      then ▷ assign τ_i to π_k;
 4                           τ(π_k) ← τ(π_k) ⋃ {τ_i}
 5                           break;
 6          end (of inner for loop)
 7          if (k > m) return PARTITIONING FAILED
 8  end (of outer for loop)
 9  for i ← (n_1 + 1) to n ▷ Phase 2: LO-criticality tasks
10          for k ← 1 to m
11                  if τ_i satisfies Condition 4.2 on processor π_k
                      then ▷ assign τ_i to π_k;
12                           τ(π_k) ← τ(π_k) ⋃ {τ_i}
13                           break;
14          end (of inner for loop)
15          if (k > m) return PARTITIONING FAILED
16  end (of outer for loop)
17  return PARTITIONING SUCCEEDED
```

Figure 4.1: Pseudo-code for Algorithm MC-PARTITION

The following lemma asserts that in assigning a task τ_i to a processor π_k Algorithm MC-PARTITION does not adversely affect the schedulability of the tasks previously assigned to the processors.

Lemma 4.1. *If the tasks previously assigned to each processor were schedulable on that processor by EDF-VD and Algorithm* MC-PARTITION *assigns task τ_i to processor π_k, then the tasks assigned to each processor (including processor π_k) remain schedulable on that processor by EDF-VD.*

Proof. Observe that the schedulability of the processors other than processor π_k is not affected by the assignment of task τ_i to processor π_k. It remains to demonstrate that, if the tasks assigned to processor π_k were schedulable by EDF-VD prior to the assignment of τ_i and Algorithm MC-PARTITION assigns τ_i to π_k, then the tasks on π_k remain schedulable by EDF-VD after adding τ_i. To see that this is true we consider two cases.

- If $i \leq n_1$, Condition (4.1) must hold for Algorithm MC-PARTITION to assign τ_i to π_k. This condition ensures that the sum of the HI-criticality utilizations of all HI-criticality tasks assigned to processor π_k remains $\leq 3/4$. Since each task's LO-criticality utilization is no greater than its HI-criticality utilization, this also means that the sum of the LO-criticality utilizations of all tasks assigned to processor π_k remains $\leq 3/4$.
- If $i > n_1$, Condition (4.2) must hold for Algorithm MC-PARTITION to assign τ_i to π_k. This condition ensures that the sum of the LO-criticality utilizations of all tasks assigned to processor π_k remains $\leq 3/4$ while the sum of the HI-criticality utilizations of HI-criticality tasks does not change.

It is thus the case that both the sum of the HI-criticality utilizations and the sum of the LO-criticality utilizations upon each processor remains $\leq 3/4$. The correctness of the lemma then follows from Theorem 4.1. □

The correctness of Algorithm MC-PARTITION can now be established by repeated applications of Lemma 4.1.

Theorem 4.3. *If Algorithm* MC-PARTITION *returns* PARTITIONING SUCCEEDED *on task system* τ*, then the resulting partitioning is schedulable by EDF-VD.*

Proof. Observe that Algorithm MC-PARTITION returns PARTITIONING SUCCEEDED if and only if it has successfully assigned each task in τ to some processor.

Prior to the assignment of task τ_1 each processor has been assigned no tasks, and is therefore trivially schedulable by EDF-VD. It follows from Lemma 4.1 that all processors remain schedulable

by EDF-VD after each task assignment as well. Hence, all processors are schedulable by EDF-VD after all tasks in τ have been successfully assigned. □

4.3.1 Run-time complexity

Algorithm MC-PARTITION can be implemented to maintain, for each processor, the cumulative HI-criticality and LO-criticality utilizations of all the tasks that have been assigned to that processor thus far. For each task τ_i and each processor π_k, Condition (4.1) or Condition (4.2) can then be evaluated in constant time. Therefore the i'th task can be assigned in $\mathcal{O}(m)$ time. For n tasks this yields an overall run-time complexity of $\mathcal{O}(n \times m)$.

4.3.2 Speedup bound

We now derive a *sufficient* schedulability condition for Algorithm MC-PARTITION in Lemma 4.2 below, and use this schedulability condition to derive a speedup bound for Algorithm MC-PARTITION in Theorem 4.4.

We would like to stress that Lemma 4.2 is not intended to be used as a schedulability test to determine whether Algorithm MC-PARTITION would successfully schedule a given sporadic task system – since the algorithm itself runs efficiently in polynomial time, the "best" (i.e., most accurate) polynomial-time sufficient schedulability test for determining whether a particular task system is successfully scheduled by it is to actually run Algorithm MC-PARTITION.

Lemma 4.2. *Suppose that Algorithm* MC-PARTITION *fails to assign some task τ_i. One or both of the following conditions must hold:*

$$U_{\text{HI}}^{\text{HI}}(\tau) > \frac{3}{4}m - (m-1)\,U_i(\text{HI}), \tag{4.3}$$

$$U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau) > \frac{3}{4}m - (m-1)\,U_i(\text{LO}). \tag{4.4}$$

Proof. Let us first consider the case when $i \leq n_1$. Since τ_i cannot be accommodated on any processor, Condition (4.1) must be violated for task τ_i on each of the m processors. Summing the

negation of Condition (4.1) across all m processors, we have

$$\begin{aligned}
&\big(\frac{3}{4} - U_i(\text{HI})\big)m < \sum_{j=1}^{i-1} U_j(\text{HI}) \\
\Leftrightarrow \quad &\frac{3}{4}m - mU_i(\text{HI}) + U_i(\text{HI}) < \sum_{j=1}^{i-1} U_j(\text{HI}) + U_i(\text{HI}) \\
\Leftrightarrow \quad &\frac{3}{4}m - (m-1)U_i(\text{HI}) < \sum_{j=1}^{i} U_j(\text{HI}) \\
\Rightarrow \quad &\frac{3}{4}m - (m-1)U_i(\text{HI}) < U_{\text{HI}}^{\text{HI}}(\tau)
\end{aligned}$$

which is as claimed.

Now let us consider when $i > n_1$. Since τ_i cannot be accommodated on any processor, Condition (4.2) must be violated for task τ_i on each of the m processors. Summing the negation of Condition (4.2) across all m processors, we have

$$\begin{aligned}
&\big(\frac{3}{4} - U_i(\text{LO})\big)m < \sum_{j=1}^{i-1} U_j(\text{LO}) \\
\Leftrightarrow \quad &\frac{3}{4}m - mU_i(\text{LO}) + U_i(\text{LO}) < \sum_{j=1}^{i-1} U_j(\text{LO}) + U_i(\text{LO}) \\
\Leftrightarrow \quad &\frac{3}{4}m - (m-1)U_i(\text{LO}) < \sum_{j=1}^{i} U_j(\text{LO}) \\
\Rightarrow \quad &\frac{3}{4}m - (m-1)U_i(\text{LO}) < U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau)
\end{aligned}$$

which is also as claimed in the lemma. □

Using Lemma 4.2 above, we now derive a speedup bound for our partitioning algorithm.

Theorem 4.4. *The speedup bound of Algorithm* MC-PARTITION *on an m-processor platform is* $(\frac{8m-4}{3m})$.

Proof. To prove this, we must show that any MC implicit-deadline sporadic task system that can be partitioned upon an m-processor platform by an optimal algorithm can be partitioned by Algorithm MC-PARTITION upon an m-processor platform in which each processor is $(\frac{8m-4}{3m})$ times as fast.

Let us assume that $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ can be scheduled by an optimal scheduling algorithm on m processors each of computing capacity equal to ξ. It must therefore be the case that

$$\begin{aligned} U_i(\text{LO}) &\leq \xi \quad \text{for each } i, 1 \leq i \leq n \\ U_i(\text{HI}) &\leq \xi \quad \text{for each } i, 1 \leq i \leq n_1 \\ U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau) &\leq m\xi \\ U_{\text{HI}}^{\text{HI}}(\tau) &\leq m\xi. \end{aligned}$$

Suppose that Algorithm MC-PARTITION fails to partition τ on m unit-capacity processors. By Lemma 4.2 above, it must be the case that at least one of Conditions (4.3) or (4.4) holds. If Condition (4.3) holds, it must be the case that

$$\begin{aligned} & U_{\text{HI}}^{\text{HI}}(\tau) > \frac{3}{4}m - (m-1)\,U_i(\text{HI}) \\ \Rightarrow \quad & m\xi > \frac{3}{4}m - (m-1)\xi \\ \Leftrightarrow \quad & (2m-1)\xi > \frac{3}{4}m \\ \Leftrightarrow \quad & \xi > \frac{3m}{4(2m-1)}. \end{aligned}$$

Similarly if Condition (4.4) holds, it must be the case that

$$\begin{aligned} & U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau) > \frac{3}{4}m - (m-1)\,U_i(\text{LO}) \\ \Rightarrow \quad & m\xi > \frac{3}{4}m - (m-1)\xi \\ \Leftrightarrow \quad & (2m-1)\xi > \frac{3}{4}m \\ \Leftrightarrow \quad & \xi > \frac{3m}{4(2m-1)}. \end{aligned}$$

We have shown that for either of Conditions (4.3) or (4.4) to hold, ξ must exceed $\frac{3m}{4(2m-1)}$. Hence if $\xi \leq \frac{3m}{4(2m-1)}$ then τ is successfully scheduled by Algorithm MC-PARTITION on m unit-speed processors; equivalently, if $\xi \leq 1$ then τ is successfully scheduled by Algorithm MC-PARTITION on m speed-$\frac{4(2m-1)}{3m}$ processors, as claimed by the theorem. □

We note that $\frac{4(2m-1)}{3m} < 8/3$ for all $m \geq 1$, asymptotically approaching $8/3$ as $m \to \infty$. Hence, $8/3 \approx 2.67$ is an upper bound on the speedup of Algorithm MC-PARTITION, for all values of m.

4.3.3 Pragmatic improvements

We now describe two modifications to Algorithm MC-PARTITION. The exact speedup bound of these modified algorithms is not known.

Algorithm MC-PARTITION-UT-0.75. This version incorporates two modifications:

§1: Preprocessing tasks with $U_i(\text{HI}) > 3/4$. Algorithm MC-PARTITION is modified to incorporate the partitioning of HI-criticality tasks with $3/4 < U_i(\text{HI}) \leq 1$. The modification assigns one such HI-criticality task per processor prior to partitioning other tasks (we call this the *pre-processing phase*.) Suppose m' processors were assigned HI-criticality tasks during the pre-processing phase. Each of these m' processors will be assigned HI-criticality tasks as long as the HI-criticality utilization does not exceed 1. Hence during phase 1 of Algorithm MC-PARTITION each of the m' processors are assigned HI-criticality tasks so long as the following condition is satisfied:

$$\left(U_i(\text{HI}) + \sum_{\tau_j \in \tau(\pi_k)} U_j(\text{HI}) \right) \leq 1. \tag{4.5}$$

For the remaining processors, Condition (4.1) remains the requirement for assigning HI-criticality tasks during phase 1.

§2: Improved utilization during phase 2. Condition (4.2) is based upon Theorem 4.1. As stated in Section 4.2, the schedulability test in Theorem 4.2 is superior to the one in Theorem 4.1. We therefore replace Condition (4.2) with the following condition based upon Theorem 4.2:

$$\left(U_i(\text{LO}) + \sum_{\tau_j \in \tau(\pi_k) \wedge \chi_j = \text{LO}} U_j(\text{LO}) \right) \leq \frac{1 - U_{\text{HI}}^{\text{HI}}(\tau(\pi_k))}{1 - \left(U_{\text{HI}}^{\text{HI}}(\tau(\pi_k)) - U_{\text{HI}}^{\text{LO}}(\tau(\pi_k)) \right)}. \tag{4.6}$$

It follows from Theorem 4.2 that satisfying Condition (4.6) will ensure that the system is schedulable. Furthermore, it is possible that tasks with LO-criticality utilization $> 3/4$ will be accommodated upon some processor.

Algorithm MC-PARTITION-UT-INC. This is obtained by replacing Condition (4.1) by:

$$\left(U_i(\text{HI}) + \sum_{\tau_j \in \tau(\pi_k)} U_j(\text{HI}) \right) \leq val \tag{4.7}$$

where *val* is a variable that iteratively takes on values in the range $[0.5, 1]$ (in pre-determined steps). The intuition behind this modification is that depending upon the value of *val* the HI-criticality tasks are assigned to processors differently, which in turn affects the partitioning of the LO-criticality tasks. As a result, different values of *val* might result in a success or failure in partitioning different task systems. In the pre-processing phase of Algorithm MC-PARTITION-UT-INC, HI-criticality tasks with utilization greater than *val* are assigned to the processors. Suppose m' processors were each assigned a HI-criticality task during the pre-processing phase. The remaining HI-criticality are assigned to the processors while ensuring that Condition (4.5) is satisfied on the m' processors and Condition (4.7) is satisfied on the processors excluding the m' processors. Also, Condition (4.2) for LO-criticality tasks is replaced by Condition (4.6). Algorithm MC-PARTITION-UT-INC returns PARTITIONING FAILED only if partitioning fails for all the values of *val* that are considered.

It is evident that Algorithm MC-PARTITION-UT-INC dominates Algorithms MC-PARTITION-UT-0.75 since MC-PARTITION-UT-INC checks with different values of *val* (0.75 can be included as a value for *val*), and returns PARTITIONING FAILED only if the partitioning failed for all the values that were considered.

4.3.4 Experimental evaluation

We experimentally evaluate whether the pragmatic improvements described in Section 4.3.3 improve the schedulability of Algorithm MC-PARTITION. In order to experimentally evaluate the improvements we randomly generated task sets and determined if the task sets could be partitioned

by Algorithm MC-PARTITION, MC-PARTITION-UT-0.75, and MC-PARTITION-UT-INC. We measured the percentage of task sets that could be successfully partitioned by each algorithm. We also compared the schedulability of the above partitioning algorithms with the schedulability of the worst-case partitioning algorithm, we refer to it as Algorithm WC-PARTITION, which corresponds to partitioning the HI-criticality tasks as per their HI-criticality utilization and the LO-criticality tasks as per their LO-criticality utilization such that for each processor π_k:

$$\sum_{\tau_i \in \pi_k} U_i(\chi_j) \leq 1.$$

The task sets were generated as follows. Each task set comprised n tasks. The UUnifast-Discard algorithm described in (Davis and Burns, 2009) was used to generate n utilization values $\{u_1 \dots u_n\}$ such that $\sum_{i=1}^{n} u_i = M$, for some value of $M \leq m$. The probability of a task becoming a HI-criticality task was determined by the parameter CP, $0 \leq CP \leq 1$. If a task τ_i became a HI-criticality task, then $U_i(\text{HI})$ was set equal to u_i, else $U_i(\text{LO})$ was set equal to u_i. Thus the total worst-case utilization $U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{HI}}(\tau)$, of a task set τ was equal to M. For a HI-criticality task the ratio of its HI-criticality utilization to its LO-criticality utilization was uniformly drawn from the range $[1, CF]$, where CF is called the criticality factor.

For $m = \{2, 4, 6, 8\}$ processors and for each value of M (total worst-case utilization) starting from $M = m/2$ and incremented in steps of $m * 0.05$ until $M = m$, 1000 task sets were generated. We determined the percentage of task sets that were schedulable under Algorithm WC-PARTITION, MC-PARTITION, MC-PARTITION-UT-0.75, and MC-PARTITION-UT-INC. From the resulting graphs we made the following observations.

Observation 1. In Figure 4.2 we observe that for $m = 4$, $n = 20$, $CP = 0.5$, and $CF = 8$, Algorithm MC-PARTITION-UT-INC is able to schedule more task sets than Algorithm MC-PARTITION-UT-0.75, and Algorithm MC-PARTITION. Note that Algorithm MC-PARTITION is unable to partition tasks with HI-criticality or LO-criticality utilization above 0.75. Thus, for higher values of total worst-case utilization AlgorithmMC-PARTITION has the least schedulability, since it

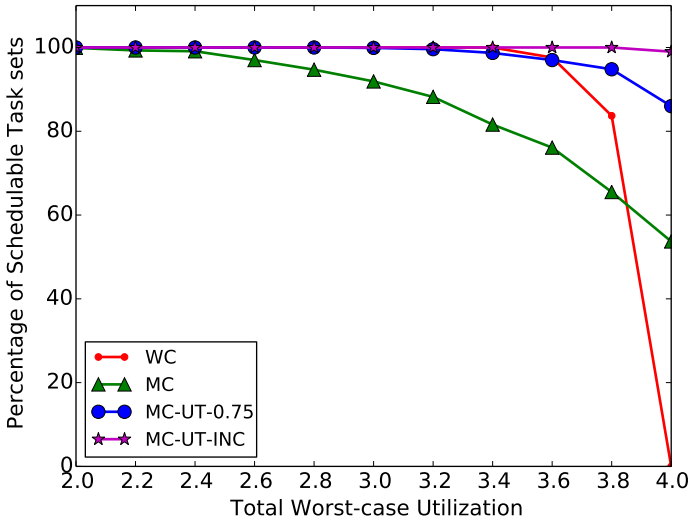


Figure 4.2: Evaluating mixed-criticality partitioning algorithms: $m = 4, n = 20, CP = 0.5, CF = 8$

is possible that the task set consists of a task with HI-criticality or LO-criticality utilization greater than 0.75.

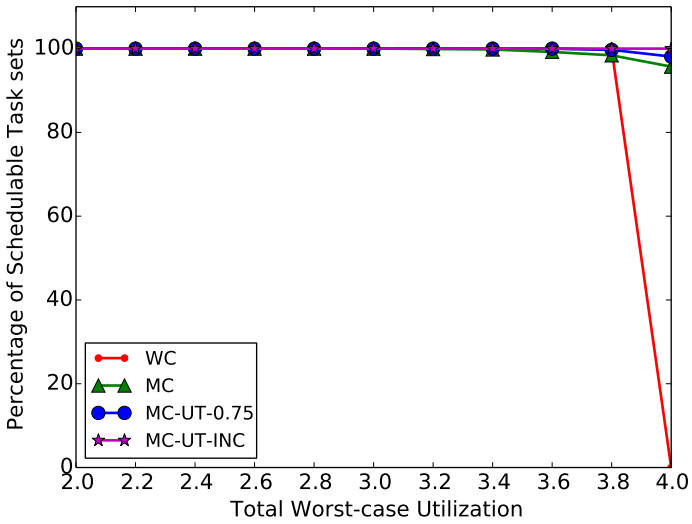


Figure 4.3: Evaluating mixed-criticality partitioning algorithms: $m = 4, n = 40, CP = 0.5, CF = 8$

Observation 2. In Figure 4.2 where $n = 20$, vs. Figure 4.3 where $n = 40$, we observe that the schedulability of all the algorithms increase. All parameters, except n, in both these figures are the

same. This is because for larger number of tasks the utilization of each task in a task set is smaller, and intuitively this increases the schedulability of partitioning algorithms.

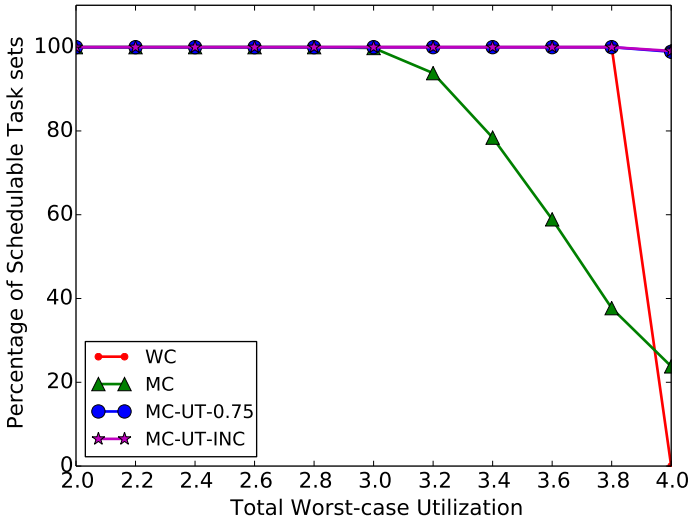


Figure 4.4: Evaluating mixed-criticality partitioning algorithms: $m = 4, n = 40, CP = 0.2, CF = 8$

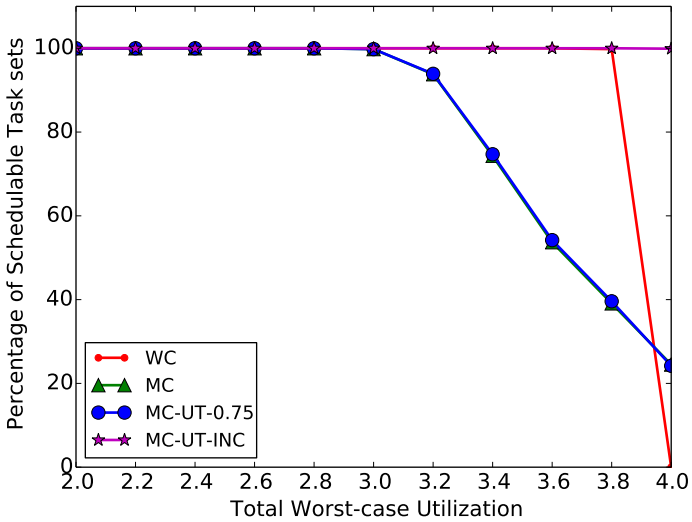


Figure 4.5: Evaluating mixed-criticality partitioning algorithms: $m = 4, n = 40, CP = 0.8, CF = 8$

Observation 3. When the probability CP is such that there are many HI-criticality tasks or many LO-criticality tasks, for example in Figure 4.4 where $CP = 0.2$ and in Figure 4.5 where $CP = 0.8$, the schedulability of Algorithm MC-PARTITION decreases because it is unable to use more than

75% of the HI-criticality or LO-criticality computing capacity. In the case where there are many HI-criticality tasks, for example in Figure 4.5 where $CP = 0.8$, the schedulability of Algorithm MC-PARTITION-UT-0.75 also decreases because it is unable to schedule the larger HI-criticality workload. Under both these situations Algorithm MC-PARTITION-UT-INC is able to maintain schedulability that is comparable to that of Figure 4.3 where $CP = 0.5$.

These observations were consistent with different values of m.

4.4 EDF-VD Extended

We have thus far assumed that our system consists of tasks with two criticality levels denoted as LO and HI. In many safety-critical application domains there may be functionalities with more than two criticality levels. For instance, the DO-178B standard specifies five criticality levels while the IEC 61508 international standard for industrial use recommends four different *Safety Integrity Levels (SILS)*. In this section we describe an extension to the EDF-VD scheduling algorithm (Baruah et al., 2012) that incorporates a fixed number L of criticality levels. We also show how to incorporate the *pessimism of run-time parameters* for higher criticality tasks in terms of larger worst-case execution times and also in terms of the *shorter periods*, that is we enable consecutive jobs of a higher criticality task to arrive at higher frequency (Baruah, 2012).

Task Model. The following task model is an extension to the mixed-criticality task model described in Section 4.1. It incorporates any fixed number L of criticality levels and enables pessimism in run-time parameters for each criticality level. We characterize an implicit-deadline sporadic mixed-criticality task τ_i by the following parameters: $\tau_i = (\chi_i, \{C_i(1), C_i(2), \ldots, C_i(L)\}, \{T_i(1), T_i(2), \ldots, T_i(L)\})$, where

- $\chi_i \in 1, 2, \ldots, L$ denotes the criticality. A task τ_i with $\chi_i = k$, where $k \leq L$, must be certified to be schedulable assuming that it can require up to $C_i(k)$ units of processing time, and it can arrive with a frequency equal to but no sooner than $T_i(k)$.

- $C_i(1)$ is the WCET of task τ_i at criticality level 1, $C_i(2)$ is the WCET of task τ_i at criticality level 2, ..., $C_i(L)$ is the WCET of task τ_i at criticality level L. We assume that $C_i(1) \leq C_i(2) \ldots \leq C_i(L)$

- $T_i(1)$ is the minimum frequency of task τ_i at criticality level 1, $T_i(2)$ is the minimum frequency of task τ_i at criticality level 2, ..., $T_i(L)$ is the minimum frequency of task τ_i at criticality level L. We assume that $T_i(1) \geq T_i(2) \ldots \geq T_i(L)$

A MC task system τ is a finite collection of MC tasks: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. We study the problem of scheduling such MC task systems on a uniprocessor platform.

Behaviors. A system is considered to exhibit *k-criticality behavior* if the execution of the system satisfies the property that for each task τ_i, all jobs of τ_i execute for at most $C_i(k)$ time units and successive jobs of τ_i arrive at least $T_i(k)$ time units apart. Any execution that exhibits behavior not allowed by any of the k criticality levels is said to be erroneous.

Correctness. A scheduling algorithm for the given system is said to be correct if it satisfies the property that for each task τ_i with criticality at least k, all jobs of τ_i complete execution before there deadline in any k-criticality behavior of the system.

We modify the EDF-VD scheduling algorithm for the mixed-criticality model described above. We also present a sufficient schedulability analysis for the derived scheduling algorithm.

The idea behind our modification to the EDF-VD scheduling algorithm is similar to the idea presented in (Baruah, 2012) and (Baruah et al., 2012) for a system with two criticality levels. The algorithm essentially ensures that if the system behavior is compliant with some k'-criticality behavior, jobs of all tasks with criticality greater than k' complete execution well before their deadline. This guarantees that if the system changes to a higher k-criticality behavior ($k' < k$) during run-time then there is sufficient computing capacity available to nevertheless complete all jobs of all tasks with criticality at least k by their deadlines, after discarding all jobs with criticality less than k.

More specifically, suppose that the system is exhibiting $(k-1)$-criticality behavior and at some time-instant t_k a job of some task τ_i executes for longer than $C_i(k-1)$ but at most $C_i(k)$ time units,

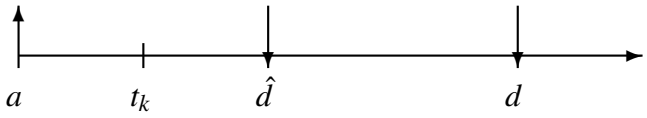


Figure 4.6: A k-criticality job arrives at time a, with deadline at d. It is scheduled using the modified deadline $\hat{d}$, which is $\leq d$. If there is no criticality change then this job can complete execution by $\hat{d}$. However, if there is a criticality change at t_k then only jobs with criticality at least k execute as per their k-criticality behavior. In the latter case, this job can meet its deadline by *only* executing over $[\hat{d}, d)$.

or if successive jobs of task τ_i arrive less than $T_i(k-1)$ but at least $T_i(k)$ time units apart. Thus a criticality level change has been triggered at time-instant t_k, and we are not required to demonstrate that jobs of tasks with criticality less than k will complete execution by their deadline. Informally, we would like to ensure that there is enough capacity available on the processor for jobs of each task τ_i with criticality at least k that happen to be currently-active –arrived but not completed execution– complete execution by their deadline, provided only jobs of tasks with criticality at least k execute henceforth (see Figure 4.6). To achieve this we ensure that as long as the system's behavior is consistent with some criticality less than k ($(k-1)$ in this case), τ_i's job would have completed execution well before its actual deadline (by $\hat{d}$ in Figure 4.6 whereas the actual deadline is denoted d). In this way we ensure that if τ_i's job is active at time-instant t_k, there is sufficient computing capacity freed up by the discarded jobs of tasks with criticality less than k over the interval $[\hat{d}, d]$ to schedule τ_i's job to completion by its actual deadline.

We can think of our scheduling algorithm as consisting of 2 phases. In the *pre-processing* phase a schedulability test is applied to the system and if successful, L additional parameters $\delta(k)$ such that $0 < \delta(k) \leq 1$ are determined for each criticality level $1 \leq k \leq L$. This readies the system for the *run-time* phase of the algorithm. During the run-time phase jobs are initially dispatched under the assumption that the system will behave according to level 1-criticality specifications. If this is violated, that is if a job of task τ_i executes for longer than $C_i(1)$ time units but at most $C_i(k)$ time units, or if successive jobs of task τ_i arrive less than $T_i(1)$ but at least $T_i(k)$ time units apart then the system behavior changes to a level k-criticality behavior. In k-criticality behavior currently-active

jobs of tasks with criticality less than k are discarded, and no such jobs are subsequently admitted in the system. Further increase in the criticality level of the system may occur as long $k < L$; if $k = L$ then any further increase in the criticality level is considered erroneous.

We now discuss the phases in greater detail.

4.4.1 The pre-processing phase

During this phase we compute the parameters $\delta(k)$ for each criticality level $1 \leq k \leq L$. These parameters are passed to the run-time phase of our algorithm which is described in Section 4.4.2. If we are able to compute the parameters $\delta(k)$ for each criticality level then the task system is schedulable by our algorithm. However, if we are unable to compute a parameter $\delta(k)$ for some criticality level k then the task system may not be schedulable. Therefore, this phase also serves as a sufficient schedulability test for our algorithm.

Let $\{\delta(1), \ldots, \delta(L)\}$, denote positive real numbers satisfying $0 \leq \delta(k) \leq 1$, where $1 \leq k \leq L$. For each criticality level k we define a corresponding *constrained-deadline* sporadic task system $\tau(k)$ as follows:

$$\tau(k) = \begin{cases} (C_i(k), ((1-\delta(k)) \times T_i(k)), T_i(k)), & \{\forall \tau_i \in \tau | \chi_i = k\} \\ (C_i(k), (\mathbf{min}(\delta(\chi_\mathbf{i}), \mathbf{1} - \delta(\chi_\mathbf{i})) \times T_i(\chi_i)), T_i(k)), & \{\forall \tau_i \in \tau | \chi_i > k\}. \end{cases} \tag{4.8}$$

Let $\tau_i(k)$ represent a task is the task system $\tau(k)$. (From Section 2.2 recall that each constrained-deadline sporadic task $\tau_i(k) \in \tau(k)$ is represented by three parameters:$(C_i(k), D_i(k), T_i(k))$, where $D_i(k) \leq T_i(k)$.)

For each task system $\tau(k)$ starting with $\tau(L)$ and then proceeding in decreasing order of criticality we can perform a binary search to derive the *largest* value of $\delta(k)$ for which the task system $\tau(k)$ is EDF-schedulable. An extension to the *Demand Bound Function* schedulability

test (Baruah et al., 1990) for EDF, which is *pseudo-polynomial with respect to task parameters* can be used to derive the largest value of $\delta(k)$ that ensures that the task system is schedulable under EDF. If we find a value $\delta(k)$ for each criticality level then the task system is schedulable.

The demand bound function $DBF(\tau_i,t)$, bounds the maximum cumulative execution requirement by jobs of a task τ_i that arrive in and have deadlines within any interval of length t. $DBF(\tau_i(k),t)$ for a task $\tau_i(k)$ in Expression 4.8 for scheduling under EDF is as follows:

$$DBF(\tau_i(k),t) = \begin{cases} max(0, \left\lfloor \frac{t-(1-\delta(k))\times T_i(k)}{T_i(k)} \right\rfloor + 1) \times C_i(k), & \text{if } \chi_i = k \\ max(0, \left\lfloor \frac{t-min(\delta(\chi_i),(1-\delta(\chi_i)))\times T_i(\chi_i)}{T_i(k)} \right\rfloor + 1) \times C_i(k), & \text{if } \chi_i > k. \end{cases} \tag{4.9}$$

However, if the system is operating at a criticality level below k, and there is a change to criticality level k, then it can be shown that the demand bound function $DBF(\tau_i(k),t)$ for a task $\tau_i(k)$ in Expression 4.8 with criticality greater than k increases (because of one additional job) as follows:

$$DBF(\tau_i,t) = \begin{cases} max(0, \left\lfloor \frac{t-(1-\delta(k))\times T_i(k)}{T_i(k)} \right\rfloor + 1) \times C_i(k), & \text{if } \chi_i = k \\ max(0, \left\lfloor \frac{t-min(\delta(\chi_i),(1-\delta(\chi_i)))\times T_i(\chi_i)}{T_i(k)} \right\rfloor + \mathbf{2}) \times C_i(k), & \text{if } \chi_i > k. \end{cases} \tag{4.10}$$

We use Equation 4.10 to derive the largest value of $\delta(k)$ that ensures that the task system is EDF-schedulable.

4.4.2 Run-time dispatching

Each criticality level k has a run queue $Q(k)$. A job of a task τ_i with criticality χ_i is queued in queue $Q(\chi_i)$ with deadline equal to its period, and is queued in the queue of every criticality level less than χ_i with deadline equal to $(\delta(\chi_i) \times T_i(\chi_i))$, where $\delta(\chi_i)$ is the parameter computed for the task system $\tau(k)$ described in Expression 4.8 such that $k = \chi_i$. Note that during a criticality change to level k-criticality, if a job of a k-criticality task is active its deadline is pushed back from $(\delta(\chi_i = k) \times T_i(k))$ to $T_i(k)$, and if a job of a task with criticality greater than k is active its deadline is pushed back from $(\delta(\chi_i) \times T_i(\chi_i))$ to $(min(\delta(\chi_i), (1 - \delta(\chi_i))) \times T_i(\chi_i))$. All currently-active jobs of tasks with criticality less than k are discarded. Subsequently, jobs of all k-criticality tasks have a deadline $T_i(k)$ time units after they arrive, and jobs of all tasks with criticality greater than k have a deadline $(\delta(\chi_i) \times T_i(\chi_i))$ time units after they arrive. Jobs of tasks with criticality less than k are not admitted.

We allow for more than one criticality change to occur at run-time. However, after a criticality change occurs the system must reach an *idle instant*, that is all jobs in the system should have finished execution and the system should be idle at some time-point before another criticality change can occur.

Additional rules can be specified to switch to a lower criticality level. This could happen for instance, if the system has been idle for a while. (We will not discuss the process of switching back to a lower criticality level since this concern is application-specific.)

4.4.3 Proof of correctness

We show the correctness of our algorithm in two steps. First, we show that all the k-criticality levels are schedulable. We then show that the algorithm correctly schedules all jobs during any criticality change that may occur during run-time.

Scheduling k-criticality compliant behavior. During the pre-processing phase we choose a parameter $\delta(k)$ for each criticality level k such that the task system $\tau(k)$ described in Expression 4.8

is EDF-schedulable. This ensures that in criticality level k all jobs of tasks with criticality k can execute for at most $C_i(k)$ time units in $D_i(k) = ((1-\delta(k)) \times T_i(k))$ time units, which is at most $T_i(k)$ since $0 \leq \delta(k) \leq 1$. Also, all jobs of tasks with criticality greater than k can execute for $C_i(k)$ time units in $D_i(k) = (min(\delta(\chi_i), 1-\delta(\chi_i)) \times T_i(\chi_i))$ time units, which is at most $T_i(\chi_i)$ and $T_i(\chi_i) \leq T_i(k)$ because $\chi_i > k$. This establishes that all k-criticality behaviors of the system are correctly scheduled by our algorithm.

Scheduling criticality change behavior. Next, consider that a criticality change occurs. Let t_k denote the first time-instant at which the system exhibits k-criticality behavior. Henceforth, all jobs of tasks with criticality less than k are discarded. If a job of a k-criticality task τ_i is active at time t_k its deadline is pushed back from $(\delta(k) \times T_i(k))$ to $T_i(k)$. Thus, it's deadline is at least $((1-\delta(k)) \times T_i(k))$ time units in the future. Subsequent jobs of such a k-criticality task τ_i will have a deadline $T_i(k)$ time units after they arrive. If a job of a task τ_i with criticality greater than k is active at time t_k its deadline is pushed back from $(\delta(\chi_i) \times T_i(\chi_i))$ to $(\min(\delta(\chi_i), 1-\delta(\chi_i)) \times T_i(\chi_i))$ time units in the future. Subsequent jobs of such a task τ_i with criticality greater than k will have a deadline $(\delta(\chi_i) \times T_i(\chi_i))$ time units after they arrive.

It can be shown by an extension to the results in (Baruah et al., 1990) that $DBF(\tau_i(k), t)$, as computed in the Equation 4.10, is the worst-case execution requirement of the jobs of a task $\tau_i(k)$ with criticality at least k in an interval t following a criticality change.

Since each task system $\tau(k)$ such that $1 \leq k \leq L$, is checked for EDF-schedulability in the pre-processing phase of our algorithm we conclude that a criticality change to level k is correctly scheduled by our algorithm. Further, if there is a subsequent criticality change we can use a similar argument to show that the criticality change is correctly scheduled. Note that in our analysis, it is essential that a subsequent criticality change occur only after the system has reached an *idle instant*. If a criticality change occurs before the system has reached an *idle instant*, the system may not be schedulable.

4.5 Conclusion

Mixed-criticality systems are increasingly being implemented upon multiprocessor platforms. We have described and evaluated an algorithm for partitioned scheduling of mixed-criticality implicit-deadline sporadic task systems upon an identical multiprocessor platform. We use an existing EDF based mixed-criticality scheduling algorithm as the uniprocessor scheduling algorithm on each processor. We have also described pragmatic improvements for the algorithm, and compared the improvements by performing schedulability experiments.

The existing EDF based mixed-criticality scheduling algorithm that we use considers that the mixed-criticality task system consists of tasks with two criticality levels denoted as LO and HI, and that the pessimism for HI-criticality tasks is expressed in terms of having a larger worst-case execution time parameter. It is however common to have tasks with more than two criticality levels in a given system, for example the DO-178B standard specifies five criticality levels. Further, the pessimism of higher criticality tasks can be expressed in terms of larger worst-case execution times and shorter periods, that is jobs of higher criticality tasks may arrive at shorter intervals. We have shown how to extend the existing EDF based mixed-criticality scheduling algorithm to incorporate tasks with more than two criticality levels, and express pessimism in the worst-case execution time and period parameters.

CHAPTER 5: LIMITED-PREEMPTION SCHEDULING

In Chapters 3 and 4 we studied partitioned scheduling on multiprocessors, and tasks on each processor were scheduled as per *preemptive* EDF or a scheduling algorithm based on *preemptive* EDF. In this chapter we study a variation of preemptive EDF scheduling in which preemptions are not always allowed. The choice of enabling or disabling preemptions is not a trivial one and many issues have to be considered.

In fully-preemptive scheduling (or simply preemptive scheduling) preemptions are enabled and a higher priority job can preempt a lower priority job at any time. The lower priority job can resume execution after all other higher priority jobs have completed execution. In non-preemptive scheduling preemptions are disabled and a higher priority job may have to wait for a lower priority job to finish executing, before it can start executing. The latter delays the execution of a higher priority job. This is one of the main disadvantages of non-preemptive scheduling.

Run-time overheads, described in Section 2.4.2, are higher in preemptive scheduling when compared to non-preemptive scheduling. Each time a job gets preempted and resumes execution run-time overheads for managing scheduling queues and reloading cache lines are incurred. This makes the worst-case execution cost of a task both larger and less predictable. This in turn makes it harder to estimate the number of preemptions a job may incur during its execution, resulting in inflated worst-case execution costs for tasks under preemptive scheduling. Further, when preemptions are enabled a preemption may be forbidden if a job is executing in a critical section. Thus, non-trivial locking protocols for arbitrating access to shared resources are needed to augment preemptive scheduling. This increases the complexity of implementing preemptive scheduling algorithms and the associated run-time overheads. In contrast, arbitrating access to shared resources is trivial in

non-preemptive scheduling on uniprocessors and requires simple synchronization techniques on multiprocessors. (Preemptions are disabled on a per-processor basis).

An alternative to fully-preemptive scheduling and non-preemptive scheduling is a restricted model of preemptive scheduling referred to as limited-preemptive scheduling. In limited-preemptive scheduling each job can execute preemptively on a processor until it needs to execute non-preemptively, possibly to access a shared resource. One of the objectives of this type of scheduling is to allow non-preemptive access to shared resources while still preserving the schedulability of the system.

Some examples of shared resources are shared memory and network bandwidth; more recently work has been done on incorporating Graphical Processing Units (GPUs) as a shared resource in real-time systems. GPUs are used widely for their ability to speed up graphical computations. General purpose computing on GPUs has allowed GPUs to be used in applications outside of graphics. GPUs can be incorporated in real-time systems as shared processing units and a task can use a GPU or CPU at different times during its execution.

In this chapter our main contribution is a *demand-based* schedulability test for limited-preemption scheduling under the global EDF (GEDF) scheduling algorithm for multiprocessors. This schedulability test was first described in (Chattopadhyay and Baruah, 2014). In addition, we show how to apply this schedulability test to a multiprocessor, multi-GPU system. In such systems the execution of a task on a GPU is non-preemptive. A task can execute preemptively on the processor and then request access to a GPU. After a request is made, one option is for the task to busy-wait non-preemptively on the processor until its non-preemptive execution on the GPU is complete. This can be thought of as a limited-preemption scheduling problem.

5.1 System Model

We consider a sporadic task system $\tau = \{\tau_1, \ldots, \tau_n\}$, with n constrained-deadline sporadic tasks $\tau_i, 1 \leq i \leq n$. In Section 2.2, we described the traditional constrained-deadline sporadic task model in which each sporadic task $\tau_i = (C_i, D_i, T_i)$ is characterized by a worst-case execution time C_i, a

relative deadline D_i, and a minimum inter-arrival separation period parameter T_i. Such a sporadic task generates a potentially infinite sequence of jobs with successive job arrivals separated by at least T_i time units. Each job has a worst-case execution requirement equal to C_i, that is fully-preemptive, and has an *absolute deadline* that occurs D_i time units after its arrival time. The utilization U_i of task τ_i is $\frac{C_i}{T_i}$.

In this chapter each sporadic task $\tau_i = (C_i, L_i, D_i, T_i)$ has an additional parameter L_i that represents the total length for which a job of a task may need to execute non-preemptively. The total execution requirement of such a task is $C_i + L_i$, where C_i is fully-preemptive and L_i is non-preemptive. We assume that L_i may be non-contiguous. For each task τ_i, we let the length of its non-preemptive execution time be represented as an ordered set $\{L_{i1}, L_{i2}, \ldots L_{ik}\}$, where L_{ij} ($j \in \{1, 2, \ldots, k\}$) represents the maximum length of the j^{th} longest contiguous non-preemptive execution of task τ_i, and $max\{L_{ij}\} = L_{i1}$. L_i is the sum of all such non-preemptive execution lengths: $L_i = \sum_{j=1}^{k} L_{ij}$. Such a sporadic task can be fully represented as, $\tau_i = (C_i, \{L_{i1}, L_{i2}, \ldots L_{ik}\}, D_i, T_i)$. Further, we assume that the preemptive and the non-preemptive execution of a task can be interleaved in any manner, i.e. we assume that we do not know the start and end points of the non-preemptive execution L_{ij} of a task.

The D_i and T_i parameters denote the same task properties as in the traditional model. The utilization of a task U_i is $\frac{C_i + L_i}{T_i}$. We henceforth refer to such tasks as limited-preemption sporadic tasks and n such tasks make up a limited-preemption sporadic task system τ. We denote $U(\tau) = \sum_{i=1}^{n} U_i$ as the total utilization.

A traditional sporadic task system is said to be a constrained-deadline sporadic task system if for each task $\tau_i \in \tau$, $D_i \leq T_i$, and an implicit-deadline sporadic task system if $D_i = T_i$. This definition is applicable to limited-preemption sporadic task systems as well. In this chapter, we restrict our attention to constrained-deadline and implicit-deadline limited-preemption sporadic task systems.

The computing platform consists of a multiprocessor with m identical unit-capacity processors. The scheduling algorithm is GEDF (global EDF). As discussed in Section 2.4.2, in GEDF scheduling

the m jobs with the earliest deadline are scheduled on the m processors. Intra-job migrations are allowed.

We derive a schedulability test for constrained-deadline and implicit-deadline limited-preemption sporadic task systems for the given computing platform. In the derivation of the schedulability test we use the concept of *Demand Bound Function*. This concept was introduced in Section 4.4.3 (Equation 4.9) with respect to mixed-criticality tasks. We now define the demand bound function of a limited-preemption sporadic task.

Definition 5.1. By an extension to the results in (Baruah et al., 1990), $DBF(\tau_i, t)$ of a limited-preemption sporadic task τ_i over an interval t is as follows:

$$DBF(\tau_i, t) = max(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1)(C_i + L_i)). \tag{5.1}$$

5.2 Related Work

A recent survey (Buttazzo et al., 2013) discusses and compares existing approaches for limited-preemption scheduling. The following approaches have been proposed in the literature: *Preemption thresholds scheduling, Deferred preemptions scheduling*, and *Fixed Preemption Points*. The approach that we adopt in this work is deferred preemptions scheduling and is described below. Please refer (Buttazzo et al., 2013, Section 2) for a description of the other approaches.

Deferred preemptions scheduling was first introduced in (Baruah, 2005) under EDF scheduling. In this approach, each task τ_i can execute non-preemptively for a total length of say, q_i. It has been explained in (Buttazzo et al., 2013) that there are two ways in which non-preemptive regions can be implemented, *floating, and activation-triggered.*

A *floating* non-preemptive region can be defined by the programmer by inserting specific primitives in the task code that disable and enable preemption. However, the start and end time of this region is not specified. Thus, from an analysis perspective the non-preemptive region can be thought as "floating" in the code with a duration not exceeding some constant q_i.

An *activation-triggered* non-preemptive region can be triggered by the arrival of a higher priority job say at time t and programmed by a timer to last exactly q_i time units, unless the currently executing job finishes earlier, after which preemption is enabled. Any further arrivals do not postpone the time $t + q_i$ at which preemptions are enabled. Once a preemption takes place at or after time $t + q_i$, a new higher-priority job can trigger another non-preemptive region.

Schedulability analysis in (Baruah, 2005) assumes floating non-preemptive regions and computes the longest non-preemptive execution q_i for each task τ_i without compromising the feasibility of the system. Analysis in (Bertogna and Baruah, 2010) assumes the activation-triggered model and computes a function $Q(t)$ that takes as input the time to the deadline of the executing job, and provides the amount of time for which such a job could execute non-preemptively when a new high-priority job arrives without compromising the feasibility of the system.

The analysis in (Baruah, 2005; Bertogna and Baruah, 2010; Short, 2011) was derived for uniprocessor EDF. The analysis presented in (Fisher and Baruah, 2006) was derived for partitioned EDF assuming floating non-preemptive floating regions. We are unaware of any demand-based schedulability analysis under GEDF for multiprocessors for the system model described in Section 5.1.

In this chapter we present a schedulability analysis for multiprocessor GEDF scheduling for the limited-preemption sporadic task model described in Section 5.1. Our model assumes floating non-preemptive regions, and in our analysis we use the demand bound function to determine whether, given the task parameters (C_i, L_i, T_i, D_i) for each task τ_i, the system is schedulable.

5.3 Schedulability Test

The schedulability test described here extends the schedulability test described in (Baruah, 2007) for fully-preemptive sporadic task systems to limited-preemption sporadic task systems. Note that if $L_i = 0$ for each task $\tau_i = (C_i, L_i, D_i, T_i)$ in τ, then the task system is a fully-preemptive sporadic task system. In the following discussion a task (task system) is assumed to be a limited-preemption sporadic task (task system) unless mentioned otherwise.

The general framework of how we derive the schedulability test is the same as described in (Baruah, 2007). We consider each task τ_k separately; when considering a specific τ_k, we identify sufficient conditions for ensuring that τ_k cannot miss any deadlines. To ensure that no deadlines are missed by any task in τ, these conditions are checked for each of the n tasks, $\tau_1, \tau_2, \ldots, \tau_n$.

Consider any legal sequence of job requests of task system τ for which GEDF misses a deadline. Suppose that a job of task τ_k is the one to first miss a deadline, and that this deadline miss occurs at time-instant t_d. Let t_a denote this job's arrival time: $t_a = t_d - D_k$.

Definition 5.2. Let t_0 denote the latest time-instant at or before t_a at which at least one processor has finished executing all jobs that arrive *before* t_0 and have *absolute deadlines* at most t_d.

Let $t = t_d - t_0$ and $A_k = t_a - t_0$. (Consequently, t is also $A_k + D_k$).

Note that the definition of t_0 is the same as that in (Baruah, 2007). However, in (Baruah, 2007) only jobs with absolute deadlines at most t_d are considered in the analysis. (This is valid in the analysis for fully-preemptive systems since jobs with absolute deadlines greater than t_d do not contribute to the deadline miss at time t_d.) In limited-preemptive systems a job of a task τ_i having absolute deadline greater than t_d can contribute to the deadline miss at time t_d. In the following Lemmas it will become clear that a job of a task τ_i with absolute deadline greater than t_d does not start executing in the interval $[t_0, t_a)$ but it can start executing in the interval $[t_a, t_d)$ and cause a deadline miss at time t_d.

Lemma 5.1. *A job of task τ_i with absolute deadline greater than t_d does not start executing in the interval $[t_0, t_a)$.*

Proof. Let us assume that a job of task τ_i with absolute deadline greater than t_d starts executing in the interval $[t_0, t_a)$. Let this job of task τ_i start executing at time-instant $t_0 + \delta$, $0 \leq \delta < (t_a - t_0)$. As per GEDF this implies that at least on one processor all jobs with absolute deadline at most t_d finished executing by time-instant $t_0 + \delta$ and no job with absolute deadline at most t_d arrived at $t_0 + \delta$.

This makes $t_0 + \delta + \varepsilon$, $\varepsilon \simeq 0$, the latest time-instant $\leq t_a$ at which at least one processor has finished executing all jobs that arrived before $t_0 + \delta + \varepsilon$ and with absolute deadline at most t_d. Since $t_0 + \delta + \varepsilon > t_0$ and by Definition 5.2 of time-instant t_0, we have a contradiction to our assumption. The lemma follows. □

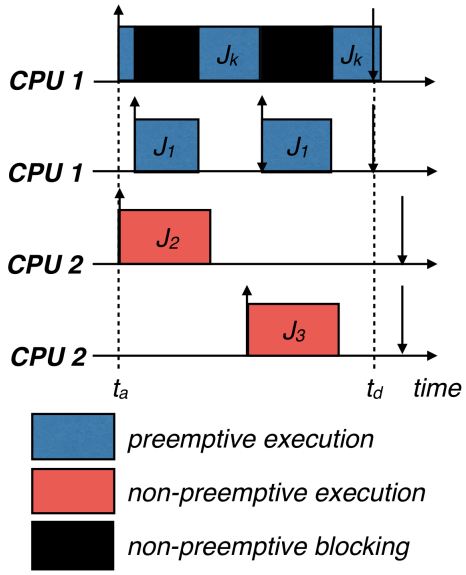


Figure 5.1: The schedule generated by GEDF on two processors, CPU1 and CPU2, for jobs of tasks τ_1, τ_2, τ_3 and τ_k is shown. Note that jobs J_k and J_2 are released at time t_a and job J_1 is released immediately after time t_a. Jobs J_2 and J_3 have an absolute deadline greater than t_d and cause job J_k to experience non-preemptive blocking which leads to job J_k missing it's deadline at time t_d.

Lemma 5.2. *A job of task τ_i with absolute deadline greater than t_d can start executing in the interval $[t_a, t_d)$.*

Proof. The scenario shown in Figure 5.1 can be observed under GEDF when non-preemptive execution is permitted. This scenario was first shown in (Block et al., 2007). Since we consider implicit-deadline and constrained-deadline tasks, only one job of task τ_i with absolute deadline greater than t_d can execute non-preemptively, for at most L_i time units, and contribute to the deadline miss of task τ_k at time t_d. Note that the preemptive execution of jobs with absolute deadline greater than t_d does not contribute to the deadline miss at time t_d. □

We now identify conditions necessary for a deadline miss to occur, that is for τ_k's job to execute for strictly less than $C_k + L_k$ time units over $[t_a, t_d)$. In order for τ_k's job to execute for strictly less than $C_k + L_k$ time units over $[t_a, t_d)$, it is necessary that all m processors execute jobs other than τ_k's for strictly more than $D_k - (C_k + L_k)$ time units over $[t_a, t_d)$. Let us denote by Γ_k a collection of intervals, not necessarily contiguous, of cumulative length exactly $D_k - (C_k + L_k)$ over $[t_a, t_d)$, during which all m processors are executing jobs other than τ_k's job in this GEDF schedule.

For each task $\tau_i, 1 \leq i \leq n$, let $I(\tau_i)$ denote the contribution of τ_i to the work done in this GEDF schedule during $[t_0, t_a) \cup \Gamma_k$. In order for a deadline miss to occur, it is necessary that the total amount of work that executes over $[t_0, t_a) \cup \Gamma_k$ satisfy the following condition:

$$\sum_{\tau_i \in \tau} I(\tau_i) > m \times (A_k + D_k - (C_k + L_k)). \tag{5.2}$$

This follows from the observation that all m processors are, by definition, completely busy executing this work over the A_k time units in the interval $[t_0, t_a)$, as well as the intervals in Γ_k of total length $D_k - (C_k + L_k)$. Note that the total length of the intervals in $[t_0, t_a) \cup \Gamma_k$ is equal to $(A_k + D_k - (C_k + L_k))$.

Let us say that τ_i has a *carry-in job* in this GEDF schedule if there is a job of τ_i that arrives before t_0 and has not completed execution by t_0. In the following discussion we compute upper bounds on $I(\tau_i)$ if τ_i has no carry-in job (this is denoted as $I_1(\tau_i)$)), or if it does (denoted as $I_2(\tau_i)$). We separately compute $B(\tau_i)$ which is the maximum non-preemptive blocking due to task τ_i.

Computing $I_1(\tau_i)$. Let us consider the situation when all jobs of τ_i arrive in the interval $[t_0, t_d)$ and as a result task τ_i has no carry-in work. $I_1(\tau_i)$ is the total work contributed by all jobs of τ_i that arrive in the interval $[t_0, t_d)$ and have absolute deadlines at most t_d. (Later, we compute $B(\tau_i)$ to determine the maximum non-preemptive blocking due to a job of task τ_i with absolute deadline greater than t_d.)

Let us first consider a task τ_i such that $i \neq k$. In this case, it follows from Definition 5.1 of the demand bound function that the total work is at most $DBF(\tau_i, A_k + D_k)$. Furthermore, this total

contribution cannot exceed the total length of the intervals in $[t_0, t_a) \cup \Gamma_k$. Hence, the contribution of τ_i to the total work that must be done by GEDF over $[t_0, t_a) \cup \Gamma_k$ is at most

$$min(DBF(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)). \tag{5.3}$$

Now consider the case $i = k$. In this case, the job of τ_k arriving at time-instant t_a does not contribute to the work that must be done by GEDF over $[t_0, t_a) \cup \Gamma_k$, hence its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_0, t_a)$ i.e., A_k.

Putting these pieces together we get the following bound on the contribution of τ_i to the total work that must be done by GEDF over $[t_0, t_a) \cup \Gamma_k$:

$$I_1(\tau_i) = \begin{cases} min(DBF(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)), & \text{if } i \neq k \\ min(DBF(\tau_i, A_k + D_k) - (C_k + L_k), A_k), & \text{if } i = k. \end{cases} \tag{5.4}$$

Computing $I_2(\tau_i)$. Let us now consider the situation when τ_i arrives before t_0, and hence potentially carries in some work in the interval $[t_0, t_d)$. It was shown in (Bertogna et al., 2005) that the total work of a sporadic task τ_i with carry-in work can be upper-bounded by considering the scenario in which some job of τ_i has a deadline at t_d, and all jobs of τ_i execute at the very end of their scheduling windows.

Let the demand bound function $DBF'(\tau_i, t)$ denote the *maximum* amount of work that can be contributed by τ_i with carry-in work over a contiguous interval of length t. The definition of $DBF'(\tau_i, t)$ in (Baruah, 2007) for sporadic tasks can be extended to limited-preemption sporadic tasks as follows:

$$DBF'(\tau_i,t) = \left\lfloor \frac{t}{T_i} \right\rfloor \times (C_i + L_i) + min(C_i + L_i, t \bmod T_i). \tag{5.5}$$

In computing τ_i's contribution to the total amount of work that must execute over $[t_0,t_a) \cup \Gamma_k$, let us first consider $i \neq k$. In this case, it follows from the definition of demand bound function DBF' that the upper bound on the amount of work contributed by task τ_i is $DBF'(\tau_i, A_k + D_k)$. Furthermore, this contribution cannot exceed the total length of the intervals in $[t_0,t_a) \cup \Gamma_k$. Hence, the contribution of τ_i to the total work that must be done by GEDF over $[t_0,t_a) \cup \Gamma_k$ is at most:

$$min(DBF'(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)). \tag{5.6}$$

Now consider the case $i = k$. In this case, we know that a job of task τ_k that arrives at time t_a has a deadline at t_d and does not contribute to the work that must be done by GEDF over $[t_0,t_a) \cup \Gamma_k$, hence its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_0,t_a)$ i.e., A_k.

From the discussion above we get the following bound on the contribution of τ_i to the total work that must be done by GEDF over $[t_0,t_a) \cup \Gamma_k$:

$$I_2(\tau_i) = \begin{cases} min(DBF'(\tau_i, A_k + D_k), A_k + D_k - (C_k + L_k)), & \text{if } i \neq k \\ min(DBF'(\tau_i, A_k + D_k) - (C_k + L_k), A_k), & \text{if } i = k. \end{cases} \tag{5.7}$$

Computing $B(\tau_i)$. The maximum non-preemptive blocking due to task τ_i over $[t_0,t_a) \cup \Gamma_k$ is caused by a job of task τ_i with absolute deadline greater than t_d. Since we consider constrained-deadline and implicit-deadline tasks there can be only one such job of task τ_i in the intervals in

$[t_0, t_a) \cup \Gamma_k$. Note that if this job arrives before t_0 then it can be considered a carry-in job and $I_2(\tau_i)$ upper bounds its contribution. Therefore, in computing $B(\tau_i)$ we only need to account for the maximum non-preemptive blocking due to a job of task τ_i that arrives in the interval $[t_0, t_d)$.

We know from Lemmas 5.1 and 5.2 that a job of task τ_i that arrives in the interval $[t_0, t_d)$ with an absolute deadline greater than t_d can start executing only in the interval $[t_a, t_d)$, of length D_k. Therefore, the maximum non-preemptive blocking due to task τ_i is as follows:

$$B(\tau_i) = \begin{cases} min(L_i, D_k), & \text{if } i \neq k \\ 0, & \text{if } i = k. \end{cases} \tag{5.8}$$

Putting the pieces together. Let us first compute the total amount of carry-in work over the intervals in $[t_0, t_a) \cup \Gamma_k$. By Definition 5.2 of t_0, at most m tasks have not completed execution at time-instant t_0. Consequently, at most m tasks can contribute an amount $I_2(\tau_i)$ and the remaining $(n-m)$ tasks must contribute $I_1(\tau_i)$. However, as per Definition 5.2, on at least one processor all tasks with absolute deadline at most t_d have completed execution before t_0. Thus, on at least one processor the carry-in work is contributed by a job of a task τ_j with absolute deadline greater than t_d. Further, it can be shown that such a task τ_j has a deadline D_j that satisfies $D_j > t$, and that the maximum amount of carry-in work that task τ_j can contribute is the length of its non-preemptive execution L_j. Hence, the total amount of carry-in work can be written as:

$$\sum_{(m-1)max} I_2(\tau_i) + max\{L_j\}_{D_j > t}, \tag{5.9}$$

where $\sum_{(m-1)max} I_2(\tau_i)$ is the sum of the $(m-1)$ largest values of $I_2(\tau_i)$, $1 \leq i \leq n$.

Note that Equation 5.9 upper bounds the maximum carry-in work and the maximum non-preemptive blocking due to jobs that arrive before t_0.

We now compute the maximum non-preemptive blocking due to jobs that arrive in the interval $[t_0, t_d)$. From Equation 5.8 we know that the total non-preemptive blocking caused by such jobs can be expressed as $\sum_{\tau_i \in \tau} B(\tau_i)$. However, this can be pessimistic for the following reason.

We know that a job of task τ_k arrives at time t_a and has a deadline at t_d, therefore on at least one processor jobs with absolute deadline greater than t_d will not start executing in the interval $[t_a, t_d)$ until the job of task τ_k has met its deadline. However, as per our assumption the job of task τ_k misses its deadline. Therefore, the total non-preemptive blocking is at most $(m-1) \times D_k$. Let B_n be the maximum non-preemptive blocking due to jobs of all tasks that arrive in the interval $[t_0, t_d)$. B_n is as follows:

$$B_n = min(\sum_{\tau_i \in \tau} B(\tau_i), (m-1) \times D_k) \tag{5.10}$$

Let us denote by $I_{Diff}(\tau_i)$ the difference between $I_2(\tau_i)$ and $I_1(\tau_i)$:

$$I_{Diff}(\tau_i) = I_2(\tau_i) - I_1(\tau_i). \tag{5.11}$$

Condition 5.2 may be re-written as follows:

$$\begin{aligned} \sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{(m-1)max} I_{Diff}(\tau_i) + max\{L_j\}_{D_j > t} + B_n \\ > m \times (A_k + D_k - (C_k + L_k)). \end{aligned} \tag{5.12}$$

Observe that all the terms in Condition 5.12 above are completely defined for a given task system, once a value is chosen for A_k. Hence for a deadline miss of task τ_k to occur, there must

exist some A_k such that Condition 5.12 is satisfied. Conversely, in order for all deadlines of task τ_k to be met it is sufficient that Condition 5.12 be violated for all values of A_k. Theorem 5.1 follows immediately:

Theorem 5.1. *Task system τ is GEDF-schedulable upon m unit-capacity processors if for all tasks $\tau_k \in \tau$ and all $A_k \geq 0$,*

$$\sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{(m-1)max} I_{Diff}(\tau_i) + max\{L_j\}_{D_j > t} + B_n \leq m \times (A_k + D_k - (C_k + L_k)) \tag{5.13}$$

where $I_1(\tau_i)$, $I_{Diff}(\tau_i)$, and B_n are as defined in Equations 5.4, 5.11, and 5.10 respectively.

5.3.1 Properties

Run-time Complexity. For a given task τ_k and A_k, it is easy to see that Condition 5.13 can be evaluated in time linear in n, the number of tasks in the task system:

- Compute $I_1(\tau_i), I_2(\tau_i), B(\tau_i)$ and $I_{Diff}(\tau_i)$ for each task τ_i - total time is $O(n)$.
- Use linear-time selection (Blum et al., 1973) on $\{I_{Diff}(\tau_1), I_{Diff}(\tau_2), \ldots, I_{Diff}(\tau_n)\}$ to determine the $(m-1)$ tasks that contribute to the second sum on the LHS.
- Compute $max\{L_j\}_{D_j > t}$ and B_n - total time is $O(n)$.

We now determine the values of A_k. First, we derive the range for the values of A_k and then determine the individual values of A_k for which Condition 5.13 must be verified.

Theorem 5.2. *If Condition 5.13 is to be violated for any A_k, then it is violated for some A_k satisfying the condition below:*

$$A_k \leq \frac{S_\Sigma - D_k(m - U(\tau)) + \sum_i ((T_i - D_i)U_i + L_i) + m(C_k + L_k)}{m - U(\tau)} \tag{5.14}$$

where S_Σ denotes the sum of the m largest $(C_i + L_i)$.

Proof. It can be seen that $I_1(\tau_i) \leq DBF(\tau_i, A_k + D_k)$, $I_2(\tau_i) \leq DBF(\tau_i, A_k + D_k) + (C_i + L_i)$, and $B_n \leq \sum_{\tau_i \in \tau} L_i$. From this, it can be shown that the LHS of Condition (5.13) is $\leq S_\Sigma + \sum_{\tau_i \in \tau} DBF((\tau_i, A_k + D_k) + L_i)$.

For this to exceed the RHS of Condition (5.13), it is necessary that:

$$
\begin{aligned}
& S_\Sigma + \sum_{\tau_i \in \tau} (DBF(\tau_i, A_k + D_k) + L_i) > m(A_k + D_k - (C_k + L_k)) \\
& \rightarrow S_\Sigma + (A_k + D_k)U(\tau) + \sum_i ((T_i - D_i)U_i + L_i) \\
& \quad > m(A_k + D_k - (C_k + L_k)) \\
& \quad \text{(bounding DBF using the technique in (Baruah et al., 1990))} \\
& \equiv S_\Sigma + D_k U(\tau) + \sum_i ((T_i - D_i)U_i + L_i) - m(D_k - (C_k + L_k)) \\
& \quad > A_k(m - U(\tau)) \\
& \equiv A_k \leq \\
& \quad \frac{S_\Sigma - D_k(m - U(\tau)) + \sum_i ((T_i - D_i)U_i + L_i) + m(C_k + L_k)}{m - U(\tau)}.
\end{aligned}
$$

The theorem follows. □

Further, we only need to consider the non-negative values of A_k. It can also be shown that Condition (5.13) need only be tested at those values of A_k at which $DBF(\tau_i, A_k + D_k)$ changes for some τ_i. To be specific, it is shown in (Brandenburg, 2011, p. 82) that it is sufficient to test only those values of A_k that satisfy:

$$A_k = D_i - D_k + j \times T_i, \tag{5.15}$$

for all $\tau_i \in \tau$ and all $j \in \{0, 1, 2, \ldots\}$ that satisfy Condition 5.14. The bound on the maximum A_k grows exponentially as $m - U(\tau)$ approaches 0. However, for values of $U(\tau)$ bounded by a constant strictly less than the number of processors m the following property holds:

Property 5.1. *The condition in Theorem 5.1 can be tested in time pseudo-polynomial in the task parameters, for all task systems τ for which $U(\tau)$ is bounded by a constant strictly less than the number of processors m.*

Sufficient/Necessary. Theorem 5.1 is an extension to the schedulability test in (Baruah, 2007). The latter was derived for fully-preemptive sporadic task systems. The schedulability test in (Baruah, 2007) has been shown to be a generalization of the uniprocessor schedulability test in (Baruah et al., 1990). It is sufficient and necessary when $m = 1$ and sufficient but not necessary when $m > 1$.

A limited-preemption sporadic task system τ is a fully-preemptive sporadic task system if $L_i = 0$ for all tasks $\tau_i \in \tau$. It can be shown that, if $L_i = 0$ for all tasks $\tau_i \in \tau$ then the schedulability test in Theorem 5.1 reduces to the schedulability test in (Baruah, 2007). This leads to the following property.

Property 5.2. *For fully-preemptive sporadic task systems the schedulability test in Theorem 5.1 is sufficient and necessary when $m = 1$ and sufficient but not necessary when $m > 1$.*

We now show that the above property continues to hold for limited-preemption sporadic task systems, such that $L_i > 0$ for some task $\tau_i \in \tau$.

Lemma 5.3. *For a task τ_k and for $m = 1$ processor, Condition (5.16) determines whether the* exact *processor demand over an interval t of length $A_k + D_k$ is at most the length of the interval.*

$$\begin{aligned} (\sum_{\tau_i \in \tau} I_1(\tau_i) + (C_k + L_k)) + max\{L_i\}_{D_i > t} \\ \leq A_k + D_k \end{aligned} \tag{5.16}$$

Proof. As per Equation (5.4) the first term in the LHS of Condition (5.16) is equal to the processor demand in an interval t by jobs arriving in this interval and having deadlines within this interval. By Definition 5.2 of t_0 no task τ_i with $D_i \leq t$ can be active at time-instant t_0 and one or many tasks with $D_i > t$ can be active at time-instant t_0. Under EDF and $m = 1$ only one such task can execute non-preemptively in the interval t. The second term in the LHS of Condition (5.16) accounts for the

maximum amount of non-preemptive blocking possible due to limited-preemptivity. Thus, the LHS of Condition (5.16) gives the exact processor demand over some interval t. □

Observe that Condition (5.13) reduces to Condition (5.16) for $m = 1$ by adding $(C_k + L_k)$ to both LHS and RHS. Thus, by Lemma 5.3, Condition (5.13) determines whether the exact processor demand over an interval t is at most the length of the interval for $m = 1$. For $m > 1$, Condition (5.13) *upper bounds* the amount of work carried in on $m-1$ processors (Refer Equation 5.7). This leads to the following property.

Property 5.3. *For limited-preemption sporadic task systems, such that $L_i > 0$ for some task $\tau_i \in \tau$, the schedulability test in Theorem 5.1 is sufficient and necessary when $m = 1$ and sufficient but not necessary when $m > 1$.*

5.4 Multi-GPU System Model

We have described a schedulability test in Theorem 5.1 and derived and discussed some of its properties. In Section 5.5 we show how Theorem 5.1 can be used as a schedulability test for a multiprocessor multi-GPU system. First, we describe the multi-GPU system model.

A job of a task running on a processor can initiate execution on a GPU. Several aspects of GPU program execution are described in (Elliott et al., 2013). A GPU has an execution engine (EE) and one or two DMA copy engines (CEs). A copy engine transmits data between system memory and GPU memory and an execution engine performs some computation on a given data. A possible sequence of events when a job executes on a GPU is described in (Elliott et al., 2013) and is as follows. First, the copy engine copies data from the system memory to the GPU memory, followed by the computation on the execution engine. Finally, the copy engine copies the results from the GPU memory back to the system memory. Further, GPU operations on the various engines are *non-preemptive*.

Let us assume that each task $\tau_i \in \tau$, makes k requests to the GPU represented as an ordered set, $\{G_{i1}, G_{i2}, \ldots G_{ik}\}$, where $G_{ij} > 0$, $j \in \{1, 2, \ldots, k\}$, represents the j^{th} longest GPU execution.

Each request G_{ij} can either be a request to a copy engine, execution engine, or a combination of requests to the copy engine and execution engine. Let G_i be the sum of the execution length of all the GPU requests a task makes: $G_i = \sum_{j=1}^{k} G_{ij}$. If a task does not make any GPU requests, $G_i = 0$.

Once a job running on a processor initiates execution on a GPU it can either *self-suspend* or it can *busy-wait* on the processor until the GPU execution is complete. When a job self-suspends, the processor is available for other jobs to execute. This is preferable because the job wastes processor cycles when busy-waiting. However, busy-waiting benefits from lower overheads (compared to the cost of suspending and resuming tasks). Therefore, busy-waiting is preferable only if for all tasks the non-preemptive critical section on the GPU is short. Empirical results obtained in (Brandenburg, 2011, Chapter 7) show that busy-waiting implemented as spin-based locks is useful if a task uses a resource for at most a few microseconds.

In the case of busy-waiting, a job can busy-wait *preemptively*, that is it busy-waits until a job with a higher priority preempts it on the CPU while it continues to execute non-preemptively on the GPU. This is different from self-suspension only because a lower or equal priority job can start executing on the CPU after a job self-suspends.

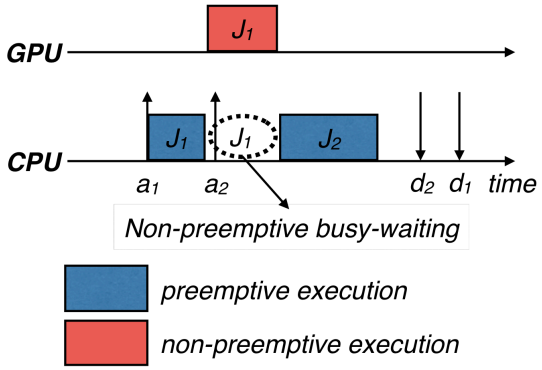


Figure 5.2: Scheduling scenario with $m = 1$ and $g = 1$ with non-preemptive busy-waiting.

An alternative to preemptive busy-waiting is *non-preemptive* busy-waiting. In this case a job busy-waits non-preemptively on a processor until it completes its GPU execution. This is illustrated with the help of Figure 5.2. In Figure 5.2, job J_1 arrives at time a_1 and has a deadline at time d_1, and job J_2 arrives at time a_2 and has a deadline at time d_2. Job J_1 starts executing on the GPU just

before time a_2 while busy-waiting non-preemptively on the CPU. This causes job J_2 with a shorter deadline, thus higher priority, to wait for job J_1 to complete its GPU execution before it can preempt it and start executing on the CPU. Non-preemptive busy-waiting has the least run-time overheads when compared to preemptive busy-waiting and self-suspensions.

Our analysis focuses on the multi-GPU system model with non-preemptive busy-waiting. Tasks under this system model can be modeled as limited-preemption sporadic tasks. For each limited-preemption sporadic task $\tau_i \in \tau$, L_{ij} is set equal to the length of the j^{th} longest non-preemptive *critical section* on a GPU.

In our system we use a simple synchronization approach (a version of the locking protocol described in (Block et al., 2007) for non-nested, short resource requests) to control access to the GPUs. For each GPU we assume there is a spin-lock controlling access to it and for each spin-lock there is a corresponding FIFO-ordered wait queue. If a job that requests for a GPU can acquire any spin-lock it can access the GPU protected by that spin-lock, otherwise it is assigned to a wait queue. A *shortest queue* mechanism is used to determine which wait queue a job is assigned to. Once a job is assigned to a wait queue, it waits on this queue until it can acquire the corresponding spin-lock. For a task τ_i the length of its non-preemptive execution L_{ij} is equal to the sum of the execution requirement on a GPU G_{ij}, and the amount of time it must wait to access a GPU. The latter is computed in Section 5.5.

We assume that all GPUs are identical, that is the execution requirement G_{ij} of a task is the same irrespective of the GPU on which it executes. The number of GPUs is denoted by g. The number of identical, unit-capacity processors continues to be denoted by m.

5.4.1 Prior GPU Analysis

Several GPU management frameworks have been designed and implemented including Time-Graph (Kato et al., 2011b), RGEM (Kato et al., 2011a), Gdev (Kato et al., 2012). Analysis of the RGEM framework includes blocking analysis that is incorporated into classical fixed-priority scheduling response-time analysis for multiprocessors. Elliott et al. designed and implemented

GPUSync (Elliott et al., 2013). In (Elliott and Anderson, 2013) a blocking analysis for a *k-exclusion* locking protocol for globally-scheduled job-level static-priority systems for self-suspending sporadic tasks has been described. The term *k-exclusion* means that there are k copies of some resource, for example GPUs. Recent analysis in (Kim et al., 2013) provides response-time analysis for self-suspending sporadic tasks under rate monotonic scheduling for multiprocessors. Blocking analysis is done using a linear programming technique. In (Cong and Anderson, 2013) a schedulability test for self-suspending tasks under GEDF scheduling is described. This work was not aimed at a GPU platform but can be extended to GPUs.

In our analysis we assume GEDF scheduling. We present a schedulability test for the non-preemptive busy-waiting multi-GPU system model using the results of the schedulability analysis obtained in Theorem 5.1.

5.5 Multi-GPU Schedulability Test

Given the execution length of each GPU request G_{ij} of a task τ_i we first need to determine the length of the non-preemptive execution L_{ij}. For this we have to determine the amount of time a task τ_i may have to wait to access one of the g GPUs. This can be computed by upper bounding the number of GPU requests at any time. The following Lemmas follow directly from the results obtained in (Block et al., 2007).

Lemma 5.4. *There can be at most m GPU requests at any time, one per processor.*

Proof. A job of a task can request for a GPU only when it is executing on some processor. If a job executing on a processor requests for a GPU then it busy-waits non-preemptively until the request is serviced by the GPU. Thus, no other job of any task can execute on this processor and as a result no other GPU request can be made from this processor. Since there are m processors there can be at most m GPU requests at any time. □

Lemma 5.5. *The length of any wait queue is at most* $\left\lceil \frac{m}{g} \right\rceil - 1$.

Proof. From Lemma 5.4 there can be at most m GPU requests at any time. If a shortest queue mechanism is used to distribute these requests across the g wait queues, one for each GPU, then the number of requests on each wait queue is $\left\lceil \frac{m}{g} \right\rceil$. Of these requests one is satisfied by the GPU. Therefore, the length of a wait queue is at most $\left\lceil \frac{m}{g} \right\rceil - 1$. Note that when $g = m$ the length of any wait queue is 0. □

It has been shown in (Wieder and Brandenburg, 2013) that FIFO-ordered spin-locks offer strong progress guarantees and is an effective mechanism for non-preemptive busy-waiting.

Let W_{ij} be the amount of time a job of a task τ_i has to wait upon making the j^{th} request to the GPU.

Theorem 5.3. *The schedulability test in Theorem 5.1 can be applied to the system model under consideration if for each task τ_i and j^{th} GPU request G_{ij}, $L_{ij} = G_{ij} + W_{ij}$, where:*

$$W_{ij} = \sum_{\left(\lceil \frac{m}{g} \rceil - 1\right) max} G_{rs},$$
$$r \in \{1, \ldots, i-1, i+1, \ldots, n\}, s \in \{1, \ldots, k\}. \tag{5.17}$$

Proof. For a task τ_i the length of its non-preemptive execution L_{ij} is equal to the sum of the execution time of its j^{th} GPU request G_{ij}, and the amount of time it must wait on a wait queue W_{ij}. From Lemma 5.5, the length of any wait queue is at most $\left\lceil \frac{m}{g} \right\rceil - 1$. Since we consider implicit-deadline and constrained-deadline task systems, two jobs of the same task cannot be on any of the g wait queues at the same time. Thus, Equation (5.17) upper bounds the term W_{ij}. □

Note that if $G_i = 0$, then a task τ_i does not make any GPU requests. Therefore, it does not have to wait to access the GPU and $L_i = 0$. Also, when $g = m$, $L_{ij} = G_{ij}$ in the multi-GPU system model with non-preemptive busy-waiting.

With non-preemptive busy-waiting we do not get any analytical benefits when $g > m$. However, we can get analytical benefits when $g > m$ for the self-suspending and preemptive busy-waiting system models discussed in Section 5.4. Also, in our system model we assume that the GPU can execute only one task at a time. However, the GPU copy engine (CE) and GPU execution engine (EE) can in fact non-preemptively execute jobs of two different tasks at the same time (see Figure 5.3). This parallelism can be exploited under non-preemptive busy waiting when $g < m$, and in the self-suspending and preemptive busy-waiting system model. We leave this as future work.

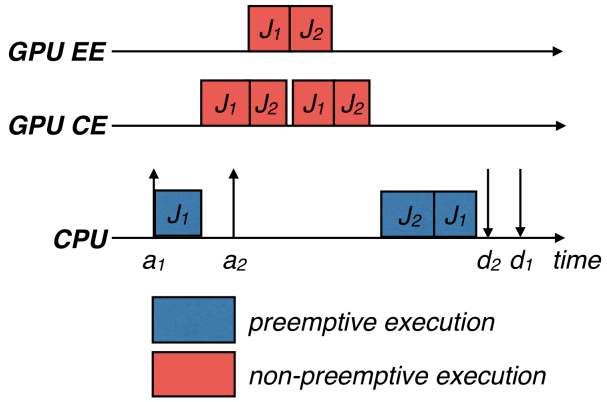


Figure 5.3: Scheduling scenario for m = 1 and g = 1 under preemptive busy-waiting. Jobs J_1 and J_2 execute in parallel on the GPU CE and GPU EE effectively reducing the total time spent executing on the GPU.

5.6 Experimental Evaluation

We perform experiments to determine the effectiveness of our schedulability test in the context of the multiprocessor, multi-GPU system model. We randomly generated task sets and determined the percentage of task sets that were schedulable by our schedulability test.

Each task set was generated as follows. The UUnifast-Discard algorithm described in (Davis and Burns, 2009) was used to generate n task utilizations $\{u_1 \dots u_n\}$ of some total utilization $u(\tau)$. The period T_i for each task was generated according to a log-uniform distribution in the range 10ms to 1000ms. All task periods were set to integer values by rounding down from any non-integer value. The total execution requirement of a task *without using a GPU* was set to $u_i \times T_i$. A portion

g_i was chosen from a uniform distribution in the range $[0, u_i \times T_i]$ to denote the execution of a task on the GPU. We assumed that a task exploits the parallelism provided by a GPU and executes in lesser time on a GPU when compared to a CPU. Speed up *SP* is the ratio of the execution time on the CPU and the execution time on the GPU. Thus, $G_i = g_i/SP$. For simplicity we assume that a job of a task makes one request to the GPU. The remaining execution time $(u_i \times T_i) - g_i$, was set to C_i. L_i was computed from G_i. If $G_i = 0$ then $L_i = 0$. Else, L_i was computed using Equation (5.17) for $k = 1$. Note that $L_i = G_i$ when $m = g$. Task deadline D_i was set equal to T_i for implicit-deadline task systems, and was chosen from a uniform distribution in the range $[u_i \times T_i, T_i]$ for constrained-deadline task systems. Utilization u_i generated above is referred to as the effective utilization of a task and $u(\tau)$ as the total effective utilization of task system τ. The actual utilization of a task is $U_i = \frac{C_i + L_i}{T_i}$ and the actual total utilization is $U(\tau) = \sum_{i=1}^{n} U(i)$.

The value of *SP* for different tasks in a task set depends on the amount of parallelism of each task on the GPU. For simplicity, in our experiments we assume that all tasks have the same speed up. A speed up strictly greater than 1 is needed to justify the use of a GPU, higher values of *SP* are better.

We randomly generated implicit-deadline task sets, as described above, for m processors with total effective utilization in the range $[m \times 0.05, m \times 2)$, and in increments of $m \times 0.1$. For each of the total effective utilization values, 1000 sets of effective utilization values were generated such that each set had n values. From the generated utilization values and speed up *SP* the following task sets were generated:

- LPE - limited-preemptive task set with $g = m$.
- LPL - limited-preemptive task set with $g = m/2$.
- FP - fully-preemptive task set that does not use GPUs for any part of its computation. The parameters for each task τ_i were as follows; $C_i = u_i \times T_i$, $G_i = 0$, $L_i = 0$. Thus, the actual utilization of a task was equal to its effective utilization.

We refer to task sets generated from the same effective utilization values as *corresponding* task sets. Thus, LPE, LPL, and FP are corresponding task sets.

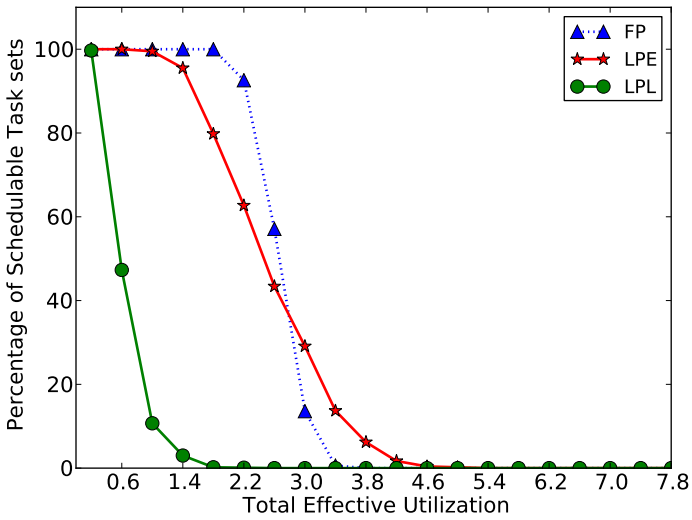


Figure 5.4: Limited-preemption schedulability test: m = 4, n = 40, SP = 30

Our schedulability test was applied to each of the above generated task sets and the results obtained are shown in Figure 5.4. Different values of m, n, and SP were used in these experiments. In Figure 5.4, the results are shown for $m = 4$, $n = 40$, and $SP = 30$.

Observation 1. We observe that task set LPE has better schedulability than FP for higher values of total effective utilization and LPE has significantly better schedulability than LPL. Note that worst-case execution times for tasks under fully-preemptive scheduling are often larger. Thus, in practice the schedulability of FP will be lower than what is shown in the graph for a given task set. From this experiment we can conclude that a small difference in the ratio of the number of processors to the number of GPUs makes a significant difference in schedulability. This is due to an increase in the length of the non-preemptive execution L_i for any task τ_i that accesses a GPU. Also, note that in Figure 5.4 the schedulability of task set FP caps at a total effective utilization of 4 because its actual total utilization is equal to its effective total utilization and $m = 4$.

To analyze the affect of speed up SP, implicit-deadline task sets were generated for m processors and g GPUs with $m = g$. The total effective utilization of the task sets generated was in the range

$[m \times 0.05, m \times 2)$ in increments of $m \times 0.1$. In Figure 5.5 for each total effective utilization, 1000 task sets were generated with $SP = 30$ and then for each of the 1000 task sets, corresponding task sets, that is task sets with the same effective utilization values were generated with $SP = 20$. The results for $m = 4$ and $n = 40$ are shown in Figure 5.5.

Observation 2. With smaller values of SP the length of the non-preemptive execution L_i of each task τ_i increases. This reduces schedulability. For comparison, we also show the schedulability of the corresponding fully-preemptive task set in Figure 5.5. We observe that for smaller values of SP the schedulability of FP can be better than that of LPE.

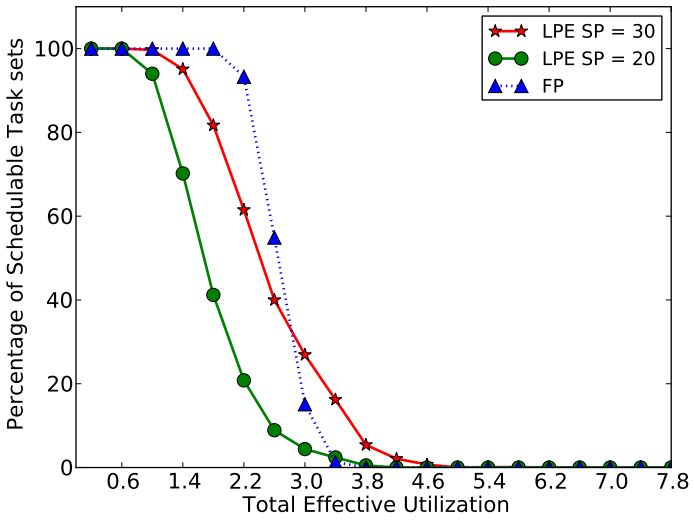


Figure 5.5: Limited-preemption schedulability test: m = 4, n = 40, m = g

The above experiment was repeated to analyze the affect of the number of tasks in a task set. In this case, for each total effective utilization mentioned above, 1000 task sets with $n = m \times 5$ tasks were generated and then 1000 task sets with $n = m \times 10$ tasks were generated. In both cases SP was the same. The results for $m = 4$ and $SP = 30$ are shown in Figure 5.6.

Observation 3. With smaller number of tasks in a task set the effective utilization of each task τ_i increases. As a result the length of its preemptive execution C_i and non-preemptive execution L_i increases. Therefore, schedulability decreases and we observe that for smaller values of total effective utilization LPE with $n = 40$ tasks has better schedulability than LPE with $n = 20$ tasks.

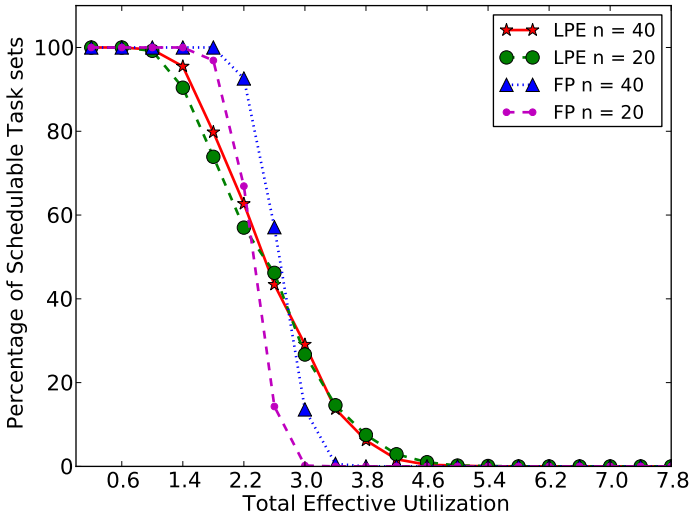


Figure 5.6: Limited-preemption schedulability test: m = 4, SP = 30, m = g

However, for larger values of total effective utilization the schedulability of LPE for larger number of tasks is dominated by the number of tasks that contribute to the term B_n (Equation 5.10), where as for smaller number of tasks, even though the length of L_i may be greater for each task, the number of tasks that contribute to the term B_n is fewer. Thus, we observe that for larger values of total effective utilization the schedulability of LPE for both $n = 40$ and $n = 20$ tasks is comparable. For comparison, we also show the schedulability of the corresponding fully-preemptive task sets in Figure 5.6 denoted as FP with $n = 40$ and $n = 20$ tasks.

5.7 Conclusion

Limited-preemption scheduling is an alternative to the extreme options of fully-preemptive scheduling and non-preemptive scheduling. While preemptions are better from a schedulability perspective, run-time overheads incurred by arbitrary preemptions can be large.

In this chapter, we have described a pseudo-polynomial time, demand-based schedulability test for limited-preemption scheduling under GEDF. We have shown that this test is necessary and sufficient for uniprocessors, and it is sufficient for multiprocessors. Further, we have shown

how to apply this schedulability test to a multiprocessor multi-GPU system with non-preemptive busy-waiting.

We have also indicated how further analysis may provide better analytical results for the multi-GPU system model under consideration. A comparison of analytical results under different multi-GPU system models is merited.

CHAPTER 6: SPEED SCALING ON UNIPROCESSORS

Thus far in this dissertation, we have abstracted away the frequency of a processor core by normalizing it, and denoting it as computing capacity or speed equal to 1. Processor cores are usually rated at a conservative estimate of the frequency at which they can operate. Thus, processors can often run at a frequency higher than the frequency rating provided by the manufacturer (that is at speed greater than 1). Running at a higher frequency improves the performance of a processor, since more computation can be done in a shorter duration. Operating a processor at a frequency higher than its suggested frequency rating is called overclocking. There is however, an upper limit on the frequency at which a processor can operate. This limit depends upon the maximum instantaneous power a processor can generate at any time t without being compromised. The relationship between frequency $f(t)$ and power $P(t)$ of a processor at any given time t is given by the following equation (Liu and Mok, 2003):

$$P(t) \propto C.V(t)^2.f(t),$$

where C is the capacitance in the wires (we assume that capacitance is a constant), $V(t)$ is the supply voltage, and $f(t)$ is the frequency of a processor at any given time t. However, $V(t)$ and $f(t)$ are related; there is a minimum voltage required to drive a processor at a desired frequency. This minimum voltage is approximately proportional to frequency. Since $V(t)$ is proportional to $f(t)$, this leads to the well known relation (Brooks et al., 2000): $P(t) \propto f(t)^3$. By ensuring that frequency $f(t)$ is under a certain limit we can limit the maximum power generated by a processor. In the remainder of this chapter we refer to frequency $f(t)$ as speed $S(t)$, speed is normalized frequency, and $f(t) \propto S(t)$. We then obtain the following relation:

$$P(t) \propto S(t)^3. \tag{6.1}$$

Thus, if a processor always operates below a certain maximum speed say $S_{\max}$, then we can limit the instantaneous power generated by the processor.

While overclocking a processor may violate the Thermal Design Power (TDP) rating of the processor. Violating the TDP rating causes a processor to generate heat at a rate faster than the cooling system can dissipate (Raghavan et al., 2012), which in turn causes the temperature of the processor to rise. In order to ensure that a processor is not damaged by the heat generated during overclocking it is necessary that the temperature of the processor is always under a certain value say $T_{\max}$.

In dynamic overclocking the speed at which a processor operates is varied at run-time while ensuring that the processor is not compromised. For example, the *Turbo Boost technology by Intel* (Rotem et al., 2012) enables dynamic overclocking when there is a demand and the operating environment is favorable. In this chapter we study dynamic overclocking, or speed scaling on uniprocessors when there is a demand, and under the constraints that the speed of the processor should not exceed $S_{\max}$ and the temperature of the processor should not exceed $T_{\max}$.

Real-time scheduling can benefit from processors with speed scaling; some sets of real-time jobs that cannot meet deadlines without overclocking may be able to meet deadlines with overclocking.

Prior work on real-time scheduling on processors with speed scaling include the following papers: (Yao et al., 1995; Liu and Mok, 2003; Bansal et al., 2004; AlEnawy and Aydin, 2004; Wang and Bettati, 2006b,a; Bansal et al., 2007; Ahn and Bettati, 2008).

The paper by (Yao et al., 1995) is a seminal paper on the theoretical study of processors with speed scaling. In (Yao et al., 1995) the objective is to vary the speed of a processor to construct a schedule for real-time jobs that minimizes the amount of *energy* used by the processor during the course of execution. (Energy is power $P(t)$ integrated over time.) The energy usage of a processor is an important concern for battery-operated devices. Other papers such as (Liu and Mok,

2003; AlEnawy and Aydin, 2004) also focus on minimizing the energy used by the processor. The algorithms, heuristics, and techniques presented in (Yao et al., 1995; Liu and Mok, 2003; AlEnawy and Aydin, 2004) are effective power management techniques that minimize the energy used by a processor, and in some cases minimize the maximum power used at any given time t.

When considering sustained execution minimizing the energy used by a processor in the primary concern. However, when the responsiveness of an application is the primary concern we need to ensure that a processor can meet the execution demand of the application in a short interval. Thus, if it is necessary for a processor to overclock in this short interval, then it is also necessary to ensure that the maximum power generated by the processor and the temperature of the processor are within desirable limits.

In (Wang and Bettati, 2006b,a; Ahn and Bettati, 2008) the authors use speed scaling to schedule real-time workload on processors such that maximum temperature reached by a processor is always below a certain maximum value. In this set of papers the authors adopt a *reactive speed scaling* technique, where the processor can overclock at speed S_H until it reaches a maximum temperature $T_{\max}$. Once $T_{\max}$ is reached the processor continues execution at a reduced equilibrium speed S_E, which keeps the temperature at or below $T_{\max}$. (Subsequently, the processor may or may not need to operate at speed S_H depending upon the workload and current temperature.) This ensures that the temperature of the processor is always below $T_{\max}$. Further, by choosing a suitable value of speed S_H we can also constrain the maximum power generated at any time t.

In one set of prior work (Bansal et al., 2004, 2007) the authors describe separate algorithms for minimizing the energy used by the processor, and for minimizing the maximum temperature reached by the processor. They also illustrate that power management techniques that are effective for minimizing energy may not be effective for minimizing temperature. The algorithm proposed in (Bansal et al., 2004, 2007) to minimize the maximum temperature reached by a processor does not assume a constraint on the maximum power or maximum speed.

In this work we assume that the maximum temperature $T_{\max}$ is given, and unlike the work in (Bansal et al., 2004, 2007) we have an additional constraint that the speed at which the processor

operates is at most $S_{\max}$, which is also given. We derive an *offline* schedule and a schedulability test for determining whether a given set of jobs specified according to the model described in Section 2.2 are schedulable on a uniprocessor platform under the given constraints. For convenience the job model is briefly described in the following section.

6.1 System Model

Consider a set of n jobs. Each job J_i has an arrival time a_i, worst-case execution time c_i, and deadline d_i. A job is scheduled correctly if it can execute for up to c_i time units in the interval $[a_i, d_i]$, that is in the interval between when it arrives at time a_i and before its deadline at time d_i. We seek to schedule a set of n such jobs J on a uniprocessor platform.

In the uniprocessor platform the temperature at any time t is denoted as $T(t)$, and the speed at any time t is denoted as $S(t)$. The temperature and speed are scaled such that the ambient temperature is 0, and the idle speed of the processor is 0. We assume that at time $t = 0$ the processor is operating at idle speed. Thus, $S(0) = 0$. Let the temperature of the processor at time $t = 0$ be denoted by T_0, where $0 \leq T_0 \leq T_{\max}$. Thus, $T(0) = T_0$. Once the processor starts executing, the following conditions should hold:

- Temperature constraint, $\forall t : 0 \leq T(t) \leq T_{\max}$,
- Speed constraint, $\forall t : 0 \leq S(t) \leq S_{\max}$.

In order to schedule a given set of jobs J on a processor with the above constraints the scheduler needs to decide at each time t, which job should execute and at what speed the processor must operate. If the scheduler always chooses a speed at most $S_{\max}$ then the speed constraint is satisfied. However, this alone does not satisfy the temperature constraint. Therefore, we derive a temperature model that can be used to determine the temperature of the processor at any time t, and then ensure that the temperature constraint is satisfied.

The temperature model we use was first described in (Bansal et al., 2004, 2007). This temperature model was derived from the observation that at any given time the net change in temperature can be shown to be proportional to:

- The heating due to the electric power generated by the device, and
- The cooling due to *Newton's law*.

To be specific, the rate of change of temperature at any given time t, $dT(t)/dt$, is proportional to $P(t)$ which is the power generated by the device at time t:

$$\frac{dT(t)}{dt} \propto P(t) \propto S(t)^3.$$

Recall that as per Equation 6.1, $P(t) \propto S(t)^3$.

According to Newton's law the rate of change of temperature of an object is proportional to the difference between the temperature of the object and the ambient temperature $T_{ambient}$, (Campbell and Haberman, 2008). Thus, $dT(t)/dt$ decreases in proportion to $(T(t) - T_{ambient})$. We make the simplifying assumption that $T_{ambient}$ is a constant, and we scale the temperatures such that $T_{ambient} = 0$. Thus, we can write Newton's law as:

$$\frac{dT(t)}{dt} \propto -T(t).$$

The rate of change of temperature, $dT(t)/dt$, can then be defined as follows:

$$\frac{dT(t)}{dt} = a \times S(t)^3 - b \times T(t), \tag{6.2}$$

where a and b are constants (Bansal et al., 2007). The constant $b \geq 0$ is called the cooling parameter. We have derived a relation between the speed of a processor and the rate of change of temperature. The temperature at any time t can be obtained by solving Equation 6.2.

6.2 Related Work

To the best of our knowledge only one set of papers (Bansal et al., 2004, 2007) consider temperature optimization. The goal in (Bansal et al., 2004, 2007) is to minimize $T_{\max}$.

Our work is different from (Bansal et al., 2004, 2007) because we assume that we are given $T_{\max}$ and our goal is to schedule jobs such that both the temperature and speed constraints are satisfied. We use results from (Bansal et al., 2007) to derive a schedule and a schedulability test for a set of n jobs. In particular, we use results from (Bansal et al., 2007) to derive the maximum work that can be done in an interval $[t_x, t_y]$ and to obtain a speed profile, that is speed as a function of time t, during this interval.

In (Bansal et al., 2007), the authors let $MaxW(t_x, t_y, T_x, T_y)$ denote the maximum work that can be done starting at time t_x at temperature T_x, and ending at time t_y at temperature T_y, subject to the temperature constraint throughout the interval $[t_x, t_y]$. The authors compute $MaxW(t_x, t_y, T_x, T_y)$ by first solving the unconstrained work problem $UMaxW(t_x, t_y, T_x, T_y)$, defined as the maximum possible work that can be done during the interval $[t_x, t_y]$ subject to the boundary constraint that $T(t_x) = T_x$ and $T(t_y) = T_y$. However, the temperature at any time in the interval $[t_x, t_y]$ is allowed to exceed $T_{\max}$. The following lemma from (Bansal et al., 2007) provides further insight into the relation between T_x and T_y.

Lemma 6.1. *(Bansal et al., 2007). Suppose that T_x and T_y are at most T_{max}. Each of the quantities $MaxW(t_x, t_y, T_x, T_y)$ and $UMaxW(t_x, t_y, T_x, T_y)$ are well defined if and only if $T_y \geq T_x e^{-b(t_y - t_x)}$.*

Proof. The Lemma follows from the fact that the power generated in the interval $[t_x, t_y]$ should be nonnegative. □

Let $UMaxT(t) = UMaxT(t_x, t_y, T_x, T_y)(t)$ denote the temperature as a function of time t that solves $UMaxW(t_x, t_y, T_x, T_y)$. Thus, $UMaxT(t)$ is the temperature curve that maximizes the amount of work done without the temperature constraint. From (Bansal et al., 2007) we know that $UMaxT(t)$ is as follows:

$$UMaXT(t) = c.e^{(-bt)} + d.e^{(-bt3/2)}, \tag{6.3}$$

$$\text{where } c+d = T_x, \text{ and } d = \frac{T_x.e^{(-b(t_y-t_x))} - T_y}{e^{(-b(t_y-t_x))} - e^{(-b(t_y-t_x)3/2)}}.$$

Let γ and β be defined as follows.

Definition 6.1. *(Bansal et al., 2007).* γ is the largest value of t_y for which the maximum temperature attained by the curve $UMaxT(0,t_y,T_x,T_{\max})(t)$ during the interval $[0,t_y]$ does not exceed $T_{\max}$.

Thus, the curve $UMaxT(0,\gamma,T_x,T_{\max})(t)$ satisfies the temperature constraint, but for any value γ' greater than γ the curve $UMaxT(0,\gamma',T_x,T_{\max})(t)$ does not satisfy the temperature constraint.

Definition 6.2. *(Bansal et al., 2007).* β is the largest value of t_y for which the maximum temperature attained by the curve $UMaxT(0,t_y,T_{\max},T_y)(t)$ during the interval $[0,t_y]$ does not exceed $T_{\max}$.

Thus, the curve $UMaxT(0,\beta,T_{\max},T_y)(t)$ satisfies the temperature constraint, but for any value β' greater than β the curve $UMaxT(0,\beta',T_{\max},T_y)(t)$ does not satisfy the temperature constraint.

$MaxT(t) = MaxT(t_x,t_y,T_x,T_y)(t)$, which is the temperature curve that maximizes the amount of work done under the temperature constraint, can be derived from $UMaxT(t)$, γ, and β as shown in the following lemma.

Lemma 6.2. *(Bansal et al., 2007). If* $(t_y - t_x) \leq (\gamma+\beta)$*, then* $MaxT(t) = UmaxT(t)$*. If* $(t_y - t_x) > (\gamma+\beta)$*, then the curve* $MaxT(t)$ *travels along the curve* $UMaxT(t_x,(t_x+\gamma),T_x,T_{max})(t)$*, then stays at* T_{max} *until* $t_y-\beta$*, and finally travels along the curve* $UMaxT((t_y-\beta),t_y,T_{max},T_y)(t)$.

We refer the reader to (Bansal et al., 2007) for details about the proof.

In this work, we use the temperature curve $MaxT(t)$ to derive a speed profile $UMaxS(t) = UMaxS(t_x,t_y,T_x,T_y)(t)$, which we define as the speed curve that maximizes the amount of work that can be done under the temperature constraint but without the speed constraint. Rewriting Equation 6.2 we get:

$$S(t) = \left(\frac{dT(t)/dt + bT(t)}{a} \right)^{1/3}.$$

Thus, by Equation 6.2 and Lemma 6.2 the speed function $UMaxS(t)$ corresponding to the temperature curve $MaxT(t)$ is as follows:

If$(t_y - t_x) \leq (\gamma + \beta)$:

$$UMaxS(t) = \left(\frac{d(MaxT(t))/dt + bMaxT(t)}{a} \right)^{1/3}, t \in [t_x, t_y]$$

Else :

$$UMaxS(t) = \left(\frac{d(MaxT(t))/dt + bMaxT(t)}{a} \right)^{1/3}, t \in [t_x, t_x + \gamma] \wedge [t_y - \beta, t_y]$$

$$UMaxS(t) = \left(\frac{bT_{\max}}{a} \right)^{1/3}, t \in (t_x + \gamma, t_y - \beta). \tag{6.4}$$

The maximum work $MaxW(t_x, t_y, T_x, T_y)$ that can be done under the temperature constraint but without the speed constraint is then the integral of $UMaxS(t)$ in the interval $[t_x, t_y]$:

$$MaxW(t_x, t_y, T_x, T_y) = \int_{t_x}^{t_y} UMaxS(t)dt. \tag{6.5}$$

The above Equation 6.5 has been solved in (Bansal et al., 2007) as follows:

If$(t_y - t_x) \leq (\gamma + \beta)$:

$$MaxW = -(d/a)^{1/3} \left(\frac{b}{2} \right)^{1/2} (1 - e^{(-b(t_y - t_x)/2)})$$

$$\text{where, } d = \left(\frac{T_x . e^{(-b(t_y - t_x))} - T_y}{e^{(-b(t_y - t_x))} - e^{-b((t_y - t_x)3/2)}} \right)$$

Else :

$$MaxW = -(d_1/a)^{1/3}\left(\frac{b}{2}\right)^{1/2}(1-e^{(-b\gamma/2)}) + (t_y - t_x - \gamma - \beta)\left(\frac{bT_{\max}}{a}\right)^{1/3} + \ldots$$
$$-(d_2/a)^{1/3}\left(\frac{b}{2}\right)^{1/2}(1-e^{(-b\beta/2)})$$
$$\text{where, } d_1 = \left(\frac{T_x.e^{(-b\gamma)} - T_{\max}}{e^{(-b\gamma)} - e^{(-b\gamma 3/2)}}\right), \text{and } d_2 = \left(\frac{T_{\max}.e^{(-b\beta)} - T_y}{e^{(-b\beta)} - e^{(-b\beta 3/2)}}\right). \quad (6.6)$$

Note that in the above equation, a and b are constants (refer Equation 6.2). Also as per Definitions 6.1 and 6.2, γ and β can be derived from the values of T_x and T_y respectively. Thus, if we are given the values of (t_x, t_y, T_x, T_y) then for any interval $[t_x, t_y]$ we can compute the value of $MaxW$ from Equation 6.6.

6.3 Offline scheduling of jobs

We propose a schedule and a schedulability test for a set of jobs J. In Section 6.3.1, we divide the schedule into many intervals I_k. We determine the subset of jobs $J(k)$ in J that must fully execute in each interval I_k. We also ensure that all jobs in J are assigned to some interval. Within an interval the jobs execute as per EDF, thus the job with the earliest deadline is chosen for execution. In Section 6.3.2, we derive a speed profile (speed as function of time) for each interval. The jobs that must execute in each interval, and the speed profile for each interval are determined before run-time, thus we are scheduling the jobs *offline*. Finally in Section 6.3.3, we put together the results from Sections 6.3.1 and 6.3.2 to obtain a sufficient schedulability test for the set of jobs J under the given temperature and speed constraints.

6.3.1 Determining intervals and jobs per interval

We first derive a schedule that is composed of one or more intervals. To compute the intervals in our schedule we use the following definitions of intensity of an interval $g(I)$ and critical interval I^* provided in (Yao et al., 1995).

Definition 6.3. *(Yao et al., 1995)* The intensity of an interval $I = [z, z']$, starting at time z and ending at time z' is:

$$g(I) = \frac{\sum c_i}{z' - z},$$

where the sum is taken over all jobs J_i with $[a_i, d_i] \in I$.

Note that $g(I)$ is the minimum constant speed the processor has to maintain in the interval I to ensure that all jobs J_i with $[a_i, d_i] \in I$ meet their deadlines.

Definition 6.4. *(Yao et al., 1995)* Let $I^* = [z, z']$ be an interval that maximizes $g(I)$. We call I^* a critical interval for J and the set of jobs $J(I^*) = \{J_i | [a_i, d_i] \in [z, z']\}$ the critical group for J.

Within the interval $I^* = [z, z']$ only the jobs in the critical group of $J(I^*)$ must execute, and the minimum constant speed of the processor must be $g(I^*)$. We consider I^* to be one of the intervals I_k in our schedule.

Repeat the following steps until J is empty:

1. Identify a critical interval $I^* = [z, z']$; this is the interval that maximizes $g(I)$. Also obtain the set of jobs $J(I^*)$ that must execute in interval I^*, and the minimum constant speed $g(I^*)$ the processor must maintain in this interval.
2. Remove $J(I^*)$ from the list of jobs J. Let $J = J - J(I^*)$.
3. If the deadline of any job J_i is in $I^* = [z, z']$ then reset the deadline of the job such that $d_i = d_i - (z' - z)$. If the arrival time of any job J_i is in $[z, z']$ then reset the arrival time of the job such that $a_i = z'$.

Figure 6.1: Obtaining critical intervals

We can obtain all the intervals in our schedule by the method described in (Yao et al., 1995), also shown in Figure 6.1. We repeatedly compute the critical interval for the remaining set of jobs. At the end of each iteration we obtain the following: i) a critical interval I^*, which is one of the intervals I_k in our schedule, ii) the jobs $J(k)$ that need to be scheduled in this interval, iii) and the

minimum constant speed $S_k = g(I^*)$ that the processor must maintain in interval I_k so that the jobs in $J(k)$ meet their deadlines. (Note that if for some interval I_k, $S_k > S_{\max}$ then we can conclude that the speed constraint of our schedule will be violated, thus the jobs are not schedulable under the given constraints. Therefore, for the rest of the discussion we assume that for all intervals I_k, $S_k \leq S_{\max}$.)

In Step 1 of Figure 6.1, we identify a critical interval but we do not specify how we do so. A simple algorithm to identify the critical interval for a set of jobs J would consider all time points in the set $\{a_i, d_i\}$ for all jobs $J_i \in J$, and compare the intensity of every interval that starts at each time point under consideration. The interval with the largest intensity is the critical interval. This is a straightforward algorithm and has a run-time complexity of $\mathcal{O}(n^2)$ where n is the number of jobs in J. Other algorithms to identify a critical interval that have better run-time complexity can be derived, but we do not discuss them here.

6.3.2 Determining a speed profile per interval

We now describe how to derive a speed profile for interval I_k such that the amount of work done in the interval is at least S_k times the length of the interval, and the temperature constraint it satisfied (I_k and S_k were derived in Section 6.3.1). Further, the speed profile we derive minimizes the temperature at the end of interval I_k. Note that for an interval I_k a possible speed profile is to operate the processor at speed S_k for the entire interval. However, with this speed profile the temperature constraint may be violated, and it depends upon the length of the interval and the value of S_k. Another possible speed profile corresponds to executing the work as soon as possible at the beginning of the interval, and to idle the processor for the rest of the interval. The speed profile we derive is different from the above mentioned speed profiles because in the above cases the temperature at the end of interval I_k may not be minimized. Intuitively, the lower the temperature at the end of an interval the more overclocking we can achieve in the next interval.

Once we compute the speed profile we check if the speed constraint is satisfied. Note that the speed profile we derive is a continuous function of speed with respect to time. We can easily

discretize the function is a naïve manner for use in a real-time system. However, we leave an effective implementation of the speed profile as future work. Before we proceed we need a few additional definitions.

- Let the time at the beginning of interval I_k be represented as t_{k-1}, and the time at the end of the interval be represented as t_k.
- Let the temperature at the beginning of interval I_k be represented as T_{k-1}, and the temperature at the end of the interval be represented as T_k.

In Section 6.3.1 we determined each interval I_k of our schedule, which also provided the start and end points of the interval. Thus, we know the values of t_{k-1} and t_k for each interval I_k. Let the values of t_{k-1} and t_k be shifted such that $t_0 = 0$. Then by the assumptions made in Section 6.1 the processor is idle at time t_0, and temperature T_0 is given. We however do not know T_k for any interval I_k. Since we know that the amount of work that needs to be done in interval I_k is $S_k \times (t_k - t_{k-1})$, we can compute a value of T_k for each interval I_k by solving Equation 6.6 such that $MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$ is set equal to $S_k \times (t_k - t_{k-1})$. ($MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$, represents the maximum work that can be done in interval I_k under the temperature constraint with the given starting and ending conditions.)

In the following Lemmas we derive some properties of $UMaxS(t)$ and $MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$, which show that computing the value of T_k as above gives the minimum value of T_k for interval I_k. (The speed profile $UMaxS(t) = UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t)$ maximizes the work done in interval I_k under the temperature constraint with the given starting and ending conditions. $UMaxS(t)$ can be greater than $S_{\max}$ for some value of t.)

Lemma 6.3. *If $T_k^* > T_k$ and $S(t_{k-1})$ is given, then:*

$$UMaxS(t_{k-1}, t_k, T_{k-1}, T_k^*)(t) \geq UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t) \forall t \in [t_{k-1}, t_k].$$

Proof. Let t^* be the last time point at which the two curves intersect. We know that such a time exists because at time t_{k-1} the value of both curves is $S(t_{k-1})$. Since both the curves maximize the amount of work done, they are identical in the interval $[t_{k-1}, t^*]$. Thus, in the interval $[t_{k-1}, t^*]$ the Lemma holds.

We need to prove that the lemma holds in the interval $[t^*, t_k]$. At time t_k, $UMaxS(t_{k-1}, t_k, T_{k-1}, T_k^*)(t_k)$ is strictly greater than $UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t_k)$, because $T_k^* > T_k$ is given. We also know that the curves do not intersect after t^*. Thus, the lemma follows. □

Lemma 6.4. *If $T_k^* > T_k$ and $S(t_{k-1})$ is given, then:*

$$MaxW(t_{k-1}, t_k, T_{k-1}, T_k^*) > MaxW(t_{k-1}, t_k, T_{k-1}, T_k).$$

Proof. This follows directly from Lemma 6.3. Note that if $MaxW(t_{k-1}, t_k, T_{k-1}, T_k^*) = MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$ then $MaxW(t_{k-1}, t_k, T_{k-1}, T_k^*)$ is not the maximum amount of work the can be done in the interval $[t_{k-1}, t_k]$, which is a contradiction. □

Lemma 6.5. *The minimum value of T_k for an interval I_k satisfies the following:*

$$S_k \times (t_k - t_{k-1}) = MaxW(t_{k-1}, t_k, T_{k-1}, T_k).$$

Proof. From Lemma 6.4 we know that for a value lower than T_k the amount of work done in interval I_k will be strictly less than $S_k \times (t_k - t_{k-1})$.

The minimum value of T_k such that $S_k \times (t_k - t_{k-1}) = MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$ can be obtained by doing a binary search in the range $[T_{k-1}e^{-b(t_k - t_{k-1})}, T_{\max}]$ (By Lemma 6.1, $T_k \geq T_{k-1}e^{-b(t_k - t_{k-1})}$). For now assume that we know T_{k-1}. As per Definition 6.1, we can derive γ given T_{k-1}.

Suppose that initially $T_k = (T_{left} + T_{right})/2$, $T_{left} = T_{k-1}.e^{-b(t_k - t_{k-1})}$, and $T_{right} = T_{\max}$. The chosen value of T_k is used to compute β as per Definition 6.2, and then to obtain $MaxW(t_{k-1}, t_k, T_{k-1}, T_k)$ by solving Equation 6.5.

- If $MaxW(t_{k-1}, t_k, T_{k-1}, T_k) = S_k \times (t_k - t_{k-1})$, then we have found the minimum value of T_k.

- If $MaxW(t_{k-1}, t_k, T_{k-1}, T_k) > S_k \times (t_k - t_{k-1})$, then from Lemma 6.4 we know that we can choose a lower value of T_k. Therefore, we set $T_{right} = T_k$.

- If $MaxW(t_{k-1}, t_k, T_{k-1}, T_k) < S_k \times (t_k - t_{k-1})$, then from Lemma 6.4 we know we have to choose a higher value of T_k. Therefore, we set $T_{left} = T_k$.

We re-compute $T_k = (T_{left} + T_{right})/2$ and repeat the above steps until we obtain a value of T_k that satisfies: $MaxW(t_{k-1}, t_k, T_{k-1}, T_k) = S_k \times (t_k - t_{k-1})$ □

For interval I_1 we already know the value of the parameters (t_0, t_1, T_0), and we compute the minimum value of T_1 as per Lemma 6.5. We use the value of the parameters (t_1, t_2, T_1) to compute the minimum value of T_2 and so on. Thus, for each interval I_k:

- We compute the parameters $(t_{k-1}, t_k, T_{k-1}, T_k)$

- Then by Equation 6.3 and Lemma 6.2, we compute $MaxT(t) = MaxT(t_{k-1}, t_k, T_{k-1}, T_k)(t)$. $MaxT(t)$ is the temperature profile that maximizes the work done in interval I_k under the temperature constraint for the given starting and ending conditions.

- From $MaxT(t)$ we compute the speed profile $UMaxS(t)$ by Equation 6.4.

In this manner, we are able to obtain the speed profile $UMaxS(t)$ for each interval. We know that this speed profile satisfies the temperature constraint. What remains is to determine whether the speed profile satisfies the speed constraint. This is verified in the following schedulability test.

6.3.3 Sufficient schedulability test

Putting together the results from Sections 6.3.1 and 6.3.2 we derive a sufficient schedulability test for a set of jobs J.

The pseudo-code for the schedulability test is shown in Figure 6.2. The schedulability test returns INTERVAL NOT SCHEDULABLE, if in Step 4 for some interval I_k, $S_k \times (t_k - t_{k-1}) > MaxW(t_{k-1}, t_k, T_{k-1}, T_{\max})$. This is in fact a necessary condition for the following reasons. As

```
1   First obtain the critical intervals as shown in Figure 6.1.
2   Initially k = 1. We are given T_{k-1} = T_0.
3   for each interval I_k:
4           if S_k × (t_k − t_{k-1}) > MaxW(t_{k-1}, t_k, T_{k-1}, T_max) :
5                   return INTERVAL NOT SCHEDULABLE
6             else
7                   Compute the minimum value of T_k (Lemma 6.5)
8                   if UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t) > S_max for some t ∈ [t_{k-1}, t_k]
9                           return INTERVAL NOT SCHEDULABLE
10  return SCHEDULABLE
```

Figure 6.2: Sufficient schedulability test for offline scheduling of jobs.

per Definition 6.4, the set of jobs $J(k)$ that execute in interval I_k have to execute in this interval because these jobs arrive and have deadlines within interval I_k. Further, we derive the minimum end temperature T_{k-1} for interval I_{k-1}, which is the same as deriving the minimum starting temperature for interval I_k. Intuitively, the lower the starting temperature of an interval, the greater the maximum amount of work that can be done in the interval. Thus, if for any interval I_k, $S_k \times (t_k - t_{k-1}) > MaxW(t_{k-1}, t_k, T_{k-1}, T_{\max})$ then as per Lemma 6.4, the set of jobs $J(k)$ are indeed not schedulable in accordance to the temperature constraint.

Suppose that in Step 7 we obtain a value for T_k such that $\forall t \in [t_{k-1}, t_k] : UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t) \leq S_{\max}$. In this case, $UMaxS(t)$ is a speed profile for interval I_k that satisfies both the speed and temperature constraints, and we can move onto the next interval.

If however the value of T_k is such that $UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t) > S_{\max}$ for some value $t \in [t_k, t_{k-1}]$ (Step 8), then by Lemma 6.4, for a lower value of T_k the amount of work done in interval I_k will be less than $S_k \times (t_k - t_{k-1})$. By Lemma 6.3, a higher value of T_k will generate a speed profile $UMaxS(t)$ that will continue to violate the speed constraint. Thus, the schedulability test returns INTERVAL NOT SCHEDULABLE. However, the speed profile we derive is not optimal with respect to the problem, therefore there may exist a speed profile $S(t)$, which unlike $UMaxS(t)$

does not maximize the amount of work done in interval I_k. Instead the speed profile $S(t)$ does only the necessary amount of work and satisfies the speed constraint for a higher value of T_k (we do not compute the speed profile $S(t)$). Thus, the schedule may still be feasible under the given constraints. In order to determine if $UMaxS(t) > S_{\max}$ for some $t \in [t_k, t_{k-1}]$, we can derive a speed profile which is a step-function that upper bounds $UMaxS(t)$, and then determine whether the derived speed profile exceed $S_{\max}$ for some $t \in [t_k, t_{k-1}]$. Thus, this test is only a sufficient schedulability test.

The **run-time complexity** of the schedulability test in Figure 6.2, depends upon the run-time complexity of i) Step 1 in which we compute the critical intervals, ii) Step 7 in which we perform a binary search to compute the minimum value of T_k for each interval, iii) and Step 8 in which we determine whether the condition- $UMaxS(t_{k-1}, t_k, T_{k-1}, T_k)(t) > S_{\max}$ for some $t \in [t_{k-1}, t_k]$ is satisfied. The run-time complexity of our schedulability test is dominated by the condition in Step 8. We can derive a speed profile, which is a step-function that upper bounds $UMaxS(t)$ in pseudo-polynomial time. We can then determine whether the derived speed profile exceeds $S_{\max}$ for some $t \in [t_{k-1}, t_k]$. Thus, the run-time complexity of the proposed schedulability test is pseudo-polynomial with respect to the job parameters.

6.4 Conclusion

In this chapter we assume that dynamic overclocking, or speed scaling is allowed on a processor as long as there is a demand for overclocking, and the temperature and speed constraints are satisfied. We have identified a temperature model described in (Bansal et al., 2007) that justifiably reflects the dynamic overclocking behavior allowed on a processor.

We have determined an offline schedule for a set of one-shot jobs on such processors. The schedule determines which jobs should be scheduled in certain intervals that we derive, and also determines a speed profile (that is speed as a function of time) for each of the intervals. We have proposed a sufficient schedulability test for the schedule that verifies whether for a given interval and speed profile, the jobs scheduled in the interval meet their deadline, and the temperature and speed constraints are both satisfied.

CHAPTER 7: SUMMARY

We now provide a summary of our contributions. We also indicate how our contributions can be extended and further improved.

We have proposed two partitioning algorithms. One is a PTAS for partitioning on processors with just one limited resource. This PTAS partitioning algorithm can be extended to partition tasks onto processors with more than one limited resource. However, for an arbitrary (but fixed) number of limited resources our extension to the PTAS partitioning algorithm has a large run-time complexity; the run-time complexity is polynomial with respect to the task parameters but the degree of the polynomial is very large. Thus, we propose a APX partitioning algorithm, which is a generalization of the first-fit partitioning algorithm for a single limited resource. Although the PTAS algorithm has a smaller resource augmentation bound, the APX algorithm is more efficient with respect to run-time complexity.

The partitioning problem is solved before run-time for any given task set and computing platform. However, it is likely that the specifications of the task set or the computing platform may change during the process of deploying the task set onto the computing platform, and a partitioning algorithm may need to be applied every time the specifications change. Thus, a partitioning algorithm with an efficient run-time complexity is preferable. This leads to our conclusion that the APX partitioning algorithm is more pragmatic than the PTAS partitioning algorithm. We also derive a first-fit decreasing heuristic for the APX partitioning algorithm. We use schedulability experiments to determine the effectiveness of our heuristic. Further, we can derive worst-fit and best-fit partitioning algorithms based on the heuristic that we have described. We however, do not know the exact resource augmentation bound of the worst-fit and best-fit partitioning algorithms.

We apply the APX partitioning algorithm and derive a partitioning algorithm for a mixed-criticality task model. We determine the resource augmentation bound and the run-time complexity of our mixed-criticality partitioning algorithm. We also propose pragmatic improvements for our algorithm. We do not know the exact resource augmentation bound of our algorithm with the pragmatic improvements, but we have shown via schedulability experiments that these improvements are in fact effective in increasing the schedulability of our algorithm.

We then consider scheduling under limited-preemptions. In limited-preemption scheduling a job of a task may execute non-preemptively over short intervals (possibly to gain exclusive access to a shared resource), but executes preemptively otherwise. We derive a demand-based schedulability test for limited-preemption scheduling under global EDF. This schedulability test extends an existing schedulability test for fully-preemptive global EDF scheduling. We show that the run-time complexity of the schedulability test is pseudo-polynomial with respect to the task parameters. We also show that the schedulability test is necessary and sufficient for uniprocessors and sufficient for multiprocessors.

Recently, GPUs are being incorporated as a shared resource in real-time systems. Thus far execution on GPUs is non-preemptive, therefore when a job is granted access to non-preemptively execute on a GPU one option is for the job to busy-wait non-preemptively on the CPU (other options are to busy-wait preemptively or self-suspend on the CPU). We have shown how to apply the schedulability test that we derived for limited-preemption scheduling, as a schedulability test for a system model that incorporates GPUs as a shared resource and allows non-preemptive busy-waiting. In this system model we assume that only one job can execute on a GPU at any given time. However, two jobs of two different tasks can in fact non-preemptively execute on a GPU at the same time; one job can execute non-preemptively on the GPU copy engine (CE), and another job can execute non-preemptively on the GPU execution engine (EE). We leave the schedulability analysis of this type of parallelism as future work.

Finally, we study the dynamic overclocking behavior allowed on a processor and determine how real-time jobs can be scheduled on processors that enable this behavior. We assume that a

processor can overclock only if it needs to meet the execution demand of the jobs being scheduled, and if the speed and temperature constraints are satisfied. The speed constraint ensures that the speed (also instantaneous power) at which the processor is operating at any time is under a desirable limit, and the temperature constraint ensures that the temperature at any time is under a desirable limit.

We identify a system model and propose an offline scheduling algorithm for scheduling jobs on such processors. In our scheduling algorithm we determine critical intervals and determine the jobs that need to be scheduled in each interval, before run-time. The jobs in each interval are scheduled as per EDF. We also determine the speed profile for each interval, which is a continuous function of speed with respect to time. (We leave the implementation of this continuous speed profile as future work.) We propose a sufficient schedulability test to determine if each interval can be scheduled as per its speed profile such that all jobs meet their deadline, and the speed and temperature constraints are satisfied. The next step is to identify a scheduling algorithm for online scheduling of periodic/sporadic tasks.

BIBLIOGRAPHY

Ahn, Y. and Bettati, R. (2008). Transient overclocking for aperiodic task execution in hard real-time systems. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

AlEnawy, T. and Aydin, H. (2004). On energy-constrained real-time scheduling. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Bansal, N., Kimbrel, T., and Pruhs, K. (2004). Dynamic speed scaling to manage energy and temperature. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*.

Bansal, N., Kimbrel, T., and Pruhs, K. (2007). Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):3:1–3:39.

Baruah, S. (2005). The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Baruah, S. (2012). Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*.

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., van der Ster, S., and Stougie, L. (2012). The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Baruah, S. and Burns, A. (2006). Sustainable scheduling analysis. In *Proceedings of the IEEE Real-time Systems Symposium (RTSS)*.

Baruah, S., Burns, A., and Davis, R. (2011). Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Baruah, S., Chattopadhyay, B., Li, H., and Shin, I. (2014). Mixed-criticality scheduling on multiprocessors. *Real-Time Systems: The International Journal of Time-Critical Computing*, 50(1):142–177.

Baruah, S. and Fisher, N. (2004). A dynamic-programming approach to task partitioning among memory-constrained multiprocessors. In *Proceedings of the International Conference on Real-time Computing Systems and Applications*. Springer-Verlag.

Baruah, S. and Fohler, G. (2011). Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Baruah, S., Mok, A., and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Bastoni, A., Brandenburg, B., and Anderson, J. (2010). An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Bertogna, M. and Baruah, S. (2010). Limited preemption EDF scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics*, 6(4):579–591.

Bertogna, M., Cirinei, M., and Lipari, G. (2005). Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill.

Brooks, D., Bose, P., Schuster, S., Jacobson, H., Kudva, P., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., and Cook, P. (2000). Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Transactions on Micro*, 20(6):26–44.

Buttazzo, G., Bertogna, M., and Yao, G. (2013). Limited preemptive scheduling for real-time systems. A Survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15.

Campbell, S. and Haberman, R. (2008). *Introduction to Differential equations with Dynamical systems*, pages 68 – 69. Princeton university press.

Chattopadhyay, B. and Baruah, S. (2011). A lookup-table driven approach to partitioned scheduling. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*.

Chattopadhyay, B. and Baruah, S. (2012). Partitioned scheduling of implicit-deadline task systems under multiple resource constraints. In *Proceedings of the IEEE International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA)*.

Chattopadhyay, B. and Baruah, S. (2014). Limited-preemption scheduling on multiprocessors. In *Proceedings of the ACM International Conference on Real-Time Networks and Systems (RTNS)*.

Chekuri, C. and Khanna, S. (2004). On multidimensional packing problems. *SIAM Journal of Computing*, 33(4):837–851.

Cong, L. and Anderson, J. (2013). Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Davis, R. and Burns, A. (2009). Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Dertouzos, M. (1974). Control robotics : the procedural control of physical processors. In *Proceedings of the International Federation for Information Processing (IFIP) Congress*.

Dorin, F., Richard, P., Richard, M., and Goossens, J. (2010). Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305 – 331.

Elliott, G. and Anderson, J. (2013). An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *Real-Time Systems*, 49(2):140–170.

Elliott, G., Ward, B., and Anderson, J. (2013). GPUSync: A framework for real-time gpu management. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Fisher, N. (2007). *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill.

Fisher, N., Anderson, J., and Baruah, S. (2005). Task partitioning upon memory-constrained multiprocessors. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

Fisher, N. and Baruah, S. (2006). The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. In *Proceedings of the ACM International Conference on Real-Time and Network Systems (RTNS)*.

Guan, N., Ekberg, P., Stigge, M., and Yi, W. (2011). Effective and efficient scheduling for certifiable mixed criticality sporadic task systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Herman, J., Kenna, C., Mollison, M., Anderson, J., and Johnson, D. (2012). RTOS support for multicore mixed-criticality systems. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Hochbaum, D. S. and Shmoys, D. B. (1987). Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162.

Huang, H.-M., Gill, C., and Lu, C. (2012). Implementation and evaluation of mixed-criticality scheduling algorithms for periodic tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Jeffay, K., Stanat, D., and Martel, C. (1991). On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Kalyanasundaram, B. and Pruhs, K. (2000). Speed is as powerful as clairvoyance. *Journal of the ACM*, 37(4):617–643.

Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. (2011a). RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011b). TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Conference on Annual Technical Conference*.

Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. (2012). Gdev: First-class gpu resource management in the operating system. In *Proceedings of the USENIX Conference on Annual Technical Conference*.

Kim, J., Andersson, B., de Niz, D., and Rajkumar, R. R. (2013). Segment-fixed priority scheduling for self-suspending real-time tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Kou, L. T. and Markowsky, G. (1977). Multidimensional bin packing algorithms. *IBM Journal of Research and Development*, 21(5):443–448.

Lakshmanan, K., de Niz, D., and Rajkumar, R. R. (2011). Mixed-criticality task synchronization in zero-slack scheduling. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Li, H. and Baruah, S. (2012). Global mixed-criticality scheduling on multiprocessors. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems (ECRTS)*.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

Liu, J. W. S. (2000). *Real-Time Systems*. Prentice-Hall Incorporated.

Liu, Y. and Mok, A. (2003). An integrated approach for applying dynamic voltage scaling to hard real-time systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Lopez, J. M., Diaz, J. L., and Garcia, D. F. (2004). Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, 28(1):39–68.

Mok, A. K. (1983). *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology. Available as Technical Report No. MIT/LCS/TR-297.

Oh, D.-I. and Baker, T. P. (1998). Utilization bounds for N-processor rate monotone scheduling with static processor assignment. *Real-Time Systems: The International Journal of Time-Critical Computing*, 15:183–192.

Park, T. and Kim, S. (2011). Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*.

Pathan, R. (2012). Schedulability analysis of mixed-criticality systems on multiprocessors. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems (ECRTS)*.

Raghavan, A., Luo, Y., Chandawalla, A., Papaefthymiou, M., Pipe, K. P., Wenisch, T. F., and Martin, M. M. K. (2012). Computational sprinting. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

Rotem, E., Naveh, A., Ananthakrishnan, A., Rajwan, D., and Weissmann, E. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Transactions on Micro*, 32(2):20–27.

Short, M. (2011). Improved schedulability analysis of implicit deadline tasks under limited preemption edf scheduling. In *Proceedings of the IEEE Conference on Emerging Technologies Factory Automation (ETFA)*.

Tamas-Selicean, D. and Pop, P. (2011). Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Wang, S. and Bettati, R. (2006a). Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Wang, S. and Bettati, R. (2006b). Reactive speed control in temperature-constrained real-time systems. In *Proceedings of the IEEE EuroMicro Conference on Real-Time Systems (ECRTS)*.

Wieder, A. and Brandenburg, B. (2013). On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*.

Yao, F., Demers, A., and Shenker, S. (1995). A scheduling model for reduced cpu energy. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*.