EDF Scheduling on Heterogeneous Multiprocessors

by
Shelby Hyatt Funk

A dissertation submitted to the faculty of 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
2004

Approved by:

Sanjoy K. Baruah, Advisor

James Anderson, Reader

Kevin Jeffay, Reader

Jane W. S. Liu, Reader

Montek Singh, Reader

Jack S. Snoeyink, Reader

ABSTRACT

SHELBY H. FUNK: EDF Scheduling on Heterogeneous Multiprocessors. (Under the direction of Sanjoy K. Baurah)

The goal of this dissertation is to expand the types of systems available for real-time applications. Specifically, this dissertation introduces tests that ensure jobs will meet their deadlines when scheduled on a uniform heterogeneous multiprocessor using the Earliest Deadline First (EDF) scheduling algorithm, in which jobs with earlier deadlines have higher priority. On uniform heterogeneous multiprocessors, each processor has a speed s, which is the amount of work that processor can complete in one unit of time. Multiprocessor scheduling algorithms can have several variations depending on whether jobs may migrate between processors — i.e., if a job that starts executing on one processor may move to another processor and continue executing. This dissertation considers three different migration strategies: full migration, partitioning, and restricted migration. The full migration strategy applies to all types of job sets. The partitioning and restricted migration strategies apply only to periodic tasks, which generate jobs at regular intervals. In the full migration strategy, jobs may migrate at any time provided a job never executes on two processors simultaneously. In the partitioning strategy, all jobs generated by a periodic task must execute on the same processor. In the restricted migration strategy, different jobs generated by a periodic task may execute on different processors, but each individual job can execute on only one processor. The thesis of this dissertation is

> *Schedulability tests exist for the Earliest Deadline First* (EDF) *scheduling algorithm on heterogeneous multiprocessors under different migration strategies including full migration, partitioning, and restricted migration. Furthermore, these tests have polynomial-time complexity as a function of the number of processors* (m) *and the number of periodic tasks* (n).
>
> - *The schedulability test with full migration requires two phases: an* $\mathcal{O}(m)$ *one-time calculation, and an* $\mathcal{O}(n)$ *calculation for each periodic task set.*
> - *The schedulability test with restricted migration requires an* $\mathcal{O}(m+n)$ *test for each multiprocessor / task set system.*
> - *The schedulability test with partitioning requires two phases: a one-time exponential calculation, and an* $\mathcal{O}(n)$ *calculation for each periodic task set.*

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Sanjoy K. Baruah. He has been a wonderful advisor and mentor. He and his wife, Maya Jerath, have both been good friends. In addition, I would like to thank my committee members Jim Anderson, Kevin Jeffay, Jane Liu, Montek Singh, and Jack Snoeyink. Each committee member contributed to my dissertation in different and valuable ways. I would also like to thank Joël Goossens, who I worked with very closely. He has been a good friend as well as colleague. I have also had the pleasure of working with a talented group of graduate students while at UNC. I would like to thank Anand Srinivasan, Phil Holman, Uma Devi, Aaron Block, Nathan Fisher, Vasile Bud, and Abishek Singh, who have all participated in real-time systems meetings.

It has been a pleasure to be a graduate student at the computer science department at UNC in large part because of the invaluable contributions of the staff. I thank each member of the administrative and technical staff for the countless ways they assisted me while I was a graduate student.

I have also had the pleasure of making wonderful friends while I was in the graduate department at UNC. These friends, both those in the department and those from elsewhere, provided me with much-needed relaxation during my graduate studies. I feel privileged to have had so much support.

Finally, I would like to thank my family. My mother, Harriet Fulbright, who was a great support during my entire time in graduate school. My grandparents, Brantz and Ana Mayor, who encouraged me to apply to graduate school in the first place. My sisters, Heidi Mayor and Evie Watts-Ward, who both are the best friends I could ask for. And Evie's family, James, Bo and Anna Ward, who have provided me a refuge in Chapel Hill.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A wide variety of applications use real-time systems: embedded systems such as cell phones, large and expensive systems such as power plant controllers, and safety critical systems such as avionics. Each of these applications have specific timing requirements, and violating the timing requirements may result in negative consequences. When timing requirements are violated in cell phones, calls could be dropped — if enough calls are dropped, the cell phone provider will lose customers. When timing requirements are violated in power plant controllers, the plant could overheat or even emit radioactive material into the surrounding area. When timing requirements are violated in avionics, an airplane could lose control, potentially causing a catastrophic crash.

In real-time systems, events must occur within a specified time frame, measured using "real" wall-clock time rather than some internal measure of time such as clock ticks or instruction cycles. Like all systems, real-time systems must maintain *logical* correctness – given a certain input, the system must generate the correct output. In addition, real-time systems must maintain *temporal* correctness – the output must be generated within the designated time frame.

Real-time systems are comprised of *jobs* and a *platform*. A job is a segment of code that can execute on a single processor for a finite amount of time. A platform is the processor or processors on which the jobs execute. When an application submits a job to a real-time system, the job specification includes a *deadline*. The deadline is the time at which the job should complete execution.

In *hard real-time systems*, all jobs must complete execution prior to their deadlines — a missed deadline constitutes a system failure. Such systems are used where the consequences of missing a deadline may be serious or even disastrous. Avionic devices and power plant controllers would both use hard real-time systems. In *soft real-time systems*, jobs may continue execution beyond their deadlines at some penalty — deadlines are considered guidelines, and the system tries to minimize the penalties associated with missing them. Such systems are used when the consequences of missing deadlines are smaller than the cost of meeting them in all possible circumstances (including the improbable and pathological). Cell phone and multimedia applications would both use soft real-time systems.

This dissertation introduces tests for ensuring that a hard real-time system will not fail due to a missed deadline. In hard real-time systems, we must be able to ensure *prior to execution* that a system will meet all of its deadlines during execution. We need tests that we can apply to the system that will guarantee that all deadlines will be met. This dissertation develops different tests for different types of systems. If a system does not pass its associated test, it will not be used for real-time applications.

This dissertation introduces several tests for hard real-time systems on multiprocessors using the *Earliest Deadline First* (EDF) scheduling algorithm, in which jobs with earlier deadlines have higher priority. The different tests depend on the parameters of the system. For example, we may be certain all deadlines are met on one multiprocessor, but we may be unable to make the same guarantee if the same jobs are scheduled a different multiprocessor.

All the tests presented in this dissertation apply to *uniform heterogeneous multiprocessors*, in which each processor has an associated *speed*. The speed of a processor equals the amount of work that processor can complete in one unit of time. Retailers currently offer uniform heterogeneous multiprocessors. For example, Dell offers several multiprocessors that allow processors to operate at different speeds. Until now, developers of real-time systems have not been able to analyze the behavior of real-time systems on uniform heterogeneous multiprocessors.

The remainder of this chapter is organized as follows: Section 1.1 introduces some basic real-time concepts. Section 1.2 introduces various multiprocessor models. This section describes the uniform heterogenous multiprocessor model in detail and explains its importance for real-time systems. Section 1.3 discusses multiprocessor scheduling algorithms. Section 1.4 introduces variations of EDF on uniform heterogeneous multiprocessors. Finally, Section 1.5 discusses this dissertation's contributions to real-time scheduling on heterogeneous multiprocessors in more detail.

1.1 Overview of real-time systems

A *real-time instance*, $I = \{J_1, J_2, \ldots, J_n, \ldots\}$, is a (possibly infinite) collection of time-constrained jobs. Each job $J_i \in I$ is described by a three-tuple (r_i, c_i, d_i), where r_i is the job's release time, c_i is its worst-case execution requirement (i.e., the maximum amount of time this job requires if it executes on a processor with speed equal to one), and d_i is its deadline. A job J_i is said to be *active* at time t if $t \geq r_i$ and J_i has not executed c_i units by time t.

In real-time systems, some jobs may repeat. For example, a system may need to read the ambient temperature at regular intervals. These infinitely-repeating jobs are generated by *periodic* or *sporadic tasks* [LL73], denoted $\tau = \{T_1, T_2, \ldots, T_n\}$. Each periodic task $T_i \in \tau$ is described by a three-tuple, (o_i, e_i, p_i), where o_i is the offset, e_i is the worst-case execution

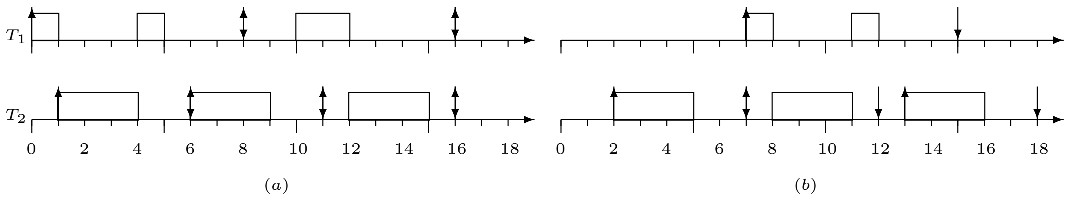


Figure 1.1: Task set $\tau = \{T_1 = (0, 2, 8), T_2 = (1, 3, 5)\}$ is a periodic task (a), and sporadic task (b). An up arrow indicates a new job arrival time. A rectangle indicates the job is executing. In periodic tasks, the period is the time that elapses between consecutive job arrivals. In sporadic tasks, the period is the minimum time that elapses between consecutive job arrivals.

requirement and p_i is the period: for each nonnegative integer k, task T_i generates a job $T_{i,k} = (r_{i,k}, e_i, d_{i,k})$ where $r_{i,k} = o_i + k \cdot p_i$ and $d_{i,k} = o_i + (k+1) \cdot p_i$. For sporadic tasks, the parameter p_i is the minimum *inter-arrival time* — i.e., the minimum time between consecutive job arrivals. Thus, the arrival time of $T_{i,k}$ is not known but it is bounded by the minimum separation: $r_{i,k+1} \geq r_{i,k} + p_i$. The arrival time of $T_{i,0}$ is bounded by the offset: $r_{i,0} \geq o_i$. The deadline of a sporadic task is always $d_{i,k} = r_{i,k} + p_i$.

Example 1.1 *Figure 1.1 illustrates the task set $\tau = \{T_1 = (0, 2, 8), T_2 = (1, 3, 5)\}$. In inset (a) of Figure 1.1, τ is a periodic task set, and in inset (b), it is a sporadic task set. In these diagrams, an upward arrow indicates a job arrival and a downward arrow indicates its deadline. A rectangle on the time line indicates that the task is executing during that interval. Notice that the periodic task's deadlines always coincide with the arrival time of the next job. Henceforth, the deadline indicators will be omitted in diagrams of periodic tasks. On the other hand, the sporadic task's deadlines do not necessarily coincide with the next arrival time — the next job can arrive at or after the previous deadline. Also, notice that in both figures the first job of task T_1 executes for one time unit, stops, and then restarts to execute for the final one time unit. The event is called a* preemption. *Throughout this dissertation, preemption is allowed.*

When analyzing a system, we need to know the requirements of each task — i.e., the amortized amount of processing time the task will need. We use a task's *utilization* to measure its processing requirement. The utilization of task T_i is the proportion of processing time the task will require if it is executed on a processor with speed equal to one: $u_i \overset{\text{def}}{=} \frac{e_i}{p_i}$. The *total utilization* of a periodic or sporadic task set, $U_{sum}(\tau) \overset{\text{def}}{=} \sum_{i=1}^{n} u_i$, measures the proportion of processor time the entire set will require.

Our goal is to develop tests that determine if a real time system will meet all its deadlines. We wish to develop tests that can be applied in polynomial time. We categorize periodic and sporadic task sets according to their utilization. We consider both the total utilization,

$U_{sum}(\tau) \overset{\text{def}}{=} \sum_{i=1}^{n} u_i$, and the maximum utilization, $u_{max}(\tau) \overset{\text{def}}{=} \max_{T_i \in \tau}\{u_i\}$. We group all task sets with maximum utilization u_{max} and total utilization U_{sum} into the same *class*, denoted $\Psi(u_{max}, U_{sum})$. Any task set $\tau \in \Psi(u_{max}, U_{sum})$ will miss deadlines if it is scheduled on a multiprocessor whose fastest processor speed is less than u_{max} or whose total processor speed is less than U_{sum}. However, we shall see that ensuring that $\tau \in \Psi(u_{max}, U_{sum})$ is not a sufficient test for guaranteeing all deadlines will be met.

A real-time instance is called *feasible* on a processing platform if it is possible to schedule all the jobs without missing any deadlines. A specific schedule is called *valid* if all jobs complete execution at or before their deadlines. If a particular algorithm A always generates a valid schedule when scheduling the real-time instance I on a platform π, then I is said to be A-*schedulable* on π. The feasibility and schedulability of I depend on the processing platform under consideration. The next section examines various types of processing platforms.

First, some additional notation.

Definition 1 ($\delta(\mathsf{A}, \pi, s_i, I, J, t)$) *Let I be any real-time instance, J be a job of I, and $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor. For any algorithm A and time instant $t \geq 0$, let $\delta(\mathsf{A}, \pi, s_i, I, J, t)$ indicate whether or not J is executing on processor s_i of π at time t when scheduled using A. Specifically,*

$$\delta(\mathsf{A}, \pi, s_i, I, J, t) \overset{def}{=} \begin{cases} 1 & \textit{if } \mathsf{A} \textit{ schedules } J \textit{ to execute on } s_i \textit{ at time } t \\ 0 & \textit{otherwise.} \end{cases}$$

The function δ can be used to determine all the work performed by algorithm A on a given job or on the entire instance.

Definition 2 ($W(\mathsf{A}, \pi, I, J, t), W(\mathsf{A}, \pi, I, t)$) *Let J be a job of a real-time instance I and let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor. For any algorithm A and time instant $t \geq 0$, let $W(\mathsf{A}, \pi, I, J, t)$ denote the amount of work done by algorithm A on job J over the interval $[0, t)$, while executing on π and let $W(\mathsf{A}, \pi, I, t)$ denote the amount of work done on all jobs of I. Specifically,*

$$\begin{aligned} W(\mathsf{A}, \pi, I, J, t) &\overset{def}{=} \sum_{i=1}^{m} \left(s_i \times \int_0^t \delta(\mathsf{A}, \pi, s_i, I, J, x)dx \right) \textit{ and} \\ W(\mathsf{A}, \pi, I, t) &\overset{def}{=} \sum_{J \in I} W(\mathsf{A}, \pi, I, J, t). \end{aligned}$$

1.2 A taxonomy of multiprocessors

A *real-time system* is a real-time instance paired with a specific computer processing platform. The platform may be a *uniprocessor*, consisting of one processor, or it may be a *multiprocessor*, consisting of several processors. If the platform is a multiprocessor, the individual processors may all be the same or they may differ from one another. We divide multiprocessors into three different categories based on the speeds of the individual processors.

- **Unrelated heterogeneous multiprocessors.** In these platforms, the processing speed depends not only on the processor, but also on the job being executed. For example, if one of the processors is a graphics coprocessor, graphics jobs would execute at a more accelerated rate than non-graphics jobs. Each (processor, job)-pair of an unrelated heterogeneous system has an associated speed $s_{i,j}$, which is the amount of work completed when job j executes on processor i for one unit of time.

- **Uniform heterogeneous multiprocessors.** In these platforms, the processing speed depends *only* on the processor. Specifically, for each processor i and for all pairs of jobs j and k, we have $s_{i,j} = s_{i,k}$. In these multiprocessors, we use a s_i to denote the speed of the i'th processor.

- **Identical multiprocessors.** In these platforms, all processing speeds are the same. In these systems, the speed is usually normalized to one unit of work per unit of time.

Until recently, research in real-time scheduling on multiprocessors has concentrated on identical multiprocessors. The research presented in this dissertation concentrates on uniform heterogeneous multiprocessors, which are a relevant platform for modelling many real-time applications:

- These multiprocessors give system designers more freedom to tailor the platform to the application requirements. For example, if a system is comprised of a few tasks with large utilization values and several tasks with much smaller utilization values, the designer may choose to use a multiprocessor with one very fast processor to ensure the higher-utilization tasks meet their deadlines and several slower processors to execute the lower-utilization tasks.

- If a platform is upgraded, either by adding processors or by enhancing currently existing processors, the resulting platform may be comprised of processors that execute at different speeds. If only identical multiprocessors are available, all processors must be upgraded simultaneously. Similarly, when adding processors, slower processors would have to be added even if faster ones were available and affordable.

- Unrelated heterogeneous multiprocessors are a generalization of uniform heterogeneous multiprocessors, which are in turn a generalization of identical multiprocessors. Analysis of uniform heterogeneous multiprocessors gives us a deeper understanding of both of the other types of multiprocessors. Specifically, any property that does not hold for uniform heterogeneous multiprocessors will not hold for unrelated heterogeneous multiprocessors. Any property that *does* hold for uniform heterogeneous multiprocessors will also hold for identical multiprocessors.

We use the following notation to describe uniform heterogeneous multiprocessors.

Definition 3 *Let $\pi = [s_1, s_2, \ldots, s_m]$ denote an m-processor uniform multiprocessor with the i^{th} processor having speed s_i, where $s_i \geq s_{i+1}$ for $i = 1, \ldots, m-1$. The following notation is used to describe parameters of π. (When the processor under consideration is unambiguous, the (π) may be removed from the notation.)*

$m(\pi)$**:** *the number of processors in π .*

$s_i(\pi)$**:** *the speed of the i^{th} fastest processor of π .*

$S_i(\pi)$**:** *the cumulative processing power of the i fastest processors of π, $S_i(\pi) \overset{def}{=} \sum_{k=1}^{i} s_k(\pi)$.*

$S(\pi)$**:** *the cumulative processing power of all processors of π, $S(\pi) \overset{def}{=} \sum_{k=1}^{m} s_k(\pi)$.*

$\lambda(\pi)$**:** *the "identicalness" of π, $\lambda(\pi) \overset{def}{=} \max_{1 \leq k < m} \frac{\sum_{i=k+1}^{m} s_i(\pi)}{s_k(\pi)}$.*

Intuitively, $\lambda(\pi)$ enumerates the level to which π diverges from being an identical multiprocessor. The "less identical" π is, the greater the likelihood that an active job whose deadline is approaching can move to a faster processor. The value of λ increases as π becomes more identical. In the extreme, π is an identical multiprocessor and no faster processors exist for a job with an approaching deadline to move to. On identical multiprocessors, λ is maximized and has the value $(m-1)$. In the opposite extreme, the value of λ can be arbitrarily small when the processors have extremely different speeds. For example, $\lambda < \epsilon$ for the m-processor multiprocessor where $s_{i+1} \leq \frac{\epsilon}{m} \cdot s_i$ for $i = 1, \ldots, m-1$. The value of λ is used to assess the penalty associated with using Earliest Deadline First (EDF), a non-optimal algorithm. As the following example illustrates, it is not enough to know the total speed when analyzing whether a real time instance will meet all deadlines. We must also know the relative speeds of the different processors, which is partially captured by the parameter $\lambda(\pi)$.

Example 1.2 *Consider two multiprocessors $\pi_1 = [6, 2]$ and $\pi_2 = [5, 3]$. Then $S(\pi_1) = S(\pi_2) = 8$. The identicalness parameters associated with π_1 and π_2 are $\frac{1}{3}$ and $\frac{3}{5}$, respectively.*

Assume we want to schedule two jobs, $J_1 = (0, 30, 6)$ and $J_2 = (0, 34, 9)$, using the EDF *scheduling algorithm and we allow migration. Inset (a) of Figure 1.2 illustrates the schedule of these jobs on π_1 and inset (b) illustrates the schedule on π_2. Initially, J_1 executes on the faster processor since it has the earlier deadline, and J_2 executes on the slower processor. Once J_1 completes executing, after 5 time units on π_1 and 6 time units on π_2, job J_2 can migrate to the faster processor. Notice that when these jobs execute on π_1, job J_1 completes executing at $t = 5$ and job J_2 completes executing at $t = 9$. By contrast, on π_2, job J_1 completes executing at $t = 6$ and job J_2 completes executing at $t = 9.2$. Thus, both jobs meet their deadlines on π_1, but it is impossible for both jobs to meet their deadlines on π_2 since J_1* must *execute on the $s_1(\pi_2)$ during the entire interval $[0, 6)$ in order to meet its deadline, but J_2 misses its deadline if it can't migrate to $s_1(\pi_2)$ before $t = 6$.*

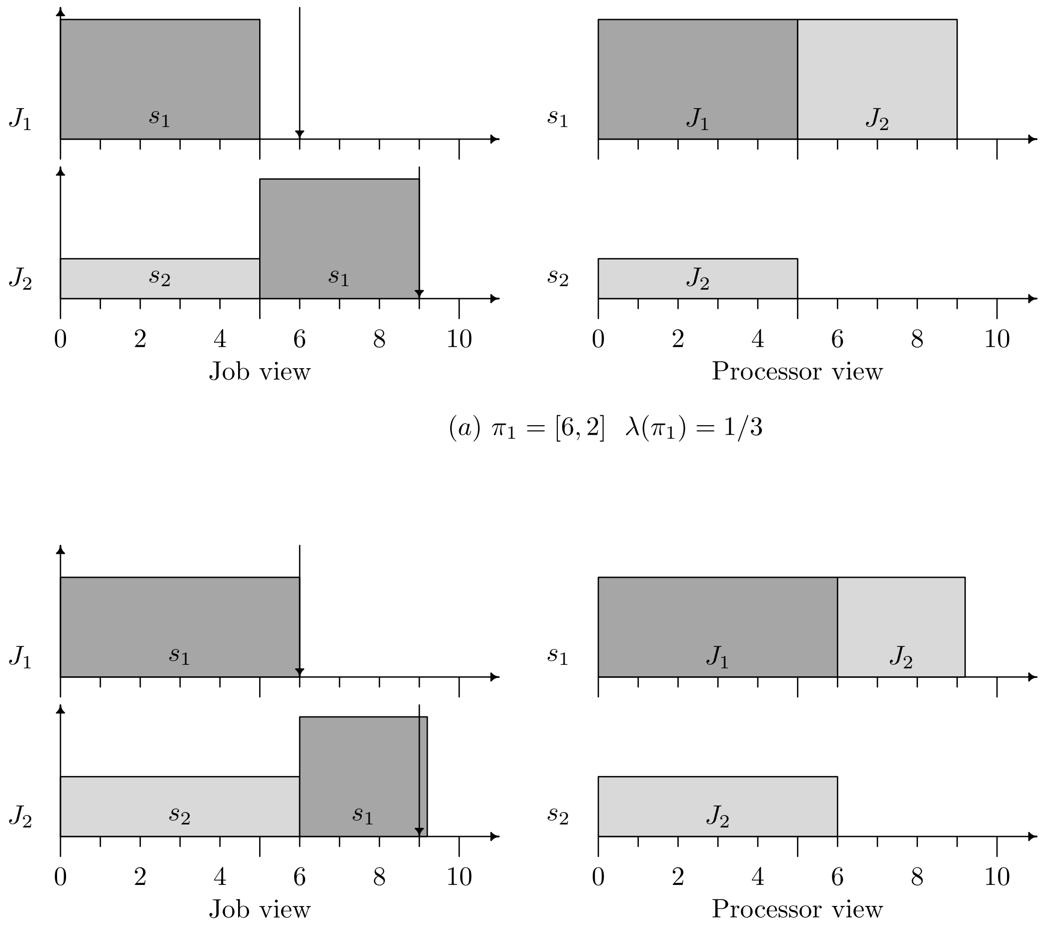
1.3 Multiprocessor scheduling algorithms

We have seen that analysis of real-time systems depend on the processing platform. In this section, we will see that analysis of real-time systems also depends on the scheduling algorithm — i.e., the method used to determine when each job executes and on which processor. This section begins by discussing several important properties of scheduling algorithms.

In *offline* scheduling algorithms, all scheduling decisions are made before the system begins executing. These scheduling algorithms select jobs to execute by referencing a table describing the pre-determined schedule. Usually, offline schedules are repeated after a specific time period. For example, if the jobs being scheduled are generated by periodic tasks, an offline schedule may be generated for an interval of length equal to the least common multiple of the periods of the tasks in the task set — after this period of time, the arrival pattern of the jobs will repeat. When the schedule reaches the end of the pre-determined table, it can simply return to the beginning of the table. This type of scheduler is often called a *cyclic executive.*

In *online* scheduling algorithms, all scheduling decisions are made without specific knowledge of jobs that have not yet arrived. These scheduling algorithms select jobs to execute by examining properties of active jobs. Online algorithms can be more flexible than offline algorithms since they can schedule jobs whose behavior cannot be predicted ahead of time. For example, the system must use an online scheduling algorithm if the task set includes sporadic tasks whose inter-arrival times are unpredictable or dynamic tasks that join and leave the system at undetermined times.

Both online and offline algorithms may be *work conserving*. A scheduling algorithm is work conserving if processors can idle only when there are no jobs waiting to execute. For uniform heterogeneous multiprocessors, work conserving scheduling algorithms must also take full advantage of the faster processor speeds. In particular, a uniform heterogeneous multiprocessor scheduling algorithm A is said to be work-conserving if and only if it satisfies



(a) $\pi_1 = [6, 2]$ $\lambda(\pi_1) = 1/3$

(b) $\pi_2 = [5, 3]$ $\lambda(\pi_2) = 3/5$

Figure 1.2: The importance of $\lambda(\pi)$. The total speed of each of these two multiprocessors equals 8. The jobs meet their deadlines when scheduled on π_1, but J_2 misses its deadline when these jobs are scheduled on π_2, whose λ-value is larger.

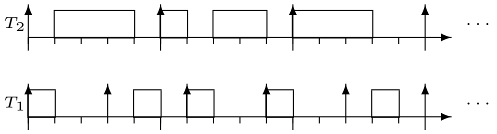


Figure 1.3: Task set $\tau = \{T_1 = (0, 1, 3), T_2 = (0, 3, 5)\}$ scheduled on a unit speed uniprocessor using EDF. Observe that initially, task T_1 has higher priority than task T_2. However, when T_1 generates a new job at $t = 3$, task T_1 has lower priority than task T_2.

the following conditions.

- No processor idles while there are active jobs waiting to execute.
- If at some instant there are fewer than m active jobs waiting to execute, where m denotes the number of processors in the multiprocessor, then these jobs execute upon the fastest processors. That is, at any instant t if A idles the j^{th}-slowest processor, then A idles the k^{th} slowest processors for all $k > j$.

In *fixed priority* scheduling algorithms, jobs generated by the same task all have the same priority. More formally, if $T_{i,k}$ has higher priority than $T_{j,\ell}$ then $T_{i,r}$ has higher priority than $T_{j,s}$ for all values of r and s. These are also called *static priority* algorithms. One very well-known fixed priority scheduling algorithm is the Rate Monotonic (RM) algorithm [LL73]. In this algorithm, the task period is used to determine priority — tasks with shorter periods have higher priority. This algorithm is known to be optimal among uniprocessor fixed-priority algorithms [LL73] — i.e., if it is possible for all jobs to meet their deadlines using a fixed priority algorithm, then they will meet their deadlines when scheduled using RM.

In *dynamic priority* scheduling algorithms, jobs generated by the same task may have different priorities. The Earliest Deadline First (EDF) algorithm [LL73] is a well-known dynamic priority algorithm. The EDF scheduling algorithm is optimal among all uniprocessor scheduling algorithms — if it is possible for all jobs to meet their deadlines, they will do so when scheduled using EDF. This algorithm is illustrated in Example 1.3.

Example 1.3 *Figure 1.3 illustrates an* EDF *schedule of the task set* $\tau = \{T_1 = (0, 1, 3), T_2 = (0, 3, 5)\}$ *on a uniprocessor of speed 1. Notice that* $T_{1,1}$ *(the first job of task* T_1*) has a deadline of 3 and has higher priority than* $T_{2,1}$*. However,* $T_{1,2}$ *has* lower *priority than* $T_{2,1}$ *since* $d_{1,2} = 6$ *and* $d_{2,1} = 5$*.*

Dynamic priority algorithms can be divided into two categories depending on whether individual *jobs* can change priority while they are active. In *job-level fixed-priority* algorithms, jobs cannot change priorities. EDF is a job-level fixed-priority algorithm. On the other

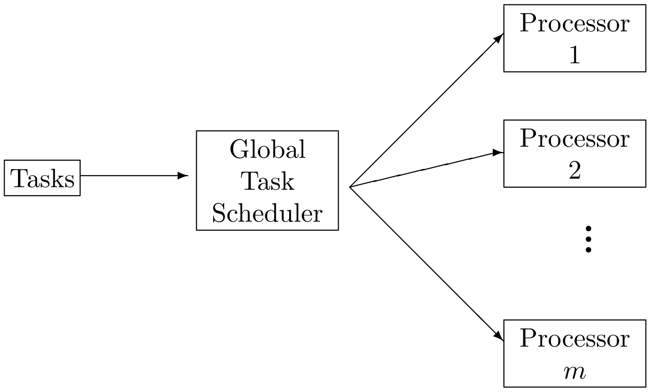


Figure 1.4: Scheduling with full migration uses a global scheduler. Tasks generate jobs and submit them to the global scheduler, which monitors both when and where all jobs will execute.

hand, in *job-level dynamic-priority* algorithms, jobs may change priority during execution. For example, the Least Laxity First (LLF) algorithm [LL73] is a job-level dynamic-priority algorithm. At time t, the laxity of a job is $(d - t - f)$, where d is the job's deadline and f is it's remaining execution requirement. Intuitively, the laxity is the maximum amount of time a job may be forced to wait if it were to execute on a processor of speed 1 and still meet its deadline. The LLF algorithm assigns higher priority to jobs with smaller laxity. Since the laxity of a job can change over time, the job priorities can change dynamically.

Finally, an algorithm is *optimal* if it can successfully schedule any feasible system. For example, EDF is optimal on uniprocessors [LL73, Der74]. While RM is not an optimal algorithm, it is optimal on uniprocessors among fixed priority algorithms [LL73] — i.e., if it is possible for a task set to meet all deadlines using a fixed priority algorithm then that task set is RM-schedulable. Uniprocessor systems that allow dynamic-priority scheduling will commonly use the EDF scheduling algorithm, while systems that can only use fixed-priority scheduling algorithms will use the RM scheduling algorithm.

On multiprocessors, scheduling algorithms can be divided into various categories depending on the amount of *migration* the system allows. A job or task migrates if it begins execution on one processor and is later interrupted and restarts on a different processor. This dissertation considers three types of migration strategies [CFH$^+$03].

Full migration. Jobs may migrate at any point during their execution. All jobs are permitted to execute on any processor of the system. However, a job can only execute on at most one processor at a time — i.e., job parallelism is not permitted. Figure 1.4 illustrates a full migration scheduler.

No migration (partitioning). Tasks can never migrate. Each task in a task set is assigned

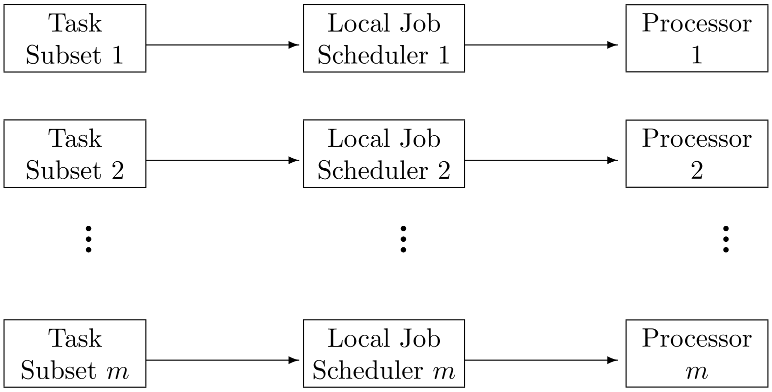


Figure 1.5: Scheduling with no migration uses a partitioned scheduler. Tasks generate jobs and submit them to the local scheduler for the processor to which the task is assigned. Every job generated by a task executes on the same processor.

to a specific processor. All jobs generated by a task can execute only on the processor to which the task is assigned. Figure 1.5 illustrates an partitioned scheduler.

Restricted migration. Task can migrate only at job boundaries. When a task generates a job, the global scheduler assigns the job to a processor, and that job can execute only on the processor to which it is assigned. However, the next job of the same task can execute on any processor. Figure 1.6 illustrates a restricted migration scheduler.

While the full migration strategy is the most flexible, there are clearly overheads associated with allowing migration. On the other hand, there are also overheads associated with not migrating jobs. Prohibiting migration may cause a system to be under-utilized to ensure enough processing power will be available on some processor when a new job arrives. If migration is allowed, the job can execute for a time on one processor and then move to another processor, allowing the spare processing power to be distributed among all the processors. Thus, there is a trade-off between scheduling loss due to migration and scheduling loss due to prohibiting migration. In some systems, we may prefer to migrate jobs and in others we may need a more restrictive strategy.

Systems that do not allow jobs to migrate must use either the partitioning or the restricted migration strategy. Between these two, the partitioning strategy is more commonly used in current systems. However, partitioning can only be used for fixed task sets. If tasks are allowed to dynamically join and leave the system, partitioning is not a viable strategy because a task joining the system may force the system to be repartitioned, thus forcing tasks to migrate. Determining a new partition is analogous to the *bin-packing problem*, which is known to be NP-hard [Joh73]. Thus, repartitioning dynamic task sets incurs too much

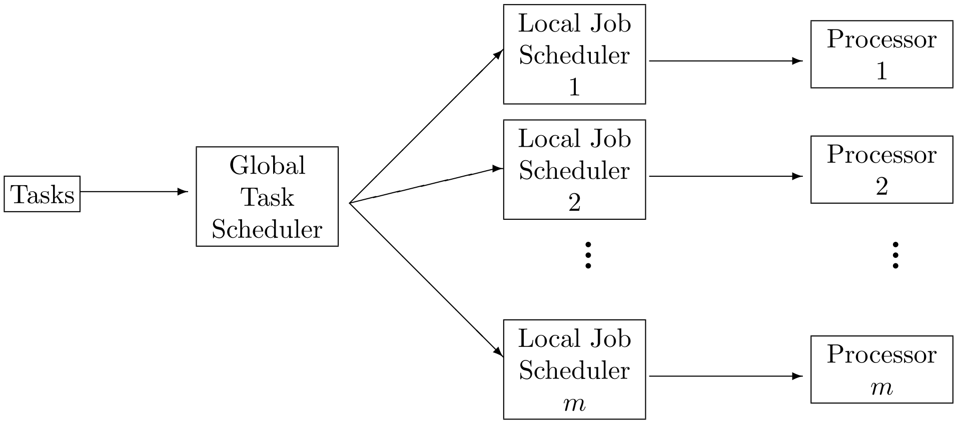


Figure 1.6: Scheduling with restricted migration uses both a global scheduler and a partitioned scheduler. Tasks generate jobs and submit them to the global scheduler. The global scheduler assigns the job to a processor and the local scheduler for that processor determines when the jobs executes. Different jobs generated by a task may execute on the different processors.

overhead.

The restricted migration strategy provides a good compromise between the full migration and the partitioning strategies. It is flexible enough to allow for dynamic task sets, but it doesn't incur large migration overheads. This strategy is particularly useful when consecutive jobs of a task do not share any data since all data is passed to subsequent jobs would have to be migrated even at job boundaries. Furthermore, the restricted-migration global scheduler is much simpler than the full-migration global scheduler. The full migration global scheduler needs to maintain information about all active jobs in the system, whereas the restricted migration global scheduler makes a single decision about a job when it arrives and then passes the job to a local scheduler that maintains information about the job from that point forward. However, this flexibility comes at a cost: Chapter 5 of this dissertation will show that any task set that is guaranteed to meet all deadlines using EDF with restricted migration would also meet all deadlines if scheduled using either of the other two migration strategies.

If we have a real-time instance I that we know is feasible on a uniprocessor, we can schedule I using an optimal online scheduling algorithm such as EDF. Thus, in order to determine whether I is EDF-schedulable on a uniprocessor, it suffices to determine whether I is feasible on the uniprocessor. Unfortunately, it has been shown that there are *no* optimal job-level fixed-priority scheduling algorithms for multiprocessors [HL88, DM89]. Since EDF is a job-level fixed-priority scheduling algorithm for multiprocessors, determining whether I is feasible on a multiprocessor π will not tell us whether I is EDF-schedulable on π. Instead, we have to use other means to determine EDF-schedulability.

In [HL88, DM89], the authors actually state that there is no optimal online scheduling algorithm for multiprocessors. However, the proof considered only job-level fixed-priority algorithms. Baruah, et al.[BCPV96], proved that there is a job-level dynamic-priority scheduling algorithm that is optimal for periodic task sets on multiprocessors. Srinivasan and Anderson [SA] later showed that this algorithm can be modified to be optimal for sporadic task sets. These results do not apply to EDF because they use a job-level dynamic-priority algorithm.

1.4 EDF on uniform heterogeneous multiprocessors

The EDF scheduling algorithm has been shown to be optimal for uniprocessors [LL73]. However, since it is an online job-level fixed-priority algorithm, we know that EDF cannot be optimal for multiprocessors. Nonetheless, there are still many compelling reasons for using EDF when scheduling on multiprocessors.

- Since EDF is an optimal uniprocessor scheduling algorithm, all local scheduling is done using an optimal algorithm. This is particularly relevant for the partitioning and restricted migration strategies since many of the scheduling decisions are made locally in these strategies.
- Efficient implementations of EDF have been designed [Mok88].
- The number of preemptions and migrations incurred by EDF can be bounded. (The bounds depend on which migration strategy is being used.) Since migration and preemption both incur overheads, it is important to be able to incorporate the overheads into any system analysis. This can only be done if the associated overheads can be bounded[1].

On uniprocessors, EDF is well defined — at all times, the job with the earliest deadline executes on the sole processor. When more processors are added to the system, there are several variations of EDF depending on the chosen migration strategy. This dissertation will analyze three variation of EDF — one for each migration strategy discussed on page 10.

Full migration EDF (f-EDF). This algorithm uses full migration, as illustrated in Figure 1.4, with the global scheduler giving higher priority to jobs with earlier deadlines. Moreover, deadlines not only determine *which* jobs execute, but also *where* they execute — the earlier the deadline, the faster the processor. Figure 1.7 illustrates an f-EDF schedule of the periodic task set $\tau = \{T_1 = (1, 2, 3), T_2 = (1, 3, 4), T_3 = (0, 6, 8)\}$ on the multiprocessor $\pi = [2, 1]$ (i.e., $s_1 = 2$ and $s_2 = 1$). In this diagram, the height of the rectangle indicates the processor speed. When a job executes on s_1, the corresponding rectangle

[1] This dissertation assumes that the overheads associated with both preemptions and migrations are included in the worst-case execution requirements.

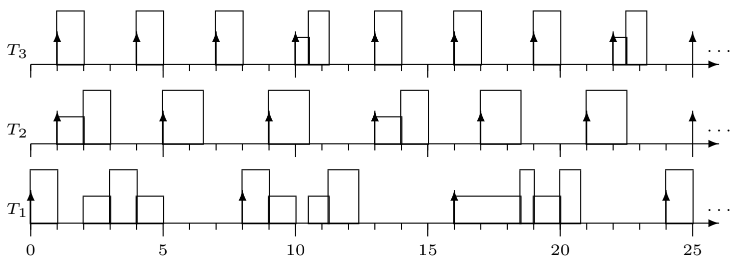


Figure 1.7: Task set $\tau = \{T_1 = (1,2,3), T_2 = (1,3,4), T_3 = (0,6,8)\}$ scheduled on $\pi = [2,1]$ using EDF with full migration (f-EDF).

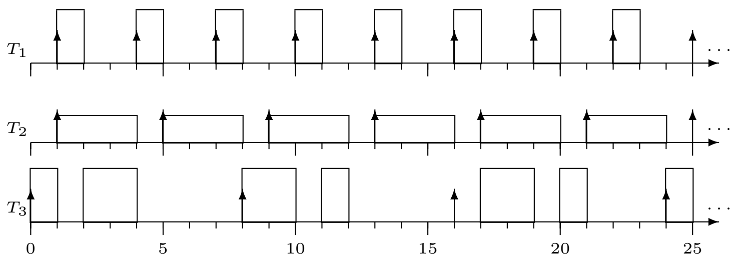


Figure 1.8: Task set $\tau = \{T_1 = (1,2,3), T_2 = (1,3,4), T_3 = (0,6,8)\}$ scheduled on $\pi = [2,1]$ using partitioned EDF (p-EDF).

is twice as high as when it executes on s_2. Thus, the total *area* of the rectangles equals the execution requirement.

Partitioned EDF (p-EDF). This algorithm uses partitioning, as illustrated in Figure 1.5, with each local scheduler using uniprocessor EDF. A task with utilization u can be assigned to a speed-s processor if and only if the total utilization of all tasks assigned to that processor is at most $s-u$. For this reason, we often refer to a processor's total speed as that processor's *capacity*. Figure 1.8 illustrates a p-EDF schedule of the periodic task set $\tau = \{T_1 = (1,2,3), T_2 = (1,3,4), T_3 = (0,6,8)\}$ on the multiprocessor $\pi = [2,1]$ with tasks T_1 and T_3 assigned to processor s_1 and task T_2 assigned to processor s_2.

Restricted migration EDF (r-EDF). This algorithm uses restricted migration, as illustrated in Figure 1.6, with each local scheduler using uniprocessor EDF. The global scheduler assigns each newly arrived job to any processor with enough available capacity to guarantee all deadlines will still be met after adding the job to the scheduling queue. Each of the local schedulers use uniprocessor EDF. Figure 1.9 illustrates an r-EDF schedule of the periodic task set $\tau = \{T_1 = (1,2,3), T_2 = (1,3,4), T_3 = (0,6,8)\}$ on the multiprocessor $\pi = [2,1]$.

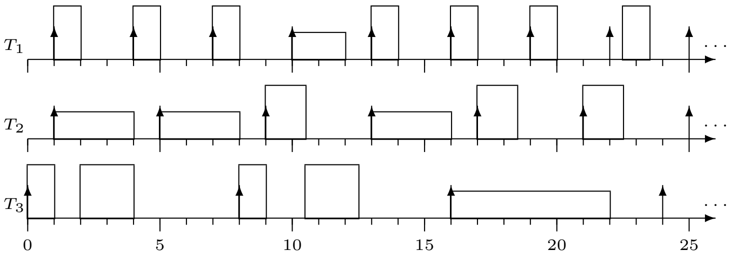


Figure 1.9: Task set $\tau = \{T_1 = (1,2,3), T_2 = (1,3,4), T_3 = (0,6,8)\}$ scheduled on $\pi = [2,1]$ using EDF with restricted migration (r-EDF).

All three variations of EDF are online algorithms since they consider only the deadlines of currently active jobs when making scheduling decisions. However, only f-EDF is a work conserving algorithm. Both of the other two algorithms may force a job to wait to execute even if there is an idling processor. For example, in Figure 1.9, T_1 does not execute during the interval $[22, 22.5)$ even though processor s_2 is idling during that time. Nonetheless, on a local level both p-EDF and r-EDF are work conserving — a processor will not idle if there is an active job *assigned to that processor* that is waiting to execute.

1.5 Contributions

The thesis for my work is as follows:

> *Schedulability tests exist for the Earliest Deadline First* (EDF) *scheduling algorithm on heterogeneous multiprocessors under different migration strategies including full migration, partitioning, and restricted migration. Furthermore, these tests have polynomial-time complexity as a function of the number of processors* (m) *and the number of periodic tasks* (n).
>
> - *The schedulability test with full migration requires two phases: an* $\mathcal{O}(m)$ *one-time calculation, and an* $\mathcal{O}(n)$ *calculation for each periodic task set.*
> - *The schedulability test with restricted migration requires an* $\mathcal{O}(m+n)$ *test for each multiprocessor / task set system.*
> - *The schedulability test with partitioning requires two phases: a one-time exponential calculation, and an* $\mathcal{O}(n)$ *calculation for each periodic task set.*

All of the schedulability tests for task sets presented in this dissertation are expressed in the following form.

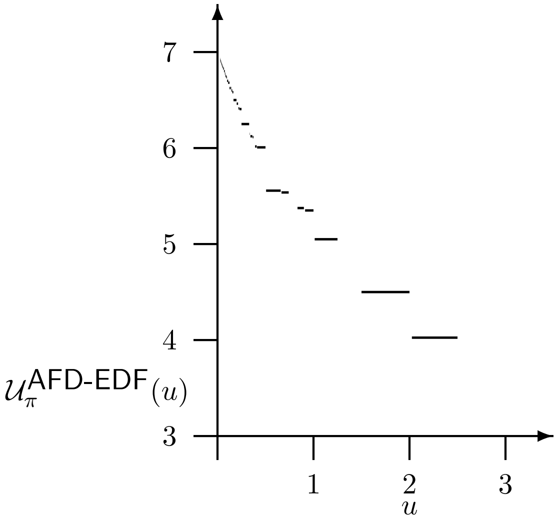


Figure 1.10: The graph of $\mathcal{U}_{\pi}^{\mathsf{AFD\text{-}EDF}}(u)$ for $\pi = [2.5, 2, 1.5, 1]$ with error bound $\epsilon = 0.1$. Any task is guaranteed to be p-EDF-schedulable if its class is below the illustrated graph.

> Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor. If $U_{sum} \leq \mathcal{U}_M(\pi, u_{max})$, then every task set τ in the class $\Psi(u_{max}, U_{sum})$ is guaranteed to be EDF-schedulable on π using migration strategy M.

For all three migration strategies, $\mathcal{U}_M$ can be graphed by drawing one or more lines on the (u_{max}, U_{sum}) plane. Any task set τ whose class falls between the lines defined by $\mathcal{U}_M(\pi, u_{max}(\tau))$ and the line $U_{sum} = u_{max}$ is guaranteed to be EDF-schedulable on π using the migration strategy corresponding to the graph.

Example 1.4 *Figure 1.10 shows an approximation of the graph of $f_{partition}$ for the multiprocessor $\pi = [2.5, 2, 1.5, 1]$. The point (1,4) is below the graph in this figure. This means that every task set τ with $u_{max}(\tau) = 1$ and $U_{sum}(\tau) = 4$ is p-EDF-schedulable on π. For example, the task set τ containing four tasks each with utilization equal to one can be successfully partitioned onto π.*

Even though the three schedulability tests will seem quite similar, they were developed using different methods.

- Determining the full migration test uses *resource augmentation* [KP95, PSTW97] methods and exploits the *robustness* [BFG03] of f-EDF on uniform heterogeneous multiprocessors. Given a task set τ, the resource augmentation method finds a slower multiprocessor π' on which τ is *feasible*. If π has "enough extra speed," then τ is guaranteed to be f-EDF-schedulable on π. The amount of extra speed that must be added to π' to ensure f-EDF-schedulability on π depends on the parameters of π and the maximum utilization, u_{max}. Resource augmentation analysis finds many, but not all, of the classes of task sets that are guaranteed to be f-EDF-schedulable on π. More classes can be found by applying the same test to "sub-platforms" π'' of π — multiprocessors whose processor speeds are all slower than the speed of the corresponding processors of π. For example, the multiprocessor $\pi'' = [50, 11, 3]$ is a "sub-platform" of $\pi = [50, 11, 4, 4]$. f-EDF was shown to be robust on uniform heterogeneous multiprocessors, meaning that anything f-EDF-schedulable on π'' is also f-EDF-schedulable on π.

- Determining the partitioning test uses approximation methods. Given any ϵ, this test will find all classes of task sets within ϵ of the actual utilization bound. Since partitioning is NP-complete in the strong sense [Joh73], it is particularly difficult to determine the utilization bound. Therefore, the p-EDF-schedulability test is an approximation of the actual utilization bound. The approximate utilization bound is found by an exhaustive search of the space of all task sets. For any u_{max}, all possible task sets are searched to find the task set τ with $u_{\max}(\tau) \leq u_{max}$ such that τ is *almost infeasible* — i.e., adding any capacity to τ will cause some deadline to be missed. A variety of methods are used to reduce the number of task sets that must be considered while still finding an approximation that is within ϵ of the actual bound. The number of points considered in the search grows in proportion to $(1/\epsilon)\log(1/\epsilon)$ — the smaller the value of ϵ, the more points and hence the longer it takes to complete the search.

- Determining the restricted migration test uses worst-case arrival pattern analysis methods — i.e., finding the pattern of job arrivals that would be most likely to cause a job to miss its deadline. Once this pattern is identified, the utilization bound can be established. In some cases this bound can be too restrictive. This is particularly true for task sets whose maximum utilization is significantly larger than their average utilization. This dissertation introduces a variation of r-EDF that restricts tasks to a subset of the processors of π without imposing a full partitioning strategy. These variations are intended to be used on systems with a few high-utilization tasks and several low-utilization tasks.

1.6 Organization of this document

The remainder of this dissertation is organized as follows. Chapter 2 discusses previous results that pertain to the research presented in this dissertation. Chapters 3, 4, and 5 present the uniform heterogeneous multiprocessor schedulability tests for the f-EDF p-EDF and r-EDF scheduling algorithms, respectively. Finally, Chapter 6 provides some concluding remarks.

Chapter 2

Background and related work

The real-time community has actively researched multiprocessor scheduling for over twenty years. This chapter presents several results that have some bearing on EDF scheduling on uniform heterogeneous multiprocessors. It is divided into four sections. Sections 2.1 and 2.2 present results pertaining to scheduling on identical and uniform heterogeneous multiprocessors, respectively. Section 2.3 presents architectural issues that arise when using uniform heterogeneous multiprocessors, and Section 2.4 provides some concluding remarks.

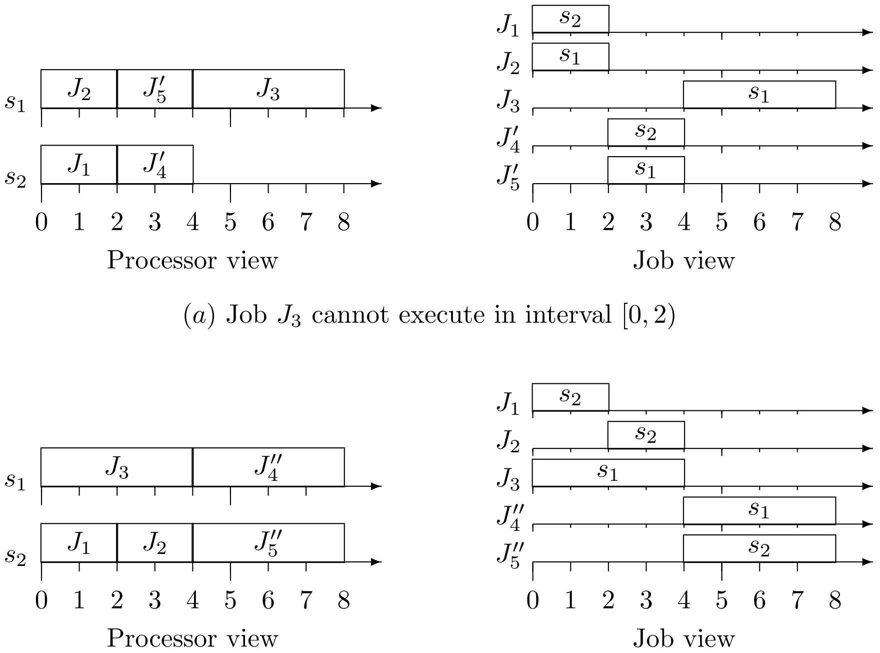
2.1 Results for identical multiprocessors

Much of the research on multiprocessor real-time scheduling has focussed on identical multiprocessors. This section presents a few important results. We first present general results pertaining to any online multiprocessor scheduling algorithm. Next, we present resource augmentation, which can be used to address some of the shortcomings that arise from using online algorithms. Also, we discuss a utilization bound that ensures partitioned EDF-schedulability on identical multiprocessors. Finally, we introduce an important property called predictability.

2.1.1 Online scheduling on multiprocessors

Hong and Leung [HL88] and Dertouzos and Mok [DM89] independently proved that there can be no optimal online algorithm for scheduling real-time instances on identical multiprocessors. Later, Baruah, et al.[BCPV96], proved that there is a job-level dynamic-priority scheduling algorithm called Pfair that is optimal for periodic task sets on multiprocessors. Srinivasan and Anderson [SA] later showed that this algorithm can be modified to be optimal for sporadic task sets. In this section, we will examine the work that applies to general real-time instances, beginning with the result developed by Hong and Leung.

Theorem 1 ([HL88]) *No optimal online scheduler can exist for instances with two or more distinct deadlines for any m-processor identical multiprocessor, where $m > 1$.*



(a) Job J_3 cannot execute in interval $[0, 2)$

(b) Job J_3 must execute in interval $[0, 2)$

Figure 2.1: No multiprocessor online algorithm can be optimal.

Hong and Leung proved this theorem with the counterexample that follows.

Example 2.1 ([HL88]) *Consider executing instances on a two-processor identical multiprocessor. Let* $I = \{J_1 = J_2 = (0, 2, 4), J_3 = (0, 4, 8)\}$*. Construct* I' *and* I'' *by adding jobs to* I *with later arrival times as follows:* $I' = I \cup \{J_4' = J_5' = (2, 2, 4)\}$ *and* $I'' = I \cup \{J_4'' = J_5'' = (4, 4, 8)\}$*. There are two possibilities depending on the behavior of* J_3*.*

Case 1: J_3 ***executes during the interval [0,2).*** *Then one of the jobs of* I' *will miss a deadline. Inset* (a) *of Figure 2.1 illustrates a valid schedule of* I' *on two unit-speed processors. Notice that the processors execute the jobs* J_1*,* J_2*,* J_4' *and* J_5' *and never idle during the interval* $[0, 4)$*. Moreover, these four jobs all have the same deadline at* $t = 4$*. Therefore, if* J_3 *were to execute for any time at all during this interval, it would cause at least one of the jobs to miss its deadline.*

Case 2: J_3 ***does not execute during the interval [0,2).*** *Then one of the jobs of* I'' *will miss a deadline. Inset* (b) *of Figure 2.1 illustrates a valid schedule of* I'' *on two unit-speed processors. Notice that the processors execute the jobs* J_1*,* J_2*, and* J_3 *during the interval* $[0, 4)$ *and all three jobs have completed execution by time* $t = 4$*. Moreover, jobs* J_4'' *and* J_5'' *both require four units of processing time in the interval* $[4, 8)$*. If job* J_3 *did not execute during the entire interval* $[0, 2)$*, it would not complete execution by time* $t = 4$*. Therefore, it would require processing time in the interval* $[4, 8)$ *and at least one of the jobs* J_3*,* J_4''*, or* J_5'' *would* *miss its deadline.*

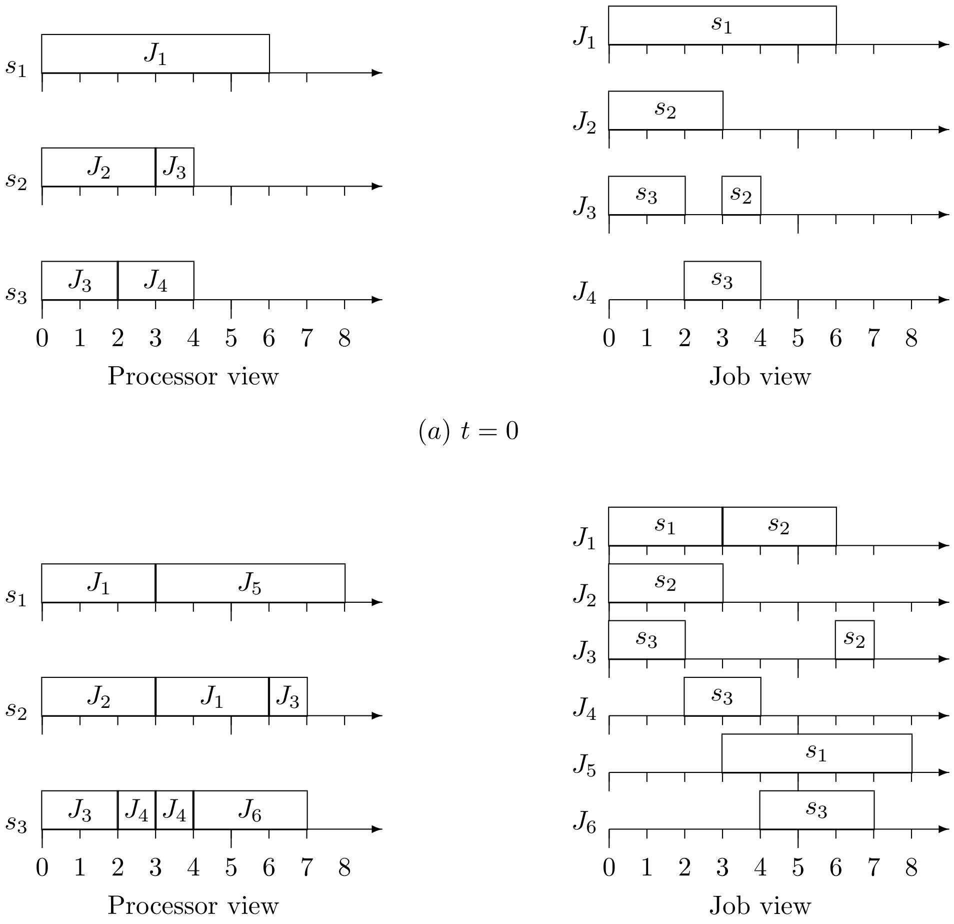
Therefore, the jobs in I cannot be scheduled in a way that ensures valid schedules for all feasible job sets without knowledge of jobs that will arrive at or after time $t = 2$.

Hong and Leung introduced the online algorithm Reschedule(I, m) that will optimally schedule jobs with common deadlines. Jobs are not assumed to have common arrival times. At each time t, algorithm Reschedule(I, m) considers only the active jobs of I. If any jobs $J_{i_1}, J_{i_2}, \ldots, J_{i_k} \in I$ have execution requirements larger than C/m, where C is the total remaining execution requirement of all active jobs, then Reschedule(I, m) assigns each of these k jobs to their own processor and recursively calls Reschedule$(I \setminus \{J_{i_1}, J_{i_2}, \ldots, J_{i_k}\}, m - k)$. If all jobs have execution requirement less than or equal to C/m, the jobs are scheduled using McNaughton's wraparound algorithm [McN59]. This algorithm lists the jobs in any order and views the list as a sequential schedule. It then cuts the sequential schedule into m equal segments of length C/m and schedules each segment on a separate processor. There is no concern about executing the same job simultaneously on different processors because McNaughton's algorithm is only executed once all jobs are guaranteed to have execution requirement at most C/m. If more jobs arrive at a later time, the jobs in I are updated by replacing their execution requirements with their *remaining* execution requirements. The new jobs are then added to the job set and the algorithm Reschedule is executed again. The following example illustrates the algorithm Reschedule.

Example 2.2 ([HL88]) *Let $I = \{J_1 = (0, 6, 10), J_2 = J_3 = (0, 3, 10), J_4 = (0, 2, 10), J_5 = (3, 5, 10), J_6 = (3, 3, 10)\}$ be scheduled on three unit-speed processors. Initially, only jobs $J_1, J_2, J_3,$ and J_4 are active and $C = 6 + 3 + 3 + 2 = 14$. Then $C/m = \frac{14}{3} = 4\frac{2}{3}$ and $3 \leq C/m < 6$, so J_1 is assigned to its own processor and* Reschedule *is recursively called with $I = \{J_2, J_3, J_4\}$ and $m = 2$. Since $\frac{8}{2} = 4$ is larger than the execution requirements of jobs J_2, J_3 and J_4, McNaughton's algorithm is used to schedule these jobs. Inset (a) of Figure 2.2 illustrates the schedule generated at time $t = 0$.*

When jobs J_5 and J_6 arrive at time $t = 3$, algorithm Reschedule *is called again. The execution requirements of jobs J_1, J_3 and J_4 become 3, 1 and 1, respectively. Job J_2 is no longer active so it is not included in the second* Reschedule *call. In the second call to* Reschedule, *$C = 3+1+1+5+3 = 13$ so $C/m = \frac{13}{3} = 4\frac{1}{3}$. Since $3 \leq 4\frac{1}{3} < 5$, the algorithm assigns job J_5 to processor s_1 recursively calls* Reschedule *with $C = 3 + 1 + 1 + 3 = 8$ and $m = 2$, at which point McNaugton's algorithm is applied. Inset (b) of Figure 2.2 illustrates the resulting schedule.*

Dertouzos and Mok [DM89] also considered online scheduling algorithms on identical multiprocessors. They isolated three job properties that must be known in order to optimally schedule jobs. This result applies to general real-time instances — it does not apply to periodic or sporadic task sets.



(a) $t = 0$

(b) $t = 3$

Figure 2.2: Algorithm Reschedule.

Theorem 2 ([DM89]) *For two or more processors, no real-time scheduling algorithm can be optimal without complete knowledge of the 1) deadlines, 2) execution requirements, and 3) start times of the jobs.*

They also found conditions under which a valid schedule can be assured even if one or more of these properties is not known. For general real-time instances, they determined that if the jobs can be feasibly scheduled when they arrive simultaneously, then it is possible to schedule the jobs without knowing the arrival times even if they do not arrive simultaneously.

Theorem 3 ([DM89]) *Let $I = \{J_1, J_2, \ldots, J_n\}$ and $I' = \{J_1', J_2', \ldots, J_n'\}$ be two real-time instances satisfying the following*

- *the jobs of I and the jobs of I' differ only in their arrival times: for all $i = 1, 2, \ldots, n$, the execution requirements are equal, $c_i = c_i'$, and the jobs have the same amount of time between arrival times and deadlines, $d_i - r_i = d_i' - r_i'$,*
- *the jobs of I' all arrive at the same time $r_i' = r_j'$ for all $i, j = 1, 2, \ldots, n$, and*
- *I' can be feasibly scheduled on m unit-speed processors.*

Then I can be scheduled to meet all deadlines even if the arrival times are not known in advance. In particular, the LLF *scheduling algorithm will successfully schedule I on m unit-speed processors.*

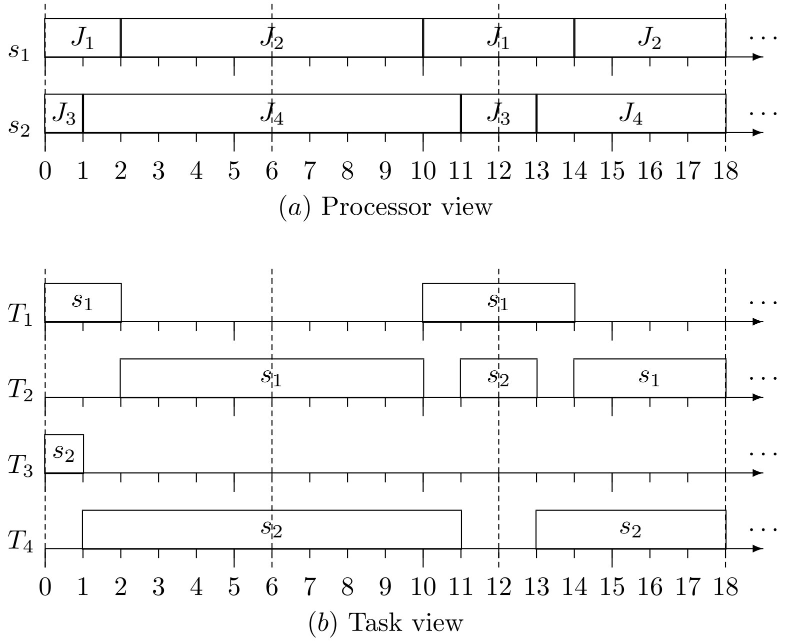
Dertouzos and Mok also found a sufficient condition for ensuring a periodic task set will meet all deadlines when the tasks are scheduled on m processors and preemptions are allowed only at integer time values.

Theorem 4 ([DM89]) *Let $\tau = \{(e_1, p_1), (e_2, p_2), \ldots, (e_n, p_n)\}$ be a periodic task set with $u_{max}(\tau) \leq 1$ and $U_{sum}(\tau) \leq m$. Define G and g as follows:*

$$\begin{aligned} G &\overset{def}{=} \gcd\{p_1, p_2, \ldots, p_n\}, \text{ and} \\ g &\overset{def}{=} \gcd\{G, G \times u_1, G \times u_2, \ldots, G \times u_n\}. \end{aligned}$$

If g is in the set of positive integers, $\mathbb{Z}^+$, then there exists a valid schedule of τ on m unit-speed processors with preemptions occurring only at integral time values.

They proved a valid schedule exists by constructing it using the concept of *time slicing*. In this strategy, the schedule is generated in slices G units long. Within each slice, task T_i receives $G \times u_i$ units of execution. While the schedule does not have to be exactly the same within each slice, each task must execute for the same amount of time within each slice. The following example illustrates a time slicing schedule.



(a) Processor view

(b) Task view

Figure 2.3: Time slicing.

Example 2.3 ([DM89]) *Let* $\tau = \{T_1 = (2,6), T_2 = (4,6), T_3 = (2,12), T_4 = (20,24)\}$*. Then* $G = \gcd\{6,6,12,24\} = 6$ *and* $g = \gcd\{6, 6 \times 2/6, 6 \times 4/6, 6 \times 2/12, 6 \times 20/24\} = \gcd\{6,2,4,1,5\} = 1$ *so the theorem applies to* τ *on* m *unit-speed processors provided* $m \geq U_{sum}(\tau) = 2$*. The schedule is divided into slices 6 units long and tasks* T_1, T_2, T_3 *and* T_4 *execute within each slice for 2, 4, 1 and 5 units of time, respectively. Figure 2.3 illustrates the time slice schedule of* τ *on two processors.*

Even though Hong and Leung and Dertouzos and Mok showed that multiprocessor online algorithms are not optimal, they can still be used for real-time systems. Kalyanasundaram and Pruhs addressed this lack of optimality by increasing the speed of the processors. The following section describes this method.

2.1.2 Resource augmentation for identical multiprocessors

A common way of analyzing online algorithms is to find their *competitive ratio*, the worst case ratio of the cost of using the online algorithm, A, to the cost of using an optimal algorithm, Opt, where the cost measure is dependent on the problem under consideration. For example, in the travelling salesman problem we need to determine the minimum length a salesman must travel in order to visit a group of cities and return home. In this case the cost is the distance travelled. In example 2.4 below, we see that the cost associated with the bin-packing problem is the number of bins required to hold a given number of items.

For minimization problems, the competitive ratio is defined as follows:

$$\rho(n) \overset{\text{def}}{=} \max_I \frac{A(I)}{\textsc{Opt}(I)} ,$$

where I is any instance containing n items, and $A(I)$ and $\textsc{Opt}(I)$ represent the cost associated with executing algorithms A and $\textsc{Opt}$ on instance I, respectively. For maximization problems the reciprocal ratio is used.

Example 2.4 *The bin packing problem addresses the following question:*

> *Let L be a list of n items with weights $w_1, w_2, \ldots, w_n$. Place the items into bins with so that the total weight of items placed in any bin is at most 1. Given an integer $k > 0$, can the items of L be placed into k bins?*

*This problem is known to be NP-complete. Therefore, any known polynomial-time algorithm will require more than the optimal number of bins. Johnson [Joh73] studied the First Fit (*FF*) algorithm, which places each item with weight w_i into the lowest indexed bin with $(1-r_\ell) \geq w_i$, where r_ℓ is the total weight of items assigned to the ℓ'th bin. He found that the competitive ratio of* FF *is 1.7. Therefore, given any list L the* FF *algorithm will never require more than 1.7·$\textsc{Opt}(L)$ bins, where $\textsc{Opt}(L)$ is the optimal number of bins required for L.*

The competitive ratio provides a measure for understanding how close a given algorithm is to being optimal and it provides us a way to compare algorithms. Kalyanasundaram and Pruhs [KP95] pointed out that the competitive ratio has some shortcomings. In particular, there are algorithms for which pathological worst-case inputs are the only inputs that incur extremely high costs, whereas common inputs always incur costs similar to the optimal algorithm. This shortcoming could cause these algorithms, which empirically perform well, to receive a poor competitive ratio.

Kalyanasundaram and Pruhs introduced the speed of the processors into the competitive analysis. Instead of comparing the two algorithms on the same processing platform, they allowed algorithm A to execute on m speed-s processors, while $\textsc{Opt}$ executes on m unit-speed processors. Thus, A is s-speed c-competitive[1] for a minimization problem if

$$\max_I \frac{A_s(I)}{\textsc{Opt}_1(I)} \leq c , \tag{2.1}$$

where each algorithm is subscripted with the executing speed of the processors the algorithm uses. Using this concept, they showed that some algorithms that have a poor competitive ratio can perform well when speed is increased even slightly. For example, they considered

[1] While Kalyanasundaram and Pruhs developed the concept of using speed in competitive analysis, Phillips, et al. [PSTW97], introduced the terminology "s-speed c-competitive."

the problem of trying to minimize the average response time of a collection of non-real-time jobs on a uniprocessor. They showed that the algorithm BALANCE, in which the processor is always shared among the jobs that have received the least amount of execution time (i.e., the jobs with the smallest balance), is $(1+\epsilon)$-speed $(1+1/\epsilon)$-competitive for minimizing response time. Thus, speeding up the processors can result in a constant competitive ratio.

For the purposes of this research, we examine the amount of work a scheduling algorithm has completed on the jobs of a real-time instance — i.e., the progress that has been made in completing each job's execution requirement, c.

Phillips, et al. [PSTW97], used the concept of s-speed c-competitive algorithms to develop tests for identical-multiprocessor real-time scheduling algorithms. They proved that, with respect to total progress toward completion of the jobs' execution requirements, all work-conserving algorithms are $(2-1/m)$-speed 1-competitive — i.e.,

$$\max_I \frac{A_{2-1/m}(I)}{\textsc{Opt}_1(I)} \leq 1 \ .$$

Thus, any work conserving algorithm will do at least as much work over time on m speed-$(2-1/m)$ processors as any other algorithm, including an optimal algorithm, will do on m unit-speed processors.

Theorem 5 ([PSTW97]) *Let π be a multiprocessor comprised of m unit-speed processors and let π' be a multiprocessor comprised of m speed-$(2-1/m)$ processors. Let* Opt *be any multiprocessor scheduling algorithm and let A be any work-conserving multiprocessor scheduling algorithm. Then for any set of jobs I and any time t,*

$$W(A, \pi', I, t) \leq W(\textsc{Opt}, \pi, I, t).$$

Using this result, Phillips, et al., were able to prove many important results regarding real-time scheduling algorithms on identical multiprocessors. In particular, they showed that any real-time instance that is *feasible* on m unit-speed processors is f-EDF-*schedulable* if the m processors execute at speed $(2-1/m)$. Thus, increasing the speed allows us to use an online algorithm even though no online algorithm can be optimal. Phillips, et al., coined the term *resource augmentation* to describe this technique of adding resources to overcome the limitations of online scheduling.

Lam and To [LT99] explored augmenting resources by adding machines as well as increasing their speed. In particular, they proved that if I is feasible on m unit-speed processors then I is f-EDF-schedulable on $(m+p)$ processors of speed $(2-\frac{1+p}{m+p})$.

Theorem 6 ([LT99]) *Let I be any real-time instance. Assume I is known to be feasible on m unit-speed processors. Then I will meet all deadlines when scheduled on $(m+p)$ speed-$(2-\frac{1+p}{m+p})$ processors using the* f-EDF *scheduling algorithm.*

In addition, they showed that the minimum speed augmentation required for any online algorithm is at least $\max\left\{\frac{km+m^2}{k^2+m^2+pm} \mid 0 \le k \le m\right\}$. Thus, if A is any online algorithm and $s < \frac{km+m^2}{k^2+m^2+pm}$, for some k, there exists a real-time instance that is feasible on m unit-speed processors but is not A-schedule on $(m+p)$ speed-s processors.

Example 2.5 *Let π be an identical multiprocessor comprised of five unit-speed processors and let π' be comprised of six speed-s processors. If $k = 2$, then $\frac{10+25}{4+25+5} = \frac{35}{34} \approx 1.029$ Therefore, if $s < 1.029$ there is some instance I that is feasible on π, but is not online schedulable on π'.*

2.1.3 Partitioned scheduling

The previous sections have explored algorithms that allow jobs to migrate. This section presents a utilization test for partitioned scheduling of periodic task sets.

Lopez, et al. [LGDG00], determined a utilization bound for p-EDF on identical multiprocessors. They considered a variety of methods for assigning tasks to processors. Some of the methods they considered were first fit (FF), in which each task is assigned to the processor with the lowest index that has enough spare capacity, best fit (BF), in which each task is assigned to the processor with the least spare capacity that exceeds the task's utilization, and random fit (RF), in which each task can be assigned to any processor that has enough spare capacity. They also considered sorting the tasks in decreasing order of utilization prior to assigning them to the processors. Interestingly, they found that all these variations have the same utilization bound.

This utilization bound is a function of $\left\lfloor \frac{1}{u_{max}(\tau)} \right\rfloor$, denoted β, which is the fewest number of tasks that can be assigned to a processor before attempting to assign a task whose utilization exceeds the processor's spare capacity. The test considers scheduling n tasks on m unit-speed processors. If $n \le \beta \cdot m$, then any reasonable allocation will clearly work. If $n > \beta \cdot m$, the utilization bound that guarantees p-EDF-schedulability on identical multiprocessors is

$$U_{sum} \le \frac{\beta m + 1}{\beta + 1}\ . \tag{2.2}$$

Figure 2.4 illustrates the utilization bound for various values of β.

This bound was also used to determine the minimum number of processors required to schedule n tasks with a given β and U_{sum}:

$$m \ge \begin{cases} 1 & \text{if } U_{sum} \le 1 \\ \min\left\{ \left\lceil \frac{n}{\beta} \right\rceil, \left\lceil \frac{(\beta+1)U_{sum}-1}{\beta} \right\rceil \right\} & \text{if } U_{sum} > 1 \end{cases}\ . \tag{2.3}$$

Example 2.6 *Can any task set τ with $U_{sum}(\tau) = 2.8$ and $u_{max}(\tau) = 0.4$ be scheduled on*

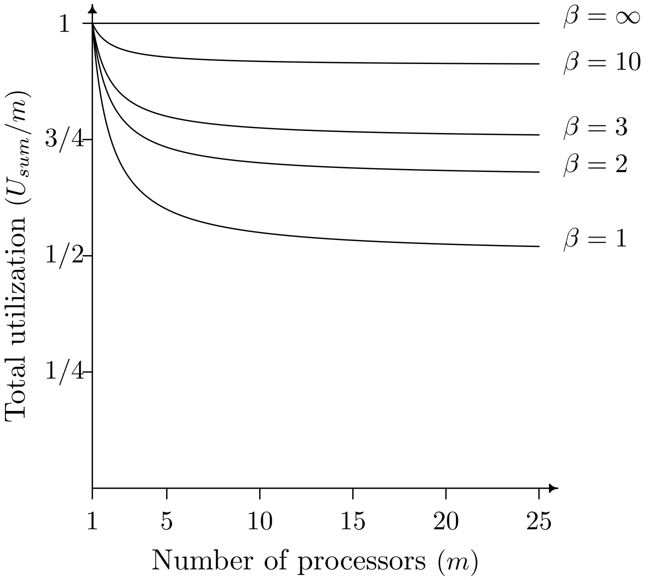


Figure 2.4: Utilization bounds guaranteeing p-EDF-schedulability on identical multiprocessors. The utilization bounds depend on β, which is the maximum number of tasks with utilization u_{max} that can fit on a single processor.

three unit speed processors? Since $u_{max}(\tau) = 0.4$, *the value of* β *is* $\lfloor 1/0.4 \rfloor = \lfloor 5/2 \rfloor = 2$. *Therefore, Condition* (2.2) *evaluates to*

$$2.8 \leq \frac{2 \cdot 3 + 1}{2 + 1} = \frac{7}{3},$$

which is false. Therefore, there may be some task set τ *with* $U_{sum}(\tau) = 2.8$ *and* $u_{max}(\tau) = 0.4$ *that cannot be partitioned onto three unit speed processors. For example, the task set comprised of seven tasks with utilization 0.4 cannot be partitioned onto three processors. Applying Condition* (2.3) *to* τ *gives*

$$m \geq \min\left\{ \left\lceil \frac{10}{2} \right\rceil, \left\lceil \frac{(2+1) \cdot 2.8 - 1}{2} \right\rceil \right\} = \min\{5, 4\} = 4,$$

so τ *can be scheduled on any identical multiprocessor with at least 4 processors.*

2.1.4 Predictability on identical multiprocessors

Ha and Liu [HL94] studied the effects of having a job complete in less than its allowed execution time. In particular, they determined the conditions under which a job completing early may cause another job to behave in an unexpected manner. In their model, they allowed jobs to have a *range* of execution requirements. Each job J_i is described by the three-tuple $(r_i, [c_i^-, c_i^+], d_i)$, where r_i is its arrival time, d_i is its deadline, and c_i^- and c_i^+ are its minimum and maximum execution requirements, respectively. This research studies the behavior of jobs as the execution requirement varies when the jobs are executed on an identical multiprocessor.

Given an instance, I, Ha and Liu considered three possible schedules. In the *actual* schedule, denoted A, each job J_i executes for c_i time units, where $c_i^- \leq c_i \leq c_i^+$. In the *minimal* and *maximal* schedules, denoted A^- and A^+, each job J_i executes for c_i^- and c_i^+ time units, respectively. For each job J_i, the start and finish times of J_i in the actual schedule are denoted $S(J_i)$ and $F(J_i)$. The start time is the moment when the job is first scheduled to execute, which may occur at or after its arrival time. Similarly, the finish time is the time at which the job has completed c_i units of work, which occurs at or before the deadline if the schedule is valid. The start and finish times in the minimal schedule are denoted $S^-(J_i)$ and $F^-(J_i)$. Finally, the start and finish times in the maximal schedule are denoted $S^+(J_i)$ and $F^+(J_i)$. The execution of J_i is *predictable* if $S^-(J_i) \leq S(J_i) \leq S^+(J_i)$ and $F^-(J_i) \leq F(J_i) \leq F^+(J_i)$. Thus, in predicable schedules, the start and finish times of the actual schedule, in which $c_i^- \leq c_i \leq c_1^+$, are bounded by the start and finish times of the maximal and minimal schedules.

While we may intuitively expect that a job's completing early would cause all subsequent jobs to arrive and finish earlier, this is not always the case. For example, the jobs are not predictable if they are scheduled using EDF without allowing migration. This is illustrated in the following example (adapted from [HL94]).

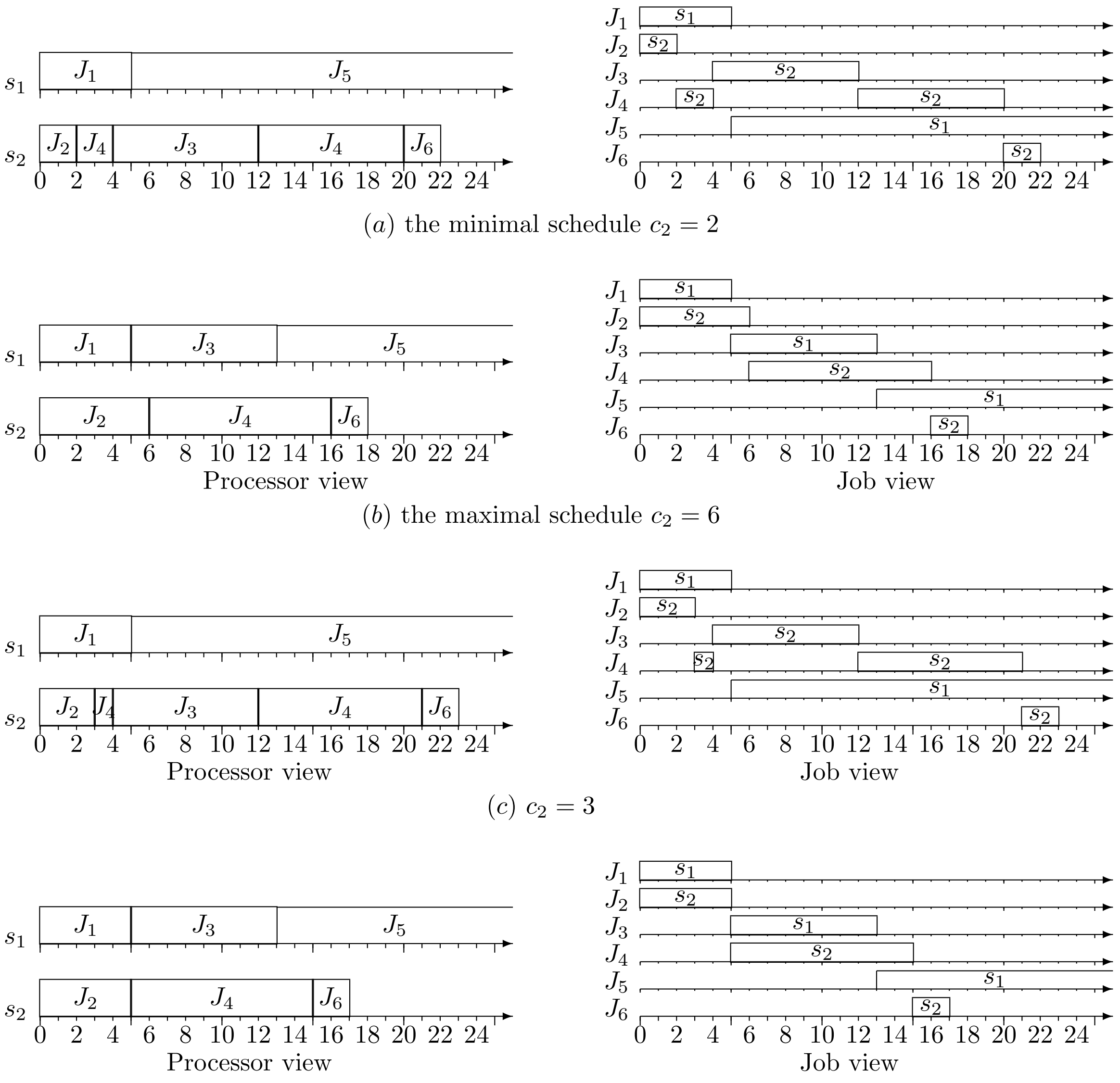
Example 2.7 *Table 2.1 lists six jobs indexed according to their* EDF*-priority. These jobs are scheduled on two unit-speed processors using* EDF *without migration. A job is assigned to a processor if either (1) the processor is idle, or (2) the job's priority is higher than any job currently assigned to the processor. If neither condition holds at the job's arrival time, the processor assignment is delayed until one of the conditions holds. Inset* (*a*) *of Figure 2.5 illustrates the minimal schedule and inset* (*b*) *illustrates the maximal schedule. Insets* (*c*) *and* (*d*) *illustrate the schedules that result when* J_2 *executes for 3 and 5 time units, respectively.*

Both jobs J_4 *and* J_6 *violate predictability.* $S(J_6)$ *is 21 when* $c_2 = 3$ *and 15 when* $c_2 = 5$ *even though* $S^-(J_6) = 20$ *and* $S^+(J_6) = 16$*. Also,* $F(J_4)$ *is 21 when* $c_2 = 3$ *and 15 when* $c_2 = 5$ *even though* $F^-(J_4) = 20$ *and* $F^+(J_4) = 16$*. Most notably, both the minimal and maximal schedules are valid even though job* J_4 *misses its deadline when* $c_2 = 3$.

Example 2.7 illustrates how important predictability can be. If a set of jobs is not predictable under a given scheduling algorithm, then it is not enough to verify that all deadlines will be met in the maximal schedule — a more comprehensive test is required. However, if we know the system is predictable, we need to verify only that the maximal schedule is valid. Ha and Liu proved that work-conserving systems that allow both preemption and migration are predictable. Also, if all jobs arrive simultaneously, then systems that do not allow migration are predictable regardless of whether preemption is allowed or not. However, if no migration

Job	r_i	d_i	$[c_i^-, c_i^+]$
J_1	0	10	[5,5]
J_2	0	10	[2,6]
J_3	4	15	[8,8]
J_4	0	20	[10,10]
J_5	5	35	[20,20]
J_6	7	40	[2,2]

Table 2.1: A job set with execution requirement ranges.



(a) the minimal schedule $c_2 = 2$

(b) the maximal schedule $c_2 = 6$

(c) $c_2 = 3$

(d) $c_2 = 5$

Figure 2.5: Four schedules of I on two unit-speed processors using EDF without migration.

is allowed and the jobs do not arrive simultaneously, predictability is assured only if jobs that arrive earlier are given higher priority — i.e., if the jobs are scheduled using First In First Out (FIFO).

2.1.5 EDF with restricted migration

Baruah and Carpenter [BC03, BC] developed schedulability tests for EDF with restricted migration (r-EDF) on identical multiprocessors. They established the following utilization bound:

Theorem 7 ([BC03, BC]) *Let τ be any task set. If*

$$U_{sum}(\tau) \leq m - (m-1) \cdot u_{max}(\tau) \ ,$$

then τ can be feasibly scheduled on m unit-speed processors using r-EDF*.*

If $u_{max}(\tau)$ is large, this utilization bound can be prohibitively small. This is illustrated in the following example.

Example 2.8 *Let τ contain three tasks with utilization 0.75, five tasks with utilization 0.3 and one task with utilization 0.2. Then $U_{sum}(\tau) = 3.95$. Assume we schedule τ on five unit-speed processors. Then the above theorem is not satisfied since $m - (m-1) \cdot u_{max}(\tau) = 2$.*

Baruah and Carpenter extended this work by focussing on the higher-utilization tasks — those tasks with utilization greater than 0.5. They observed that if τ has one or more high-utilization tasks, the test may fail. However, the schedule may be valid if one processor is reserved solely for each of the high-utilization tasks and the remaining tasks execute on the remaining processors. This algorithm, called r-fpEDF, is based on the algorithm fpEDF [Bar04], which reserves processors for the high-utilization tasks and schedules the low-utilization tasks on the remaining processors using full migration EDF. Baruah and Carpenter found that both r-fpEDF and fpEDF have the same utilization bound.

Theorem 8 ([BC03, BC]) *Let τ be any task set. If the following condition is satisfied*

$$U_{sum}(\tau) \leq \begin{cases} m - (m-1)u_{max}(\tau) & \text{if } u_{max}(\tau) \leq 0.5 \\ m/2 + u_{max}(\tau) & \text{if } u_{max}(\tau) \geq 0.5 \ , \end{cases}$$

then τ can be successfully scheduled on m unit-speed processors using either algorithm fpEDF *or* r-fpEDF*.*

In Chapter 5 of this dissertation, we extend these results for uniform heterogeneous multiprocessors.

2.2 Results for uniform heterogeneous multiprocessors

The previous section discussed real-time scheduling results on identical multiprocessors. This section discusses the use of uniform heterogeneous multiprocessors. Section 2.2.1 presents work by Liu and Liu [LL74] and by Horvath, et al. [HLS77]. The work in this section concerns scheduling jobs without deadlines with the goal of minimizing the time required to finish all the jobs. Section 2.2.2 presents work by Hochbaum and Shmoys [HS87, HS88] concerning bin-packing with bins of different sizes. Since bin packing and partitioned scheduling have a strong correlation, the results of Hochbaum and Shmoys can be used to schedule jobs on uniform heterogeneous multiprocessors. Finally, Section 2.2.3 presents work by Baruah [Bar02] regarding the robustness of f-EDF on uniform heterogeneous multiprocessors.

2.2.1 Scheduling jobs without deadlines

Liu and Liu [LL74] studied non-real-time scheduling on uniform heterogeneous multiprocessors. They considered jobs with *precedence constraints*, which impose an ordering of the jobs over time. A job J_i precedes job J_j, denoted $J_i < J_j$, if J_j cannot begin to execute before J_i has completed executing. The notation $(\mathcal{J}, <)$ represents a set of jobs $\mathcal{J}$ with precedence constraints. Given a set $(\mathcal{J}, <)$, Liu and Liu considered schedules satisfying the following properties:

- job-level fixed-priority,
- work conserving, and
- non-preemptive.

Schedules satisfying these three properties can differ due to different priority assignments.

Deadlines are not a concern for non-real-time jobs. Nonetheless, the amount of time required to complete all the jobs, called the *makespan*, may be of concern. Liu and Liu determined the maximum ratio between the makespan of two schedules.

Theorem 9 ([LL74]) *Let π be any m-processor uniform heterogeneous multiprocessor with maximum speed $s_1(\pi)$ and total speed $S(\pi)$ and let $(\mathcal{J}, <)$ be a set of jobs with precedence constraints. Assume these jobs are scheduled on π using two different schedules. Let ω be the makespan of a job-level fixed-priority, work-conserving, non-preemptive schedule, and let ω' be the makespan of an arbitrary schedule. Then*

$$\frac{\omega}{\omega'} \leq \frac{s_1(\pi)}{s_m(\pi)} + 1 - \frac{s_1(\pi)}{S(\pi)}.$$

Moreover, this bound is the best possible.

Therefore, the makespan of any schedule of $(\mathcal{J}, <)$ can be used to bound the makespan of any other schedule. Furthermore, even though ω may not be the minimum makespan, this theorem can be used to determine how far any schedule may be from the optimal.

Liu and Liu also compared the makespan of schedules on two *different* uniform heterogeneous multiprocessors.

Theorem 10 ([LL74]) *Let π and π' be any two uniform heterogeneous multiprocessors and let $(\mathcal{J}, <)$ be a set of jobs with precedence constraints. Assume these jobs are scheduled on π and π' using two different schedules. Let ω be the makespan of a job-level fixed-priority, work-conserving, non-preemptive schedule on π, and let ω' be the makespan of an arbitrary schedule on π'. Then*

$$\frac{\omega}{\omega'} \leq \frac{s_1(\pi')}{s_m(\pi)} + \frac{S(\pi') - s_1(\pi')}{S(\pi)} \ ,$$

where $s_1(\pi)$ and $s_m(\pi')$ are the fastest and slowest processor speeds of π and π', respectively, and $S(\pi)$ and $S(\pi')$ are their total processing speeds. Moreover, this bound is the best possible.

Horvath, et al. [HLS77], studied the problem of scheduling a set of non-real-time preemptable jobs, all of which arrive at the same time. They showed that when independent non-real-time jobs execute on a uniform heterogeneous multiprocessor, the minimum makespan depends on the cumulative execution requirements of the jobs and the cumulative speeds of the processors.

Theorem 11 (Horvath, et al. [HLS77]) *Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n independent non-real-time jobs, each with arrival time equal to zero and indexed according to non-increasing execution requirements, $c_i \geq c_{i+1}$ for all $i = 1, 2, \ldots n-1$, and let C_i denote the cumulative execution requirement of the i largest jobs for all $i = 1, 2, \ldots, n$. Then for any m-processor uniform heterogeneous multiprocessor, π, the minimum makespan for scheduling I on π is*

$$\omega \overset{def}{=} \max\left(\max_{1 \leq i \leq m}\left\{\frac{C_i}{S_i(\pi)}\right\}, \frac{C_n}{S(\pi)}\right) \ . \tag{2.4}$$

They introduced the *level algorithm*, which always completes execution at time ω. The *level* of job J at time t is its remaining amount of work. The level algorithm recursively assigns jobs to processors. Starting at time $t = 0$, the algorithm sets j equal to m and k equal to the number of jobs with the largest execution requirement (i.e., the highest initial level). If $k > j$, the k jobs are jointly assigned to the j processors (the details of jointly assigning jobs to processors is described below). Otherwise, the k jobs are assigned to the k fastest processors and the remaining jobs are assigned to the slower processors in a similar manner. This processor assignment will remain until either some set of jobs completes executing or the level of one group equals the level of the group below it.

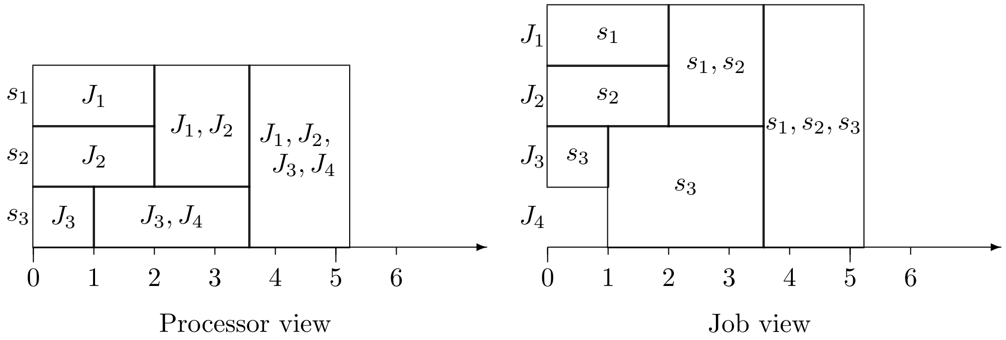


Figure 2.6: Jobs J_1, J_2, J_3, and J_4 with execution requirements $c_1 = 20, c_2 = 16, c_3 = 6$, and $c_4 = 5$ scheduled on $\pi = [5, 3, 1]$ using the level algorithm.

Example 2.9 *Let $\pi = [5, 3, 1]$ and let I be the set of jobs with execution requirements $c_1 = 20$, $c_2 = 16$, $c_3 = 6$, and $c_4 = 5$. Figure 2.6 illustrates the level-algorithm schedule of I on π. Notice that the diagrams only have one time line for all the processors (or jobs) to more easily reflect when jobs execute jointly on one or more processors. Initially, all the jobs have distinct levels so J_1, J_2 and J_3 execute on s_1, s_2 and s_3, respectively. At $t = 1$, the levels of J_3 and J_4 are both equal to 5, so these two jobs jointly execute on s_3. At $t = 2$, the levels of J_1 and J_2 are both equal to 10, so these jobs jointly execute on processors s_1 and s_2. At $t = 3\frac{4}{7}$, all the jobs have a level of $3\frac{5}{7}$ so they all execute jointly on all three processors until they complete at $t = 4\frac{17}{21}$.*

Jobs are jointly assigned to processors by dividing the shared intervals into smaller intervals and scheduling the jobs in a round-robin fashion. More specifically, if j jobs, $J_1, J_2, \ldots, J_j$, are scheduled for t time units on k processors, $s_1, s_2, \ldots, s_k$, where $k \leq j$, then the interval is divided into j subintervals of length (t/j) and each job executes on each processor for exactly one subinterval and idles for $(k - j)$ subintervals. During the first subinterval J_i executes on s_i where $1 \leq i \leq k$. After this, each job shifts down to the next processor so J_j executes on s_1 and J_i executes on s_{i+1}, where $1 \leq i \leq k - 1$. Figure 2.7 illustrates a schedule sharing five jobs among four processors for five time units.

When $\mathcal{J}$ has precedence constraints, the level of each job J_i does not depend only on c_i. Instead, the level includes the longest *chain* that starts at J_i. A chain is a sequence of jobs $J_{i_1}, J_{i_2}, \ldots, J_{i_{n'}}$ such that $J_{i_k} < J_{i_{k+1}}$ for each $k = 1, 2, \ldots n' - 1$. The length of the chain is sum of its component jobs' execution requirements: $c_{i_1} + c_{i_2} + \ldots + c_{i_{n'}}$. While the level algorithm minimizes the makespan for independent jobs, Horvath, et al. [HLS77], found that the level algorithm does not minimize the makespan when $\mathcal{J}$ has precedence constraints. The following example illustrates a set $(\mathcal{J}, <)$ whose makespan is not minimized when scheduled using the level algorithm.

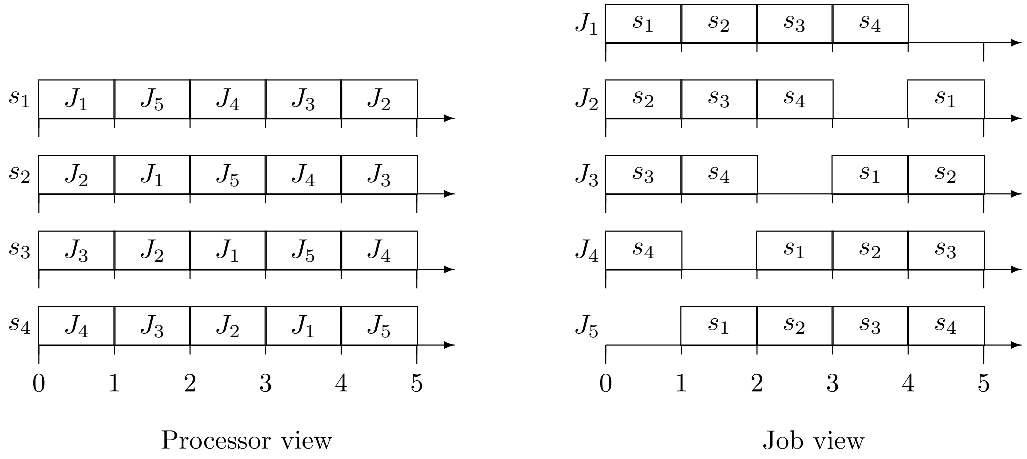


Figure 2.7: Executing five jobs on four processors.

Example 2.10 ([HLS77]) *Let $\mathcal{J}$ be comprised of 13 jobs with $c_1 = c_2 = \ldots = c_9 = 7$, and $c_{10} = c_{11} = c_{12} = 30$, and $c_{13} = \epsilon$. Assume that for all $i \leq 9$, job J_i precedes job J_{12} and that jobs J_{10}, J_{11} and J_{12} precede job J_{13}. The precedence relation for this job set is illustrated in Figure 2.8. Assume $(\mathcal{J}, <)$ is scheduled on $\pi = [s_1 = 4, s_2 = s_3 = \ldots = s_9 = 1]$. Inset (a) of Figure 2.9 illustrates the schedule of $(\mathcal{J}, <)$ on π generated by the level algorithm. Since jobs $J_1, J_2, \ldots, J_9$ all have the largest initial level, they are scheduled on the nine processors until they complete execution. After these jobs complete, jobs J_{10}, J_{11} and J_{12} can execute on the fastest three processors. Finally, once these jobs complete, job J_{13} executes on the fastest processor. The makespan of this algorithm is $20.25 + \epsilon/4$. Inset (b) of Figure 2.9 illustrates the optimal algorithm, which schedules jobs $J_1, J_2, \ldots, J_9$ on seven of the unit-speed processors until they complete executing at time 9. During this time, the remaining two processors execute jobs J_{10} and J_{11}. At time 9, job J_{12} is eligible to execute because all the jobs that precede it have completed executing. Therefore, at this time J_{12} executes on the fastest processor and jobs J_{10} and J_{11} execute on two of the unit-speed processors. All three jobs complete executing after 7.5 time units at time 16.5. At this point job J_{13} can execute on the fastest processor. The makespan of the optimal schedule is $16.5 + \epsilon/4$, which is less than the makespan of the level algorithm.*

Even though the level algorithm does not find the minimal makespan when jobs have precedence constraints, the ratio between the makespan of the level algorithm and the minimal makespan is bounded. The bound depends on the speeds of the processors of π. For any

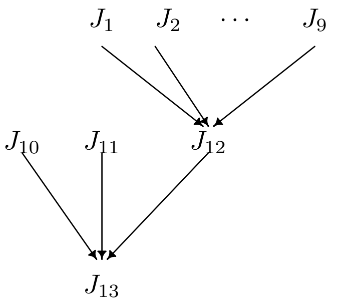
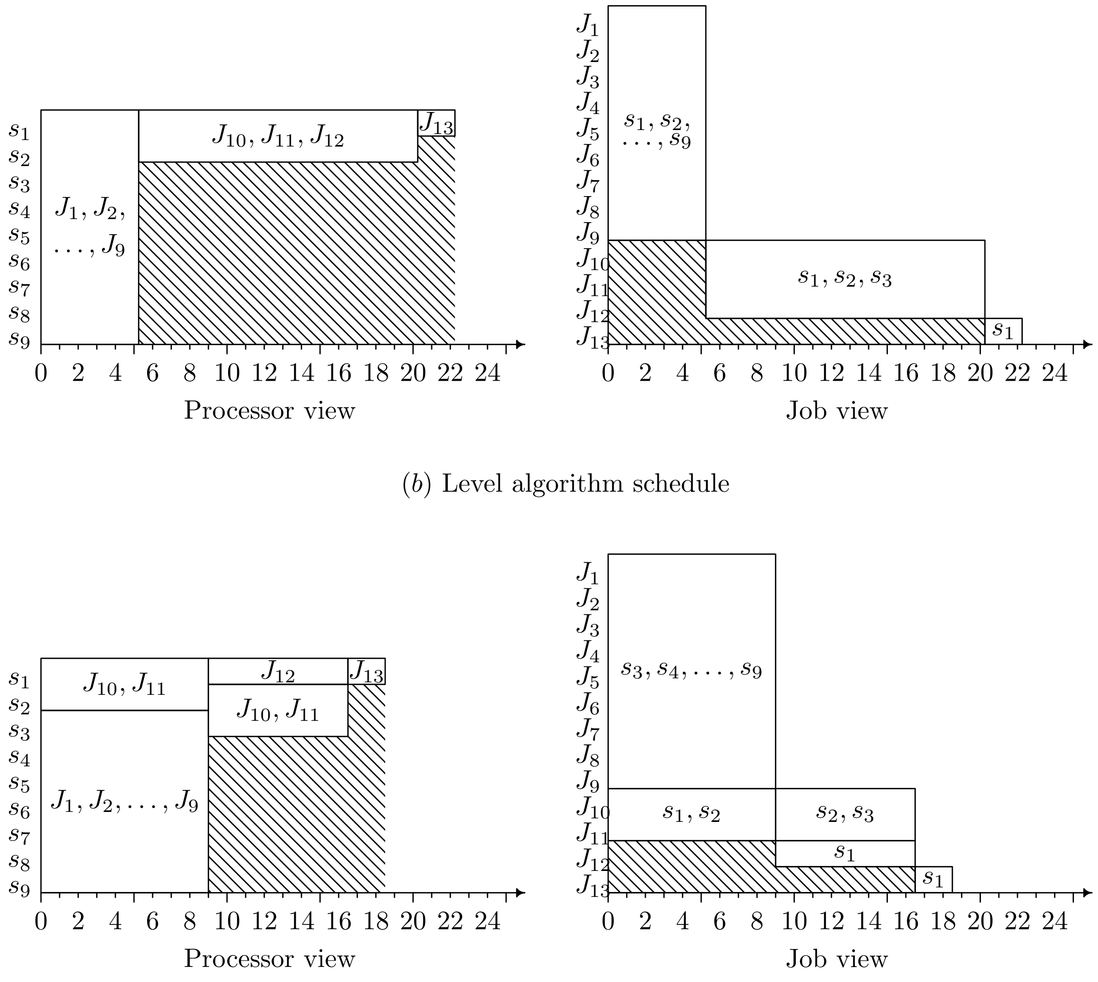


Figure 2.8: Precedence graph.



(b) Level algorithm schedule

(b) Optimal schedule

Figure 2.9: The level algorithm is not optimal when jobs have precedence constraints.

uniform heterogeneous multiprocessor, define α_i for each $i = 2, 3, \ldots, m-1$ as follows:

$$\alpha_i \stackrel{\text{def}}{=} \frac{S_i(\pi)/i}{S(\pi) - S_i(\pi)}.$$

Intuitively, α_i is the ratio of execution rate versus idle rate when the i fastest processors are executing i jobs. Define α and β as follows:

$$\alpha \stackrel{\text{def}}{=} \min\{\alpha_i \mid 1 < i < m\}, \text{ and}$$

$$\beta \stackrel{\text{def}}{=} \frac{S_2(\pi)}{S(\pi) - S_2(\pi)}.$$

Given these values for α and β, Horvath, et al. [HLS77], determined the the maximum ratio between the makespan of the level algorithm and an optimal schedule.

Theorem 12 ([HLS77]) *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor and let $(\mathcal{J}, <)$ be a set of jobs with precedence constraints. Assume the level algorithm schedules $(\mathcal{J}, <)$ on π with a makespan of ω. Let ω' be the minimal makespan of $(\mathcal{J}, <)$ on π. Then*

$$\frac{\omega}{\omega'} \leq 1 + \min\left\{\frac{1}{\beta}, \frac{s_1(\pi)}{\alpha s_m(\pi)}\right\}.$$

Example 2.11 *For $\pi = [s_1 = 4, s_2 = s_3 = \ldots = s_9 = 1]$, the values of α_i for $i = 1, 2, \ldots, 8$ are*

$$\alpha_1 = \frac{4}{8}, \alpha_2 = \frac{5/2}{7}, \alpha_3 = \frac{6/3}{6}, \alpha_4 = \frac{7/4}{5}, \alpha_5 = \frac{8/5}{4}, \alpha_6 = \frac{9/6}{3}, \alpha_7 = \frac{10/7}{2}, \alpha_8 = \frac{11/8}{1}.$$

Therefore

$$\alpha = \max\{\alpha_i \mid 1 \leq i \leq 8\} = \frac{1}{3} \text{ and } \beta = \frac{S_2(\pi)}{S(\pi) - S_2(\pi)} = \frac{5}{7}.$$

Applying Theorem 12 to π, we have

$$\frac{\omega}{\omega'} \leq 1 + \min\left\{\frac{5}{7}, \frac{4}{1/3 \cdot 1}\right\} = 1\frac{5}{7} \approx 1.714,$$

so the makespan of the level algorithm will never exceed 1 5/7 times the optimal makespan. Comparing the makespans found in Example 2.10, we see that

$$\frac{20.25 + \epsilon/4}{16.5 + \epsilon/4} < 1.23 \leq 1\frac{5}{7}.$$

In this section, we have examined reducing the completion time of non-real-time jobs. In the next section, we will examine partitioned scheduling of real-time tasks on a uniform heterogeneous multiprocessor.

2.2.2 Bin packing using different-sized bins

The optimality of EDF on uniprocessors suggests a correlation between bin packing and partitioned scheduling. Recall that the bin-packing problem considers a set of *items*, each with a given *weight* less than 1. Each item can be placed in a single bin and the total weight of all items placed in any bin cannot exceed 1. The question posed by the bin-packing problem is "Can a given set of items fit into k bins?" The *variable-sized bin-packing* problem is a modification of the original bin-packing problem in which each bin has an associated size. Instead of requiring that the total weight of items placed in a bin to be at most 1, this problem requires that the total weight is at most the size of the bin. Therefore, the question posed by this problem is "Can a given set of items fit into k bins with sizes $b_1, b_2, \ldots, b_k$?".

Partitioning tasks onto a uniform heterogeneous multiprocessor can easily be reduced to the variable-sized bin-packing problem: each bin of size b is equated with a processor of speed b, and each item of weight w is equated with a task with utilization w. By the optimality of EDF on uniprocessors [LL73], a set of tasks can be scheduled to meet all deadlines on a processor of speed b if and only if the total utilization of the tasks is at most b. Therefore, a set of tasks can be partitioned onto k processors of speed $s_1, s_2, \ldots, s_k$ if and only if the corresponding items can be placed into the k bins with sizes $s_1, s_2, \ldots, s_k$.

Johnson [Joh73] showed that the bin-packing problem is NP-hard in the strong sense even when the bins are all the same size. Since bin-packing with variable bin sizes is a generalization of the bin-packing problem, it, too, is NP-hard in the strong sense. Therefore, it is unlikely that there exists an algorithm that runs in polynomial time and solves the variable sized bin-packing problem exactly. Hochbaum and Shmoys [HS87, HS88] designed a polynomial-time approximation scheme (PTAS) for the variable-sized bin-packing problem. For any ϵ, this algorithm *relaxes* the variable-sized bin-packing problem by a factor of ϵ: Items with weight totalling as much as $(1 + \epsilon) \cdot b$ can be assigned to a bin of size b. The relaxed algorithm, denoted A_ϵ, either finds a packing of the items into the relaxed bins or states that there is no solution to the original (unrelaxed) problem. Therefore, the approximation scheme gives three possible answers to the variable-sized bin-packing problem:

Yes: If A_ϵ finds a solution to the relaxed problem and none of the bins are overfilled, then this is also a solution to the original (unrelaxed) problem.

No: If A_ϵ cannot find a solution to the relaxed problem, then the original problem has no solution.

Maybe: If A_ϵ finds a solution to the relaxed problem and one or more bins are overfilled, then it is unknown whether the original problem has a solution.

The approximation algorithm for the variable-sized bin-packing problem is a modification of the approximation algorithm for the standard bin-packing problem, also developed by

Hochbaum and Shmoys [HS87]. The algorithm A_ϵ approximates the standard bin-packing problem by allowing bins to be filled up to a level of at most $(1+\epsilon)$. This algorithm has two phases. The first phase places all pieces whose size is larger than ϵ into bins and the second phase places the smaller pieces into bins. The smaller pieces may be placed into any bin that is not already overfilled. If all bins are overfilled, a new bin may be started. Notice that the second phase can never cause the number of used bins to exceed the minimum number of bins required since it only increases the number of bins when all the currently opened bins are overfilled.

The first phase of A_ϵ begins by partitioning each bin into $q = \lceil 1/\epsilon^2 \rceil$ equal-sized segments, $(\ell_1, \ell_2], (\ell_2, \ell_3], \ldots, (\ell_q, \ell_{q+1}]$, where $\ell_1 = \epsilon$ and $\ell_{q+1} = 1$. It then maps each piece to the interval which contains the piece's size. Finally, the algorithm places the pieces into bins. When a piece with weight w, where $\ell_i < w \leq \ell_{i+1}$, is placed in a bin, i segments of the bin become occupied. The amount of space reserved for each piece is always less than the actual size of the piece by at most ϵ^2. Since each bin contains at most $\lfloor 1/\epsilon \rfloor$ items (since only large pieces are considered in the first phase), the total size of the pieces packed in each bin is at most $(1+\epsilon)$.

Thus, A_ϵ reduces the bin-packing problem to one in which there are a fixed number of piece sizes, which can be solved in polynomial time using dynamic programming (for example, see [CLRS01]). The dynamic programming table has $O(n^q)$ entries and each entry takes at most $(1/\epsilon)^q$ time to compute. Therefore, the minimum number of required bins can be calculated in $O((n/\epsilon)^q) = O((n/\epsilon)^{\lceil i/\epsilon^2 \rceil})$ time.

When extending this algorithm to account for variable-sized bins, the bin sizes (and hence the pieces) are normalized so that the largest bin has a size of 1. Recall that when bin sizes are identical, all bins are partitioned into $\lceil 1/\epsilon^2 \rceil$ equal sized partitions. In the identical bin-packing problem, the segment size is the same for every bin. However, when the bin sizes vary, the segment sizes vary as well. Therefore, a piece reserves a different number of segments depending on the bin in which it is placed. Moreover, the size that determines if a piece is small depends on the bin in question. For example, if there is a bin whose size is ϵ^2, a piece whose size is ϵ will not fit in that bin. Therefore, it makes no sense to call such a piece small with respect to the given bin.

In order to address these two issues, both bins and pieces are categorized according to their sizes. A bin with size b is put into category $\jmath$ if $\epsilon^{\jmath+1} < b \leq \epsilon^\jmath$ and an item with weight w is placed in the category ℓ if $\epsilon^{\ell+1} < w \leq \epsilon^\ell$. Moreover, pieces are rounded down to their nearest multiple of $\epsilon^{\ell+2}$ — *i.e.*, w is replaced by $w' = \lfloor p/\epsilon^{\ell+2} \rfloor \cdot \epsilon^{\ell+2}$. Since the interval $(b \cdot \epsilon, b]$ may overlap two piece categories, an additional "medium-sized" category is also considered for each bin size. Table 2.2 describes the relationship between bins and pieces. Notice that a small piece for a bin is never more than ϵ times the bin's size.

As in the identical bin-packing algorithm, bins are packed one at a time. The bins are

$w \leq \epsilon^{\ell+2}$	i is small for bin b	b is enormous for i
w in category $\ell + 1$	i is medium for bin b	b is huge for i
w in category ℓ	i is large for bin b	b is large for i

Table 2.2: Relationship of an item i with weight w to bin b in size category ℓ

packed starting with bins in category 0 (i.e., bins with weight between ϵ and 1) and working to bins in category k_ϵ, where k_ϵ is the total number of categories. Each bin is packed in two phases: The large and medium phase and the small phase. If the bin is not overfilled after the large and medium phase, the small phase places pieces in the bin that are small with respect to the bin. The large and medium phase is implemented using a layered graph rather than a dynamic programming table due to the added complexity caused by the varied bin sizes.

Hochbaum and Shmoys show that the layered graph contains at most $O(2m \cdot n^{2/\epsilon^2+3} \cdot 1/\epsilon^6)$ nodes, where m is the number of bins and n is the number if items, and that the number of arcs originating at each node is $O((1/\epsilon^2)^{2/\epsilon^2})$, giving a total of $O(2m(n/\epsilon^2)^{(2/\epsilon^2+3)})$ arcs in the graph. Each arc evaluation requires at most $(2/\epsilon^2) - 1$ additions and one comparison. Furthermore, the updates between stages require at most n additions and 3 comparisons. This gives a total complexity of $O((2m(n/\epsilon^2)^{(2/\epsilon^2+3)}) \cdot 2/\epsilon^2 + k_\epsilon \cdot n)$, where k_ϵ is the number of bin categories of π for the given ϵ.

2.2.3 Real-time scheduling on uniform heterogeneous multiprocessors

The use of uniform heterogeneous multiprocessors for real-time scheduling has not received much attention until recently. This dissertation presents EDF-schedulability tests for uniform heterogeneous multiprocessors. The full migration test relies on a result developed by Baruah concerning the robustness of f-EDF on uniform heterogeneous multiprocessors. In addition, Baruah and Goossens developed an RM-schedulability test for uniform heterogeneous multiprocessors. These results will be described in more detail in this section, beginning with the RM-schedulability test.

Rate monotonic scheduling. Baruah and Goossens [BG03a, BG03b] developed a schedulability test for the rate monotonic (RM) scheduling algorithm with full migration on uniform heterogeneous multiprocessors. Recall that RM is a fixed-priority scheduling algorithm that assigns higher priority to tasks with smaller periods. All the jobs generated by a given task have the same priority. Baruah and Goossens developed the following schedulability test for

RM on uniform heterogeneous multiprocessors.

Theorem 13 ([BG03a, BG03b]) *Let τ be any task set and let π be any uniform heterogeneous multiprocessor. If*

$$U_{sum}(\tau) \leq \frac{1}{2}\big(S(\pi) - (1 + \lambda(\pi)) \cdot u_{max}(\tau)\big) \ ,$$

where $S(\pi)$ is the total speed of π and $\lambda(\pi) = \max\{(\sum_{i=k+1}^{m} s_i)/s_k \mid 1 \leq k < m\}$ is π's identicalness parameter, then τ can be successfully scheduled on π using RM *with full migration.*

This utilization bound is quite similar to the full migration EDF bound found in Chapter 3 of this dissertation. The bound for RM is smaller than the bound for EDF. In EDF, active jobs gain higher priority as their deadlines approach. By contrast, RM's jobs always have the same priority even if they are about to reach their deadlines. The decreased bound compensates for the difficulties inherent in fixed-priority scheduling.

Baruah also developed results concerning the dynamic scheduling algorithm f-EDF on uniform heterogeneous multiprocessors. In the remainder of this section, we will present these results.

The robustness of f-EDF.

We can determine if a set of jobs is schedulable on the multiprocessor π if we consider multiprocessors that are "less powerful" than π. The following definition provides a structure by which some processors can be seen to be more powerful than others.

Definition 4 (π dominates π' ($\pi \trianglerighteq \pi'$)) *Let π and π' be two uniform heterogeneous multiprocessors. Then π* dominates *π' if **(i)** $m(\pi) \geq m(\pi')$, and **(ii)** $s_i(\pi) \geq s_i(\pi')$ for all $i = 1, 2, \ldots, m(\pi')$.*

If a scheduling algorithm A is guaranteed to successfully schedule any real-time instance I on π whenever I is A-schedulable on some multiprocessor π' dominated by π, then A is said to be *robust with respect to domination.* Baruah [Bar02] showed that f-EDF is robust with respect to domination. Robustness is desirable because processors can be upgraded without requiring rigorous re-analysis of the entire system. Theorem 14 below asserts that f-EDF is robust with respect to domination.

Theorem 14 (Baruah [Bar02]) *Let I be any real-time instance that is* f-EDF*-schedulable on a uniform heterogeneous multiprocessor π'. Let π denote any uniform heterogeneous multiprocessor such that $\pi \trianglerighteq \pi'$. Then I is* f-EDF*-schedulable upon π.*

Baruah's work on robustness of f-EDF can be used to extend the f-EDF-schedulability test presented in Chapter 3, which identifies a subset f-EDF-schedulability region associated with

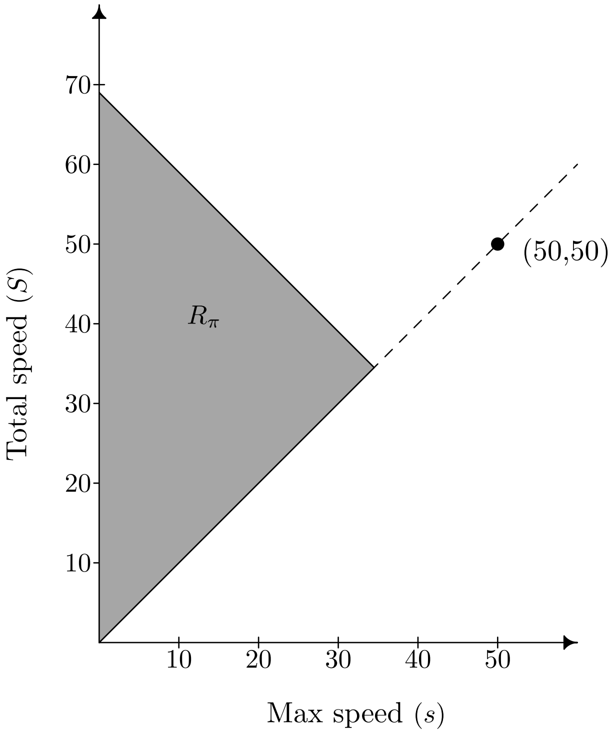


Figure 2.10: The region R_π for $\pi = [50, 11, 4, 4]$ contains all points (s, S) with $S \leq S(\pi) - s \cdot \lambda(\pi)$. Any instance I is guaranteed to be f-EDF-schedulable on π if I is feasible on some multiprocessor with fastest speed s and total speed S, where (s, S) is in the region R_π.

each uniform heterogeneous multiprocessor π. This region, denoted R_π, contains the points (s, S) such that

$$S \leq S(\pi) - \lambda(\pi) \cdot s \; .$$

Any real-time instance I that is feasible on a uniform heterogeneous multiprocessor π' with $s_1(\pi') = s$ and $S(\pi') = S$ is f-EDF-schedulable on π. For example, Figure 2.10 illustrates the region R_π for $\pi = [50, 11, 4, 4]$.

Example 2.12 *Let* $\pi = [50, 11, 4, 4]$. *Then* $\lambda(\pi) = \max\{19/50, 8/11, 4/4\} = 1$ *and* $S(\pi) = 69$. *Therefore,* $R_\pi = \{(s, S) \mid 0 < s \leq S \leq 69 - s\}$. *Figure 2.10 illustrates this area. If* (s, S) *is in the shaded area and* I *is feasible on some* π' *with* $s_1(\pi') = s$ *and* $S(\pi') = S$ *then* I *is* f-EDF-*schedulable on* π.

Unfortunately, R_π does not necessarily contain *all* the points that guarantee f-EDF-schedulability on π. Notice that $(50, 50) \notin R_\pi$. This point represents multiprocessors whose maximum and total speed are both equal to 50 — i.e., speed-50 uniprocessors. Clearly, any instance that is feasible on a speed-50 uniprocessor is feasible on a "multiprocessor" dominated by π — namely $\pi' = [50]$. By the robustness of f-EDF on uniform heterogeneous multiprocessors, any such instance is also f-EDF-schedulable on π. Therefore, all instances feasible on speed-50 uniprocessors are guaranteed to be f-EDF-schedulable on π even though $(50, 50) \notin R_\pi$.

The robustness of f-EDF on uniform heterogeneous multiprocessors can help find the points that guarantee f-EDF-schedulability but are not included in R_π. In addition to the points in R_π, we also want to include all points in $R_{\pi'}$ for every multiprocessor π' such that $\pi \trianglerighteq \pi'$. Since the set of all such multiprocessors is quite large, it behooves us to try to consider only a subset of the multiprocessors dominated by π. To that end, Baruah considered multiprocessors *cleanly dominated* by π.

Definition 5 (π cleanly dominates π^*.) *Uniform heterogeneous multiprocessor* π cleanly dominates *uniform heterogeneous multiprocessor* π^* *(denoted* $\pi \overset{\star}{\trianglerighteq} \pi^*$*) if and only if* **(i)** $\pi \trianglerighteq \pi^*$*, and* **(ii)** $s_i(\pi^*) = s_i(\pi)$ *for all* $i = 1, 2, \ldots, m(\pi^*) - 1$.

Example 2.13 *Let* $\pi = [50, 11, 4, 4]$*. Consider the uniform heterogeneous multiprocessors* $\pi_1 = [4, 3, 3, 1]$*,* $\pi_2 = [50, 11, 2]$ *and* $\pi_3 = [50, 4, 6]$*. Then* $\pi \trianglerighteq \pi_1$ *since the speed of each processor of* π *is greater or equal to the speed of the corresponding processor of* π_1*. Also,* $\pi \trianglerighteq^* \pi_2$ *since* $s_1(\pi) = s_1(\pi_2)$ *and* $s_2(\pi) \geq s_2(\pi_2)$*. Finally,* π *does not dominate* π_3 *since* $s_3(\pi) < s_2(\pi_3)$*.*

When trying to extend the region R_π to include *every* point for which feasibility on any uniform heterogeneous multiprocessor represented by the point guarantees f-EDF-schedulability on π, we need to find $R_{\pi'}$ for all processors π' dominated by π. If we can restrict our attention to only the multiprocessors *cleanly* dominated by π, the number of new processors we need to consider is substantially reduced. The following theorem states that we can restrict our attention in this manner.

Theorem 15 *Given a uniform heterogeneous multiprocessor* π *and constants* s *and* S*, if there is a uniform heterogeneous multiprocessor* π_1 *such that*

$$(\pi \trianglerighteq \pi_1) \text{ and } ((s, S) \in R_{\pi_1})$$

then there is a uniform heterogeneous multiprocessor π_2 *such that*

$$(\pi \overset{\star}{\trianglerighteq} \pi_2) \text{ and } ((s, S) \in R_{\pi_2})\,.$$

Thus, it suffices to consider only the multiprocessors cleanly dominated by π when applying the robustness of f-EDF on uniform heterogeneous multiprocessors. Chapter 3 develops these concepts more fully.

2.3 Uniform heterogeneous multiprocessor architecture

Throughout this dissertation, we represent a uniform heterogeneous multiprocessor by a vector of speeds, $\pi = [s_1, s_2, \ldots, s_m]$. In this section, we will explore the actual machines represented by this notation. In particular, we will see that uniform heterogeneous multiprocessors can have different architectures. For example, their memory may be centrally located or distributed. In this section, we will investigate how the execution requirement associated with a job may be affected by the multiprocessor architectures.

This dissertation assumes that all jobs are independent and that each processor of the system has an associated speed s, which is the amount of work that processor can complete in one unit of time. We will see that these assumptions simplify the actual behavior of real processors. Nonetheless, this model provides a close approximation of the actual behavior. In the following sections, we will see how these simplifications manifest themselves and how we can compensate for the associated errors. Sections 2.3.1 and 2.3.2 will introduce shared- and distributed-memory multiprocessors, respectively.

2.3.1 Shared-memory multiprocessors

In *shared-memory multiprocessors* (SMPs), all the processors access the memory remotely. Memory and processors connect over a bus or a general interconnection network. Figure 2.11 illustrates a shared-memory multiprocessor. Each processor has its own cache and the processors retrieve data and code from the memory via a bus. Notice that this system does not have a centralized clock — each processor operates at the speed dictated by its local clock. For example, the Dell PowerEdge 2600 can have two processors with speeds ranging from 2.4 to 3.2 GHz [Del04]. The two processors may operate at different speeds and they may have different sized caches.

In these systems the amount of time required to access memory is the same regardless of which processor is accessing the memory. This can skew the relative speeds of the processors. If a job executes on two processors, one of speed s and one of speed s', the ratio of total time it takes the job to complete executing on these two processors will probably not be s/s'. This is illustrated in the following example.

Example 2.14 *Consider a two-processor multiprocessor with processor speeds $s_1 = 2$ and $s_2 = 1$. While it is natural to infer that a job will execute twice as quickly on processor s_1 as it will on s_2, this is not necessarily the case. The difficulty arises because of the time required to access memory. Assume a job executes for 3 time units on the unit-speed processor s_2 — 2 time units executing on the processor and 1 time unit accessing memory. Then the same job executing on processor s_2 will require half the amount of time for execution, but the same amount of time required to access memory — i.e., a total of 2 units of time to complete*

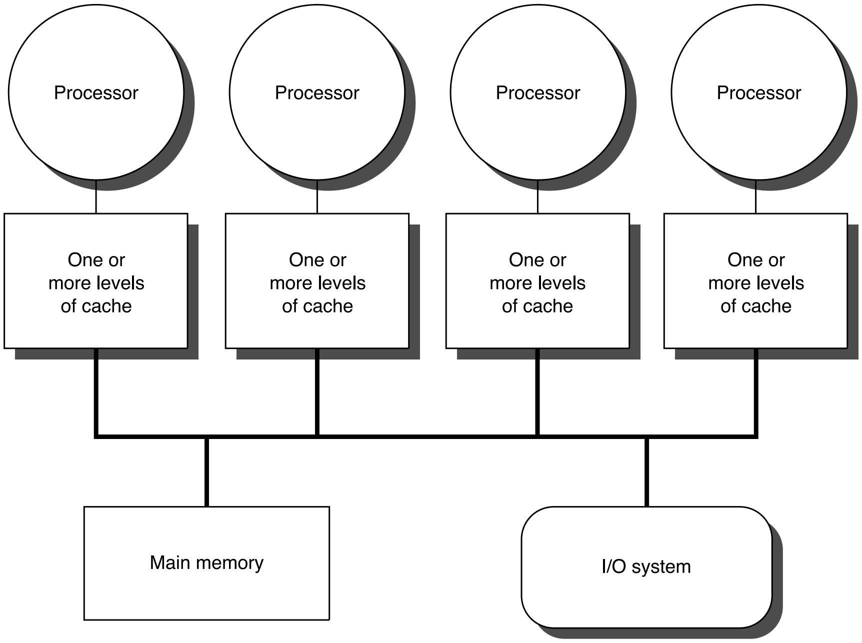

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

Figure 2.11: A shared-memory multiprocessor.

execution. Therefore, while the job does execute *at twice the speed on processor* s_1*, it does not* complete executing *in half the time.*

The above example illustrates a weakness in the model we use to describe these systems — we do not make a distinction between the *execution time* and the *total time to complete executing*. On identical SMPs, the memory access can be included in the execution time without causing any skew. However, on uniform heterogeneous multiprocessors, we need to make some adjustments. Recall that our model uses the *maximum* execution requirement, c, for each job. We do not assume that the job executes for c time units every time it is submitted — we only assume that it never executes for more than c time units. Below we describe a method for determining c if we are given the amount of time the job spends executing on a unit-speed processor and the amount of time spent accessing memory.

Assume a job J takes x units of time to execute on a unit-speed processor and y units of time to access memory. Then the total amount of time the job takes to complete execution on an s-speed processor is $x/s + y$. Clearly, the memory access becomes a larger proportion of the total time to complete as the processor speed increases. We can address the skew caused by the memory accesses by adjusting the job's memory access time. If we inflate y appropriately, then we can be sure the total processing time always remains within the proscribed limits. In particular, instead of saying that J's execution requirement is $(x + y)$, we should scale up the memory access time by the fastest processing speed, s_1, and set J's execution requirement to $(c = x + s_1 \cdot y)$. We can be sure that the total processing speed will never exceed this value. If J were to execute in a processor of speed s_k, then the total time to complete would be $(x/s_k + y)$. According to our model, the total time to complete would be $c/s_k = x/s_k + (s_1/s_k)y$. Therefore, the actual time to complete will never exceed the time the calculated by this method. Therefore, this strategy can be used to find each job's worst case execution requirement.

The time required to access memory also belies our assumption that jobs are independent. In order to access memory, the program must send a request to the memory, access the memory and send the contents back to the requesting program. Each of these steps involves accessing a resource that is *shared* among all the processors — namely the bus to send and receive the memory contents, or the memory module. In addition to accounting for the time required to operate each of the operations, the memory access time must include delay time due to *contention.* In some cases, the contention delay can dominate the memory access time. Throughout this discussion, the memory access time is assumed to include the *worst-case* contention delay.

Migration costs. Migrating a job from one processor to another incurs overhead. If the job were to remain on the same processor, it would be able to quickly access any cached information — both the program and its associated data may be cached. When the job restarts on a new processor, the cache in the new processor will not contain the necessary information. Therefore, both the program and its associated data will need to be accessed in memory instead of cache, which takes considerably more time. This added time is the cost associated with migration.

Of course, if migration is allowed the migration overhead will have to be included in the evaluation of the execution requirement. If the worst case migration costs are evaluated independently of the worst case execution requirements, they can simply be added to the execution costs and the scaled memory-access costs. In this case, there is no need to multiply the migration cost by the fastest processor speed, s_1. However, this may overestimate the execution requirement — the memory-access costs are evaluated assuming the job is executing on the fastest processor whenever it accesses memory and the migration costs are evaluated assuming the worst-case cost of migration is incurred. It is nearly impossible for the job to migrate among processors with all memory accesses occurring while the job is assigned to only one of the processors. The actual maximum overhead associated with memory accesses and migration combined would be much more difficult to find.

Consider the configuration $\{y_1, y_2, \ldots, y_m, r\}$, where y_i is the memory-access time that occurred while J was assigned to the i'th processor and r is the total migration time. The adjusted overhead associated with memory accesses and migration is

$$r + \sum_{i=1}^{m} s_i \cdot y_i \ ,$$

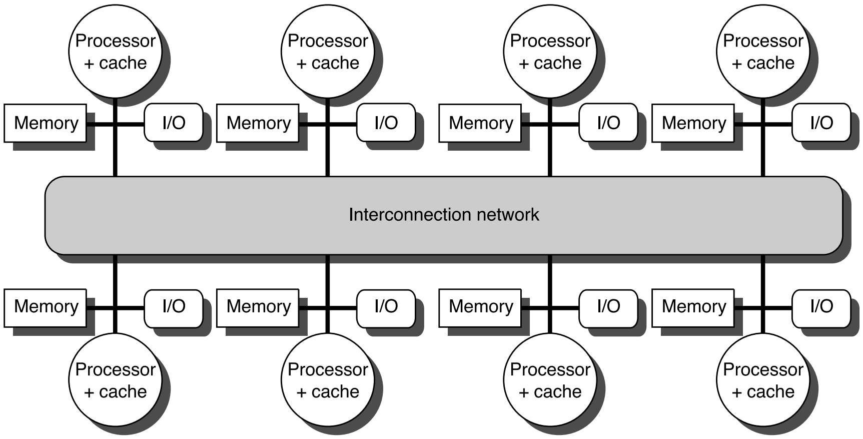
where s_i is the speed of the i'th processor of π. Therefore, the worst-case overhead is found by maximizing this expression over all valid configurations of memory accesses with their associated migration overheads. Determining this maximum is a complex problem that is beyond the scope of this dissertation. It falls in the area of worst-case execution time analysis, which is a rich area of research that is orthogonal to feasibility analysis, the subject of this dissertation. However, since uniform heterogeneous multiprocessors have not received much attention in real-time systems, we need to take note of how this architecture may affect worst-case execution time analysis.

2.3.2 Distributed memory multiprocessors

As indicated by the name, the memory in *distributed-memory multiprocessors* is distributed among the processors. The connection between processors can be via a bus or a more general network connection. Usually, a bus can only be used for smaller systems because bus traffic causes a bottleneck for larger systems [LLG$^+$92]. Figure 2.12 illustrates a distributed memory system in which each processor has its own associated memory. The SGI Origin 2000 [BFS89] is a distributed memory multiprocessor. As with SMPs, the processors in these systems may have different speeds. However, in these systems, the memories may also have different speeds. Therefore, in our calculations we need to use both of these speeds to evaluate the worst-case combination. We calculate the worst-case execution requirement by finding the total execution time, including memory access time, on each processor and scaling by that processor's speed. We then pick the maximum among the results. This is equivalent to scaling the normalized memory access time by the maximum ratio of the processing speed to memory access speed, as the following example illustrates.

Example 2.15 *Let J be a job that executes for 8 time units on a unit-speed processor and accesses memory for 4 time units on a unit-speed memory module. Assume J executes on a 2-processor distributed memory multiprocessor where the first processor's CPU speed is 8 and a memory access speed is 2 and the second processor's CPU speed is 2 and memory access speed is 1. The total execution time on processor 1 is* $(8/8+4/2 = 3)$*. Therefore the execution requirement on this system after scaling up by the processor speed is* $8 \cdot 3 = 24$*. The total execution time on processor 2 is* $(8/2 + 4/1 = 8)$ *and the scaled execution requirement is* $2 \cdot 8 = 16$*. Therefore, the worst-case execution requirement for this job on this system is 24.*

Notice that the maximum ratio between processing speed and memory access speed is $\max\{8/2, 2/1\} = 4$*, and the worst-case execution requirement we found was* $8 + 4 \cdot 4 = 24$*.*



© 2003 Elsevier Science (USA). All rights reserved.

Figure 2.12: A distributed memory multiprocessor.

It is easy to see that scaling the memory access requirement by this ratio will always find the worst-case execution requirement.

It is interesting to note that in the above example, the worst-case execution occurs on the first processor even though both the CPU- and the memory access speeds are faster on this processor. This is because the slower processor's memory access speed is closer to its processing speed. Therefore, memory accesses do not disrupt the execution progress to the same degree as they do on the faster processor, which has long memory access times relative to its executing times.

The migration calculations for distributed memory multiprocessors are similar to those for shared-memory multiprocessors. Either the worst-case migration costs should be added to the execution cost and the scaled memory access cost or the total overhead for both migration and memory access costs should be bounded by considering the worst-case configuration of migrations. On distributed memory multiprocessors, it is essential that when a job is migrated, its code and all its associated data is also migrated. This is because each migration has a large cost regardless of the size, and migration of large amounts of information does not take that much more time than migrating small amounts of information [LH86, BFS89].

2.4 Summary

This chapter has presented a variety of multiprocessor real-time scheduling results. Of course, this is a rich area of research and we have only touched on a few of the many interesting results on the subject of multiprocessor scheduling. The results presented here were chosen because they are related to the research presented in this dissertation. The next three chapters present EDF-schedulability tests for uniform heterogeneous multiprocessors under three different migration strategies. The results in these chapters build upon those presented above.

Chapter 3

Full migration EDF (f-EDF)

Recall that Theorems 1 and 2 in Chapter 2 state that no online job-level fixed-priority scheduling algorithm can be optimal for uniform heterogeneous multiprocessors [HL88]. Therefore, since deadlines do not change while jobs are active, EDF cannot be optimal on multiprocessors. Nonetheless, we can develop sufficient schedulability tests for EDF on uniform heterogeneous multiprocessors. In this chapter, we develop a sufficient schedulability test for EDF with full migration (f-EDF). In particular, this chapter describes the *characteristic region* associated with π, denoted CR_π.

Definition 6 (Characteristic region of π) *Let π be any uniform heterogeneous multiprocessor. Then the* characteristic region of π*, denoted CR_π, is the set of points (s, S) that satisfy the following property:*

> *Any real-time instance I that is feasible on some uniform heterogeneous multiprocessor π' with fastest speed $s_1(\pi') = s$ and total speed $S(\pi') = S$ is guaranteed to be* f-EDF*-schedulable on π.*

Example 3.1 *Figure 3.5 considers the multiprocessor $\pi = [50, 11, 4, 4]$ and divides all points (s, S) into three categories: **(i)** points known to be in CR_π, **(ii)** points known to be outside CR_π, and **(iii)** points whose membership in CR_π is unknown. For every uniform heterogeneous multiprocessor π' represented by a point in CR_π, feasibility on π' implies* f-EDF*-schedulability on π. For example, the region in this graph includes the point $(10, 20)$. Therefore, any real-time instance feasible on a uniform heterogeneous multiprocessor π' with $s_1(\pi') = 10$ and $S(\pi') = 20$ is* f-EDF*-schedulable on π. For every uniform heterogeneous multiprocessor π' represented by a point outside CR_π, feasibility on π' does not imply* f-EDF*-schedulability on π. For example, the point $(30, 65)$ is outside CR_π. Therefore, there is some real-time instance I and some uniform heterogenous multiprocessor π' with $s_1(\pi') = 30$ and $S(\pi') = 65$ such that I is feasible on π' but I is not* f-EDF*-schedulable on π.*

Verifying that an instance will meet all its deadlines when scheduled using a *specific* scheduling algorithm is often much more difficult than verifying that there is *some way* to successfully schedule the instance. Furthermore, π' is only described in terms of its fastest and total processing speeds, allowing for even more flexibility in determining the feasibility of I on π'. Therefore, the test that is presented in this chapter solves a complex problem — determining f-EDF-schedulability on the specific multiprocessor π — by examining a relatively easier problem — determining *feasibility* on a *loosely defined* multiprocessor π'.

This chapter is divided into three sections. Section 3.1 develops an f-EDF-schedulability test for uniform heterogeneous multiprocessors using resource augmentation techniques. In Section 3.2, we find that in some cases this test does not find the entire characteristic region. We will exploit the robustness of f-EDF on uniform heterogeneous multiprocessors to extend the original test and find the entire characteristic region CR_π associated with any uniform heterogeneous multiprocessor π. Finally, Section 3.6 uses the characteristic region to develop a utilization based f-EDF-schedulability test for periodic task sets on uniform heterogeneous multiprocessors.

3.1 An f-EDF-schedulability test

Resource augmentation techniques are used to find a portion of the characteristic region of π. Specifically, we assume that an instance I is known to be *feasible* (but not necessarily online schedulable) on some other uniform heterogeneous multiprocessor π'. We then show that if π is more powerful than π' by at least a certain amount f, then I is f-EDF-schedulable on π. The value of f depends on the processor speeds of both π and on π'.

Lemma 1 below specifies a condition upon the uniform heterogeneous multiprocessors π and π' under which any work-conserving algorithm A executing on π is guaranteed to complete at least as much work by each time instant t as any other algorithm A' executing on π' when both algorithms are scheduling the same real-time instance I. This condition depends on the value of $\lambda(\pi)$, the identicalness parameter associated with π, which was defined on page 6. Using this condition, we can use Lemma 1 to compare work done by EDF to work done by some optimal scheduling algorithm. Condition (3.1) expresses the amount by which the total computing capacity of π must exceed that of π' in order to ensure f-EDF-schedulability on π. Notice that the smaller the value of $\lambda(\pi)$, the smaller the amount of required excess processing capacity. Therefore, the minimum difference between the total computing capacities of the two multiprocessors is larger when π is closer to being an identical multiprocessor.

Lemma 1 *Let π and π' denote two uniform heterogeneous multiprocessors and let I be any real-time instance. Let S and S' denote schedules of I on π and π', respectively, and assume that S was generated by a work-conserving scheduling algorithm. If the following condition is satisfied,*

$$S(\pi) \geq \lambda(\pi) \cdot s_1(\pi') + S(\pi') \ , \tag{3.1}$$

then for any real-time instance I and any time instant $t \geq 0$,

$$W(\mathsf{S}, \pi, I, t) \geq W(\mathsf{S}', \pi', I, t) \ . \tag{3.2}$$

Proof: The proof is by contradiction. Suppose that Condition (3.2) does not hold; i.e., there is some real-time instance I and some time instant by which work-conserving schedule S on π has performed strictly less work than some other schedule S' on π'. Let $J_a = (r_a, c_a, d_a)$ be a job in I with the earliest arrival time such that there is some time instant t_o satisfying

$$\begin{aligned} W(\mathsf{S}, \pi, I, J_a, t_o) \quad &< \quad W(\mathsf{S}', \pi', I, J_a, t_o) \ , \ \text{and} && (3.3)\\ W(\mathsf{S}, \pi, I, t_o) \quad &< \quad W(\mathsf{S}', \pi', I, t_o) \ . && (3.4) \end{aligned}$$

By the choice of r_a, it must be the case that

$$W(\mathsf{S}, \pi, I, r_a) \geq W(\mathsf{S}', \pi', I, r_a) \ .$$

Therefore, the progress toward completion of the jobs of I in schedule S' during the interval $[r_a, t_o)$ is strictly greater than the progress in schedule S during the same interval.

For each $\ell = 1, 2, \ldots, m(\pi)$, let x_ℓ denote the cumulative length of time during the interval $[r_a, t_o)$ during which ℓ processors are executing jobs in schedule S. Observe the following:

- In schedule S, job J_a is active during the entire interval $[r_a, t_o)$. Therefore, there is always at least one busy processor during this interval. Hence,

$$t_o - r_a = x_1 + x_2 + \cdots + x_{m(\pi)} \ . \tag{3.5}$$

- Consider the behavior of J_a on π' during the interval $[r_a, t_o)$. In the best case, J_a is always executing on the fastest processor of π' in schedule S'. Therefore,

$$W(\mathsf{S}', \pi', I, J_a, t_o) \leq s_1(\pi') \cdot (t_o - r_a) \ . \tag{3.6}$$

- Now consider the behavior of J_a on π during the same interval. Since S was generated by a work-conserving scheduling algorithm, J_a must be executing whenever at least one processor is idle. However, J_a may be forced to wait whenever all $m(\pi)$ processors of π are busy. Therefore, J_a is only guaranteed to be executing for $(\sum_{j=1}^{m(\pi)-1} x_j)$ units of time. Furthermore, in the worst case, J_a executes the slowest processor that is executing some job. By the work-conserving property, if j processors are executing jobs, the slowest processor upon which J_a can be executing will have speed $s_j(\pi)$. Therefore,

$$W(\mathsf{S}', \pi', I, J_a, t_o) \geq \left(\sum_{j=1}^{m(\pi)-1} x_j s_j(\pi) \right) \ . \tag{3.7}$$

- In schedule S, the progress toward completion of the jobs of I is equal to $\sum_{j=1}^{m(\pi)} (x_j S_j(\pi))$ during $[r_a, t_o)$, while in schedule S' the progress is *at most* $(t_o - r_a) \cdot S(\pi')$ during this same interval. Thus, by Condition (3.4) the following condition holds:

$$\sum_{j=1}^{m(\pi)} (x_j S_j(\pi)) \quad < \quad (t_o - r_a) \cdot S(\pi') \ . \tag{3.8}$$

Combining Conditions (3.3), (3.6) and (3.7) gives

$$\sum_{j=1}^{m(\pi)-1} x_j s_j(\pi) < s_1(\pi') \cdot (t_o - r_a) \ . \tag{3.9}$$

Multiplying both sides of Condition (3.9) above by $\lambda(\pi)$, and noting that

$$\begin{aligned} (x_j \cdot s_j(\pi)\lambda(\pi)) &= \left(x_j \cdot s_j(\pi) \max_{1 \leq i \leq m(\pi)} \left\{ \frac{S(\pi) - S_i(\pi)}{s_i(\pi)} \right\} \right) \quad \text{(by definition of } \lambda) \\ &\geq \left(x_j \cdot s_j(\pi) \frac{S(\pi) - S_j(\pi)}{s_j(\pi)} \right) \\ &= x_j (S(\pi) - S_j(\pi)) \ , \end{aligned}$$

gives

$$\sum_{j=1}^{m(\pi)-1} (x_j \cdot (S(\pi) - S_j(\pi))) \quad < \quad s_1(\pi') \cdot \lambda(\pi) \, (t_o - r_a) \ . \tag{3.10}$$

Adding Conditions (3.8) and (3.10) gives

$$\sum_{j=1}^{m(\pi)} (x_j S_j(\pi)) + \sum_{j=1}^{m(\pi)-1} (x_j (S(\pi) - S_j(\pi))) < (t_o - r_a) \left(s_1(\pi') \cdot \lambda(\pi) + S(\pi') \right) \ .$$

Combining the two summations gives

$$x_m S(\pi) + \sum_{j=1}^{m(\pi)-1} [x_j (S_j(\pi) + S(\pi) - S_j(\pi))] < (t_o - r_a) \left(s_1(\pi') \cdot \lambda(\pi) + S(\pi') \right) \ .$$

Cancelling the $S_j(\pi)$ terms gives

$$x_m S(\pi) + \sum_{j=1}^{m(\pi)-1} x_j S(\pi) < (t_o - r_a)\left(s_1(\pi') \cdot \lambda(\pi) + S(\pi')\right) .$$

Combining the two left-hand side terms gives

$$S(\pi) \cdot \sum_{j=1}^{m(\pi)} x_j < (t_o - r_a)\left(s_1(\pi') \cdot \lambda(\pi) + S(\pi')\right) .$$

Substituting Condition (3.5) gives

$$S(\pi) \cdot (t_o - r_a) < (t_o - r_a)(s_1(\pi') \cdot \lambda(\pi) + S(\pi')) .$$

Dividing by $(t_o - r_a)$ gives

$$S(\pi) < s_1(\pi') \cdot \lambda(\pi) + S(\pi') ,$$

which contradicts the assumption made in the statement of the lemma (Condition (3.1)). ■

Lemma 1 allows us to reason about the total execution of work-conserving algorithms. The next theorem shows that we can use this knowledge to deduce whether a particular work-conserving algorithm (namely, f-EDF) can feasibly schedule a real-time instance. It states that if uniform heterogeneous multiprocessors π and π' satisfy Condition (3.1) of Lemma 1 then any real-time instance I that is feasible on π' will be f-EDF-schedulable on π.

Theorem 16 *Let I denote a real-time instance that is feasible on some uniform heterogeneous multiprocessor π', and let π denote another uniform heterogeneous multiprocessor. If Condition* (3.1) *of Lemma 1 is satisfied – i.e., $S(\pi) \geq \lambda(\pi) \cdot s_1(\pi') + S(\pi')$ — then I is* f-EDF*-schedulable on π.*

Proof: Let opt be a valid schedule of I on π'. Since π and π' satisfy Condition (3.1), it follows from Lemma 1 that the work done by f-EDF scheduling I on π during the interval $[0, t)$ is at least as much as the work done by opt scheduling I on π' during the same interval:

$$W(\textsf{f-EDF}, \pi, I, t) \geq W(\textsf{opt}, \pi', I, t) \quad \text{for all } t \geq 0 .$$

This condition can be used to prove inductively that f-EDF schedules I to meet all deadlines on π. The induction is on the number of jobs in I. Specifically, let $I_k \overset{\text{def}}{=} \{J_1, \ldots, J_k\}$ denote the k jobs of I with the highest f-EDF-priority.

Claim. I_k is f-EDF-schedulable on π for all $1 \leq k \leq |I|$.

Base case. Since I_o denotes the empty set, $\mathsf{f\text{-}EDF}$ clearly schedules I_o to meet all deadlines on π.

Induction step. Assume that $\mathsf{f\text{-}EDF}$ can schedule I_k on π for some k with $I_k \neq I$ and consider the $\mathsf{f\text{-}EDF}$-generated schedule of I_{k+1} on π. Note that $I_k \subset I_{k+1}$ and that the job J_{k+1} does not effect the scheduling decisions made by $\mathsf{f\text{-}EDF}$ on the jobs $\{J_1, J_2, \ldots, J_k\}$ while it is scheduling I_{k+1}. That is, the schedule generated by $\mathsf{f\text{-}EDF}$ for $\{J_1, J_2, \ldots, J_k\}$ while scheduling I_{k+1}, is identical to the schedule generated by $\mathsf{f\text{-}EDF}$ while scheduling I_k. Hence by the induction hypothesis, the k highest priority jobs of I_{k+1} all meet their deadlines. It remains to prove that J_{k+1} also meets its deadline.

Consider the schedules generated by opt executing on π'. Since I is assumed to be feasible on π', it follows that I_{k+1} is also feasible on π' and hence opt will schedule I_{k+1} on π' to meet all deadlines. That is,

$$W(\mathsf{opt}, \pi', I_{k+1}, d_{k+1}) = \sum_{i=1}^{k+1} c_i \ ,$$

where d_{k+1} denotes the deadline of J_{k+1}, which is the latest deadline of all the jobs of I_{k+1}. By Lemma 1,

$$W(\mathsf{f\text{-}EDF}, \pi, I_{k+1}, d_{k+1}) \geq W(\mathsf{opt}, \pi', I_{k+1}, d_{k+1}) = \sum_{i=1}^{k+1} c_i \ .$$

Since the total execution requirement of all the jobs in I_{k+1} is $\sum_{i=1}^{k+1} c_i$ it follows that job J_{k+1} meets its deadline.

Therefore, $\mathsf{f\text{-}EDF}$ successfully schedules all the jobs of I_{k+1} to meet their deadlines on π. ■

The result of Phillips, Stein, Torng, and Wein [PSTW97] concerning $\mathsf{f\text{-}EDF}$-scheduling on identical multiprocessors is an immediate corollary to Theorem 16 above.

Corollary 1 *If a set of jobs is feasible on an identical m-processor multiprocessor, then the same set of jobs will be scheduled to meet all deadlines by* $\mathsf{f\text{-}EDF}$ *on an identical m-processor multiprocessor in which the individual processors are $(2 - \frac{1}{m})$ times as fast as in the original platform.*

Proof: Assume the multiprocessors π and π' of the statement of Theorem 16 are each comprised of m identical multiprocessors. Let the speeds of the processors of π and π' be s and s' respectively. The parameter $\lambda(\pi)$ evaluates to $(m(\pi) - 1)$:

$$\lambda(\pi) \stackrel{\text{def}}{=} \max_{1 \leq j \leq m(\pi)} \left\{ \frac{\sum_{k=j+1}^{m(\pi)} s_k(\pi)}{s_j(\pi)} \right\} \quad = \quad \max_{1 \leq j \leq m(\pi)} \left\{ \frac{(j-1)s}{s} \right\} \quad = \quad m(\pi) - 1 \ .$$

Substituting this into the condition $S(\pi) \geq \lambda(\pi) \cdot s_1(\pi') + S(\pi')$ from the statement of Theorem 16 gives

$$m(\pi) \cdot s \geq (m(\pi) - 1) \cdot s' + m(\pi') \cdot s' \ .$$

By assumption, $m(\pi) = m(\pi') = m$. Therefore

$$m \cdot s \geq (m - 1) \cdot s' + m \cdot s' \ .$$

Combining the two right-hand side terms gives

$$m \cdot s \geq (2m - 1) \cdot s' \ .$$

Dividing by m gives

$$s \geq \left(2 - \frac{1}{m}\right) \cdot s' \ ,$$

from which the corollary follows. ■

3.2 The Characteristic Region of π (CR_π)

Section 3.1 introduces a test to determine f-EDF-schedulability of an instance that is known to be feasible on some other multiprocessor π'. Chapter 2 discusses the robustness of f-EDF with respect to multiprocessor domination. This section combines these two methods to determine the characteristic region of π.

Theorem 16 can be used to determine some of the points in CR_π — if Condition (3.1) of this theorem is satisfied for some π', then any instance feasible on π' is also feasible on π. Of course, since $s_1(\pi')$ is a processor speed, it must be a positive value. Also, since $S(\pi')$ is the cumulative processing power, $S(\pi')$ must be at least as large as $s_1(\pi')$. Condition (3.1) combined with these restrictions on $s_1(\pi')$ and $S(\pi')$ can be used to define a set R_π that is a subset of CR_π.

Definition 7 (**R_π**) *Let π be any uniform heterogeneous multiprocessor. The set of points satisfying the conditions of Theorem 16 is denoted R_π. Specifically,*

$$R_\pi \overset{def}{=} \{(s, S) \mid 0 < s \leq S \text{ and } S \leq S(\pi) - \lambda(\pi) \cdot s\}.$$

Example 3.2 *Let $\pi = [50, 11, 4, 4]$. Then $\lambda(\pi) = \max\{19/50, 8/11, 4/4\} = 1$ and $S(\pi) = 69$. Therefore, $R_\pi = \{(s, S) \mid 0 < s \leq S \leq 69 - s\} \subseteq CR_\pi$. Figure 3.1 illustrates this area. If (s, S) is in the shaded area and I is feasible on some π' with $s_1(\pi') = s$ and $S(\pi') = S$ then I is* f-EDF*-schedulable on π.*

Unfortunately, Figure 3.1 does not illustrate the entire characteristic region of π. Consider,

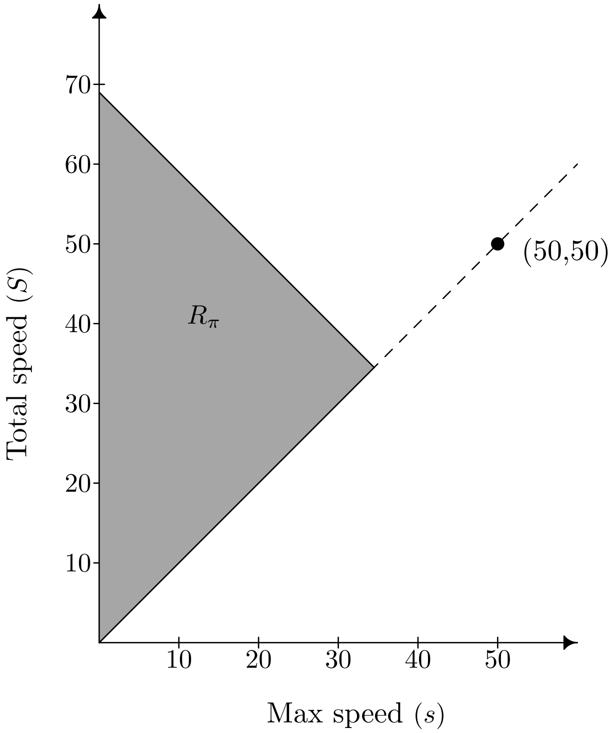


Figure 3.1: The region R_π for $\pi = [50, 11, 4, 4]$ contains all points (s, S) with $S \leq S(\pi) - s \cdot \lambda(\pi)$. Any instance I is guaranteed to be f-EDF-schedulable on π if I is feasible on some multiprocessor with fastest speed s and total speed S, where (s, S) is in the region R_π.

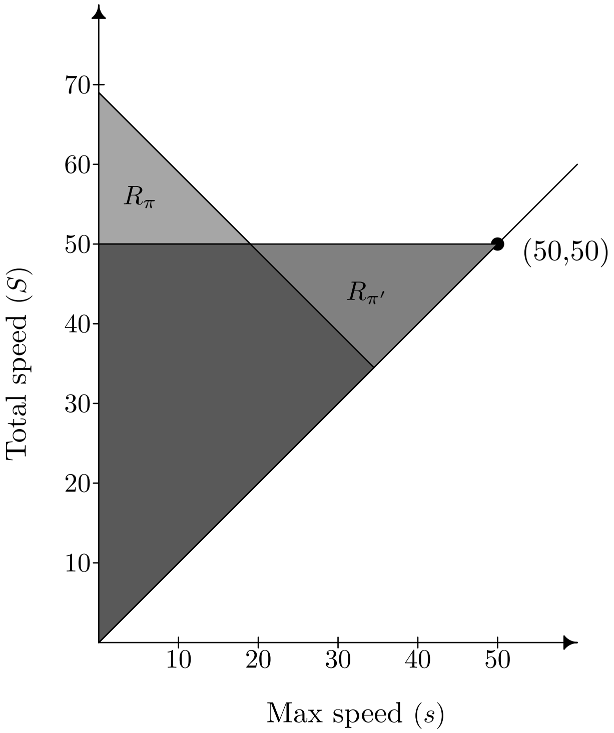


Figure 3.2: The regions associated with $\pi = [50, 11, 4, 4]$ and $\pi' = [50]$.

for example, the point $(50, 50)$, which is not in the region R_π. This point represents the *uniprocessor* with speed 50. If we let π' be this uniprocessor, then any instance I feasible on π' is clearly EDF-schedulable on π' by the optimality of EDF on uniprocessors. Since EDF and f-EDF generate the same schedule on uniprocessors, any instance I that is feasible on π' must also be f-EDF-schedulable on π'. Furthermore, since $\pi \overset{*}{\trianglerighteq} \pi'$ and f-EDF is robust on uniform heterogeneous multiprocessors, any instance f-EDF-schedulable on $\pi' = [50]$ must also be f-EDF-schedulable on $\pi = [50, 11, 4, 4]$. Figure 3.2 illustrates R_π and $R_{\pi'}$. Notice there are *many* points in $R_{\pi'}$ that are not in R_π. Therefore, the region R_π must be a *proper* subset of CR_π.

This demonstrates that Theorem 14 in Chapter 2 can be used to extend the region provided by Theorem 16 above. In fact, this technique provides us with the following lower bound on the region CR_π. Recall π cleanly dominates π^* (denoted $\pi \overset{*}{\trianglerighteq} \pi^*$) if $m(\pi) \geq m(\pi^*)$ and $s_i(\pi) = s_i(\pi^*)$ for every $i = 1, 2, \ldots, m(\pi^*) - 1$ and $s_{m(\pi^*)}(\pi) \geq s_{m(\pi^*)}(\pi^*)$.

Definition 8 (cr_π) *Let π be any uniform heterogeneous multiprocessor. The set of points $(s, S) \in R_{\pi^*}$ for some π^* cleanly dominated by π is denoted cr_π:*

$$cr_\pi \overset{def}{=} \bigcup_{\pi \overset{*}{\trianglerighteq} \pi^*} R_{\pi^*} \ .$$

According to this definition, all the points in shaded areas of Figure 3.2 are in cr_π. In addition, there may be other points in cr_π that are included due to other multiprocessors dominated by π. For example, consider the multiprocessor $\pi^* = [50, 11, 3]$. Clearly, $\pi \trianglerighteq^* \pi^*$. In this case $\lambda(\pi^*) = \max\{14/50, 3/11\} = 14/50$. Since $64 - 30 \cdot 14/50 = 55.6$, the point (30,55) is in R_{π^*}. Therefore, this point is also in cr_π even though it is not in either shaded region of Figure 3.2.

Clearly, cr_π is a subset of CR_π. In addition to finding points in CR_π, we need to determine what points are outside of CR_π. While Section 3.3 finds an efficient way to describe all the points in cr_π, Section 3.4 proves that many points outside cr_π are also outside CR_π. Section 3.5 describes the points whose membership in CR_π is currently unknown.

3.3 Finding the subset cr_π of CR_π

Recall that cr_π is the union of the regions R_{π^*} where π cleanly dominates π^*. Lemma 2 below shows the following set of points must be above or on the border of R_{π^*} for any π^* such that $\pi \overset{*}{\trianglerighteq} \pi^*$.

Definition 9 (A_π) *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor. Then*

$$A_\pi \overset{def}{=} \{(s_1, S_1), (s_2, S_2), \ldots, (s_m, S_m), (s_{m+1}, S_{m+1})\}, \text{ where } (s_{m+1}, S_{m+1}) \overset{def}{=} (0, S(\pi)) \ .$$

Each of the first m points of A_π may be viewed as a prefix of π. If we consider $\pi_i = [s_1, s_2, \ldots, s_i]$ to be the uniform heterogeneous multiprocessor comprised of the i fastest processors of π, then the first m points in A_π pair the slowest processor speed of π_i with $S(\pi_i)$. The $(m+1)^{st}$ point can be viewed the same way if we add a "zero-speed" processor to π. Of course, no processor can have a speed of 0. Nonetheless, this point belongs in A_π because, as we shall see below, it is one of the points that forms an upper bound for CR_π.

Example 3.3 *Let $\pi = [50, 11, 4, 4]$. Then $\pi_1 = [50]$ and the first point in A_π is (50,50). Similarly, $\pi_2 = [50, 11]$ and the second point in A_π is (11,61). In total A_π contains $m(\pi)+1 =$ 5 points, $A_\pi = \{(50, 50), (11, 61), (4, 65), (4, 69), (0, 69)\}$.*

Lemma 2 *Let π be any uniform heterogeneous multiprocessor and let $(s_i(\pi), S_i(\pi))$ be some point in A_π. Then for any π^* such that $\pi \overset{*}{\trianglerighteq} \pi^*$*

$$S_i(\pi) \ \geq \ S(\pi^*) - \lambda(\pi^*) \cdot \ s_i(\pi) \ . \tag{3.11}$$

Proof: The theorem is proved considering three cases depending on the value of $S(\pi^*)$.

Case 1: $S(\pi^*) < s_1(\pi)$**.** Since $\pi \overset{*}{\trianglerighteq} \pi^*$, we know that the speed each processor speed of each processor of π^* — except, possibly the slowest processor — is the same as the speed of the corresponding processor of π. Since $S(\pi^*) < s_1(\pi)$, it follows that π^* the must be a uniprocessor. Therefore, $\lambda(\pi^*) = 0$, and

$$S(\pi^*) - \lambda(\pi^*) \cdot s_i(\pi) = S(\pi^*) \ .$$

Furthermore, for all i with $1 \leq i \leq m+1$,

$$S_i(\pi) > s_1(\pi) > S(\pi^*) \ .$$

Combining these two conditions gives

$$S_i(\pi) \ > \ S(\pi^*) - \lambda(\pi^*) \cdot \ s_i(\pi)$$

so Condition (3.11) holds for every $i = 1, 2, \ldots, m+1$ whenever $S(\pi^*) < s_1(\pi)$.

Case 2: $S(\pi^*) = S(\pi)$**.** In this case π and π^* must have the exact same parameters — i.e., $m(\pi) = m(\pi^*)$ and $s_i(\pi) = s_i(\pi^*)$ for all i such that $1 \leq i \leq m(\pi)$. Recall that $\lambda(\pi) = \max_{1 \leq i \leq m}\{\frac{\sum_{k=i+1}^{m} s_k(\pi)}{s_i(\pi)}\}$ Therefore, for all i such that $1 \leq i \leq m$, the following holds

$$\lambda(\pi) \geq \frac{\sum_{k=i+1}^{m} s_k(\pi)}{s_i(\pi)} \ .$$

Multiplying both sides of this condition by $s_i(\pi)$ and noting that $\sum_{k=1+i}^{m} s_k(\pi) = S(\pi) - S_i(\pi)$ gives

$$\lambda(\pi) \cdot s_i(\pi) \geq S(\pi) - S_i(\pi) \ .$$

Adding $S_i(\pi) - \lambda(\pi) \cdot s_i(\pi)$ to both sides of this inequality gives

$$S_i(\pi) \geq S(\pi) - \lambda(\pi) \cdot s_i(\pi) \ .$$

Since π and π^* have the same processor speeds, we know that $\lambda(\pi) = \lambda(\pi^*)$. Substituting $\lambda(\pi^*)$ for $\lambda(\pi)$ gives

$$S_i(\pi) \geq S(\pi^*) - \lambda(\pi^*) \cdot s_i(\pi) \ .$$

Case 3: $S(\pi) > S(\pi^*) \geq s_1(\pi)$**.** Let β be the largest index such that $s_\beta(\pi) = s_\beta(\pi')$. Since $S(\pi) > S(\pi^*) \geq s_1(\pi)$, it follows that $m(\pi) > \beta \geq 1$. Consider the following two possibilities for the value of i.

Case 3a: $\beta < i \leq m(\pi) + 1$**.** By definition of β, $S_i(\pi) > S(\pi^*)$. Furthermore, since $\lambda(\pi^*) \geq 0$, we know $S(\pi^*) \geq S(\pi^*) - \lambda(\pi^*) \cdot s_i(\pi)$. Therefore, Condition (3.11) holds for every i such that $\beta < i \leq m(\pi) + 1$.

Case 3b: $1 \leq i \leq \beta$**.** Assume Condition (3.11) does not hold for some point $(s_i(\pi), S_i(\pi))$. Then

$$S_i(\pi) < S(\pi^*) - \lambda(\pi^*) \cdot s_i(\pi) \ .$$

Since $\pi \overset{*}{\trianglerighteq} \pi^*$ and $i \leq \beta$, we know $s_i(\pi) = s_i(\pi^*)$ and $S_i(\pi) = S_i(\pi^*)$. Substituting into the inequality above gives

$$S_i(\pi^*) < S(\pi^*) - \lambda(\pi^*) \cdot s_i(\pi^*) \ .$$

Adding $\lambda(\pi^*) \cdot s_i(\pi^*) - S_i(\pi^*)$ to both sides of the inequality gives

$$\lambda(\pi^*) \cdot s_i(\pi^*) < S(\pi^*) - S_i(\pi^*) \ .$$

Dividing by $s_i(\pi^*)$ gives

$$\lambda(\pi^*) < \frac{\sum_{i=i+1}^{m(\pi^*)} s_i(\pi^*)}{s_i(\pi^*)} \ ,$$

which violates the definition of $\lambda(\pi^*)$. Therefore, Condition (3.11) holds for every i such that $1 \leq i \leq \beta$.

Since the three cases above are exhaustive, Condition (3.11) holds. ■

Lemma 2 states that the points in A_π are either above or on the upper border of R_{π^*} for any π^* such that $\pi \overset{*}{\trianglerighteq} \pi^*$. Therefore, A_π may be useful in understanding the shape of cr_π. Below we define a specific subset, H_π, of A_π. This subset is the lower portion of the convex hull of the points of A_π. The piecewise linear function $L_\pi(s)$ formed by connecting consecutive points in H_π can be used to fully describe cr_π.

Definition 10 **($H_\pi, L_\pi(s)$)** *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor. Define the set*

$$H_\pi \overset{def}{=} \{(s_{h_1}, S_{h_1}), (s_{h_2}, S_{h_2}), \ldots, (s_{h_{m'}}, S_{h_{m'}})\} \subseteq A_\pi$$

recursively as follows.

- $s_{h_1} = s_1(\pi)$.
- *If $s_{h_i} \neq 0$, then $s_{h_{i+1}}$ is the slowest processor speed less than s_{h_i} such that all points in A_π are either on or above the line defined by (s_{h_i}, S_{h_i}) and $(s_{h_{i+1}}, S_{h_{i+1}})$. Specifically,*

$$s_{h_{i+1}} = \min\left\{ s_k(\pi) < s_{h_i} \mid S_\ell(\pi) \geq S_k - \frac{S_k - S_{h_i}}{s_{h_i} - s_k}(s_k - s_\ell) \forall \ell = 1, 2, \ldots, m+1 \right\} \ .$$

- *If $s_{h_i} = 0$ then $i = m'$ (thus, H_π is completely defined).*

Finally, define the function $L_\pi(s)$ with domain $0 < s \leq s_1(\pi)$ to be the piecewise linear

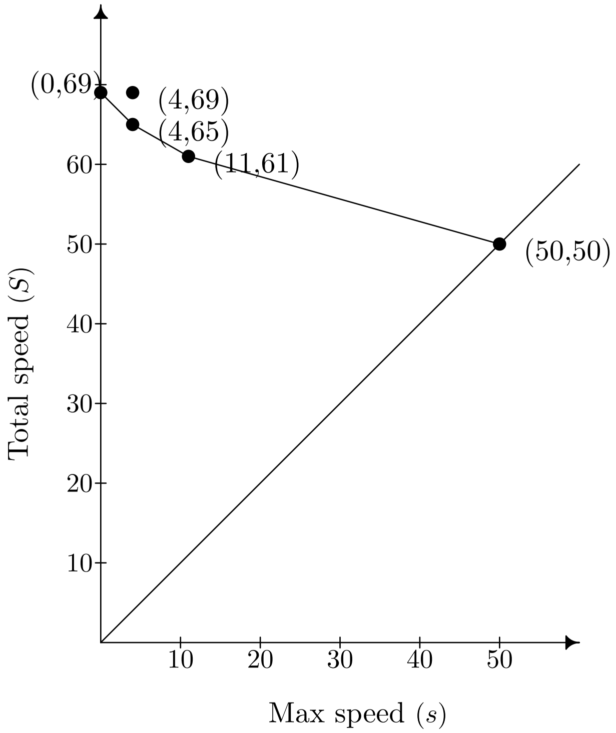


Figure 3.3: The set A_π and the function $L_\pi(s)$ for $\pi = [50, 11, 4, 4]$.

function found by joining consecutive points in H_π.

$$L_\pi(s) = S_{h_{i+1}} - \frac{S_{h_{i+1}} - S_{h_i}}{s_{h_i} - s_{h_{i+1}}}(s - s_{h_{i+1}}) \text{ where } s_{h_{i+1}} \leq s \leq s_{h_i} \text{ and } i = 1, 2, \ldots, m' - 1.$$

Figure 3.3 illustrates the points in A_π and the function $L_\pi(s)$ for $\pi = [50, 11, 4, 4]$. H_π is comprised of all the points along the line $L_\pi(s)$ — i.e., all of A_π except the point $(4, 69)$.

The remainder of this section proves that for every uniform heterogeneous multiprocessor π, $L_\pi(s)$ is the upper bound of cr_π. Lemma 4 below states that each segment of $L_\pi(s)$ is a segment of the upper bound of R_{π^*} for some $\pi \overset{*}{\trianglerighteq} \pi$. Theorem 17 uses this result to show that

$$cr_\pi = \{(s, S) \mid (0 < s \leq s_1) \wedge (s \leq S \leq L_\pi(s))\}.$$

These results rely on the following lemma (Lemma 3) which proves specific properties about the upper bound of R_π for any uniform heterogeneous multiprocessor π.

Lemma 3 *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor and let $\mathtt{L}$ be a line on the (s, S)-plane with the following properties:*

- *$\mathtt{L}$ contains the points $(0, S(\pi))$ and (s_k, S_k) for some $k = 1, 2, \ldots, m$, and*

- *All points in A_π are on or above* L.

Then the equation of the line L *is* $S = S(\pi) - \lambda(\pi) \cdot s$. *Moreover,* $\lambda(\pi) = (\sum_{i=k+1}^{k} s_i)/s_k$.

Proof: The slope of L is $(S(\pi) - S_k)/(0 - s_k) = -(\sum_{i=k+1}^{m} s_i)/s_k$. Therefore the equation of this line is $S = S(\pi) - \frac{\sum_{i=k+1}^{m} s_i}{s_k} \cdot s$. Since all points in A_π are on or above L, the following inequalities must hold for $\ell = 1, 2, \ldots, m$.

$$\begin{aligned} S_\ell &\geq S(\pi) - \frac{\sum_{i=k+1}^{m} s_i}{s_k} \cdot s_\ell \\ \Rightarrow \frac{\sum_{i=k+1}^{m} s_i}{s_k} \cdot s_\ell &\geq S(\pi) - S_\ell \\ \Rightarrow \frac{\sum_{i=k+1}^{m} s_i}{s_k} &\geq \frac{\sum_{i=\ell+1}^{m} s_i}{s_\ell}. \end{aligned}$$

Therefore, $\frac{\sum_{i=k+1}^{m} s_i}{s_k} = \max_{1 \leq \ell \leq m} \left\{ \frac{\sum_{i=\ell+1}^{m} s_i}{s_\ell} \right\} = \lambda(\pi)$. ■

Lemma 4 *Let π be any uniform heterogeneous multiprocessor and let the lower portion of the convex hull of A_π be $H_\pi = \{(s_{h_1}, S_{h_1}), (s_{h_2}, S_{h_2}), \ldots, (s_{h_{m'}}, S_{h_{m'}})\}$. Then for every i such that $1 \leq i < m'$, there exists a multiprocessor π^* such that $\pi \overset{*}{\trianglerighteq} \pi^*$, and both the points (s_{h_i}, S_{h_i}) and $(s_{h_{i+1}}, S_{h_{i+1}})$ are on the line $S = S(\pi^*) - \lambda(\pi^*) \cdot s$.*

Proof: Consider any two consecutive points in H_π, namely (s_{h_i}, S_{h_i}) and $(s_{h_{i+1}}, S_{h_{i+1}})$. Let L be the line defined by these points. Then the equation of this line is $S = S_{h_{i+1}} - \frac{S_{h_{i+1}} - S_{h_i}}{s_{h_i} - s_{h_{i+1}}}(s - s_{h_{i+1}})$. Let α be the S-intercept of L — *i.e.*,

$$\alpha = S_{h_{i+1}} + s_{h_{i+1}} \cdot \frac{S_{h_{i+1}} - S_{h_i}}{s_{h_i} - s_{h_{i+1}}} . \tag{3.12}$$

Since the point $(0, S(\pi))$ is in A_π, it must be on or above L. Therefore, $\alpha \leq S(\pi)$ and there exists a uniform heterogeneous multiprocessor cleanly dominated by π with total capacity α — let π^* be this multiprocessor.

Note that $S_{h_{i+1}} \geq S_{h_i}$ and $s_{h_i} > s_{h_{i+1}}$. Therefore, by Equation (3.12), $\alpha \geq S_{h_{i+1}} \geq s_1(\pi)$. Furthermore, note that $\alpha \geq S_{h_{i+1}} > S_{h_i}$. Therefore, (s_{h_i}, S_{h_i}) and $(s_{h_{i+1}}, S_{h_{i+1}})$ are both in A_{π^*}. Thus, L contains the points $(0, S(\pi^*))$ and $(s_k(\pi^*), S_k(\pi^*))$ for some $k = 1, 2, \ldots, m(\pi^*)$. If all points in A_{π^*} can be proven to be on or above L then, by Lemma 3, L *must* be the line $S = S(\pi^*) - \lambda(\pi^*) \cdot s$.

Since $\pi \overset{*}{\trianglerighteq} \pi^*$, at most two points in A_{π^*} are not in A_π — namely $(s_{m(\pi^*)}(\pi^*), S(\pi^*))$ and $(0, S(\pi^*))$. Therefore, since all points in A_π are known to be on or above L and $(0, S(\pi^*))$ is known to be on L, $(s_{m(\pi^*)}(\pi^*), S(\pi^*))$ is the only point in A_{π^*} whose relationship to L is unknown. Clearly, this point is above L because the S-intercept of L is $(0, S(\pi^*))$ and its slope is negative. Therefore, by Lemma 3, L must be the line $S = S(\pi^*) - \lambda(\pi^*) \cdot s$. ■

By Lemma 4 we see that all points on or below $L_\pi(s)$ are in some region R_{π^*} for some π^* cleanly dominated by π. It remains to be shown that no other points can be in any region R_{π^*} .

Theorem 17 *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor and let the lower portion of the convex hull of A_π be $H_\pi = \{(s_{h_1}, S_{h_1}), (s_{h_2}, S_{h_2}), \ldots, (s_{h_{m'}}, S_{h_{m'}})\}$. Define the function $L_\pi(s)$ to be the piecewise linear function connecting the points of H_π. Let*

$$Q = \{(s, S) \mid (0 < s \leq s_1) \wedge (s \leq S \leq L_\pi(s))\}.$$

Then $Q = cr_\pi$.

Proof: Lemma 4 demonstrates that $Q \subseteq cr_\pi$. Clearly $0 < s \leq S$ for any $(s, S) \in cr_\pi$ because processor speeds must be positive and S is the total of positive values including s. Also s cannot be greater than s_1 since the 1-job instance $\{(0, s, 1)\}$ will be schedulable on any multiprocessor with fastest speed equal to s, but it will be feasible on π only if $s \leq s_1$. It remains to show that $S \leq L_\pi(s)$ for any $(s, S) \in cr_\pi$.

Assume there is some multiprocessor π^* and some point $(\hat{s}, \hat{S})$ such that

- $\pi \overset{*}{\trianglerighteq} \pi^*$,
- $\hat{S} = S(\pi^*) - \lambda(\pi^*) \cdot \hat{s}$, and
- $\hat{S} > L_\pi(\hat{s})$.

Let L denote the line $S = S(\pi^*) - \lambda(\pi^*) \cdot\ s$. Notice that L contains the points $(0, S(\pi^*))$ and $(\hat{s}, \hat{S})$. Since $L_\pi(0) = S(\pi) \geq S(\pi^*)$ and $L_\pi(\hat{s}) < \hat{S}$, L and $L_\pi(s)$ must have at least one point of intersection between 0 and $\hat{s}$. Furthermore, this must be a single point of intersection, because no line containing a segment of $L_\pi(s)$ can have any point above $L_\pi(s)$ (by the construction of $L_\pi(s)$). Let (s^*, S^*) be the point of intersection between the two lines. Note that for $s > s^*$, $L_\pi(s)$ is below L. Since two straight non-collinear lines can have only one point of intersection, $L_\pi(s)$ must remain below L until its slope changes. But the slope changes only at points in $H_\pi \subseteq A_\pi$. This implies that there is a point in A_π below L. This contradicts Lemma 2. Therefore, there can be no point in cr_π above $L_\pi(s)$ ■

Theorem 17 provides a simple method for determining a lower bound for the characteristic region of π — namely find the convex hull of the points in A_π. Figure 3.4 shows this region for the uniform heterogeneous multiprocessor $\pi = [50, 11, 4, 4]$. The next section proves that in many cases this is the entire characteristic region of π.

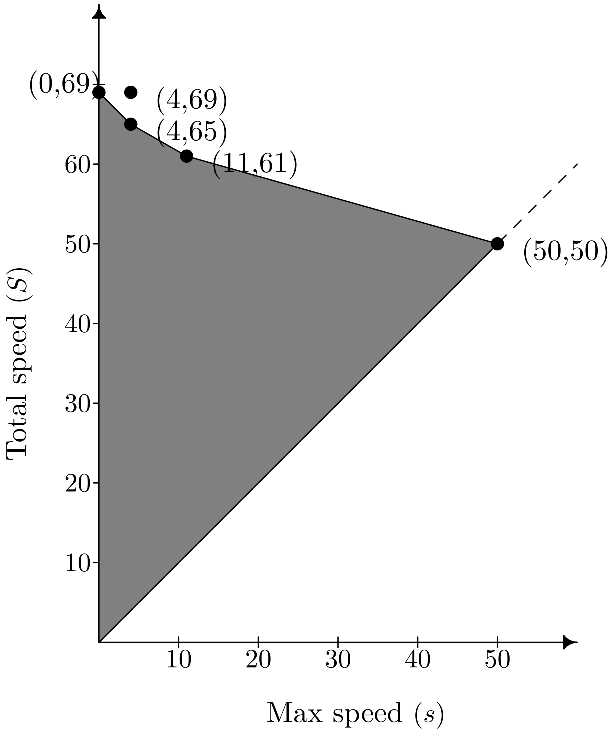


Figure 3.4: The region cr_π for $\pi = [50, 11, 4, 4]$

3.4 Finding points outside CR_π

While we have shown that $cr_\pi \subseteq CR_\pi$, we have not yet shown that any points *outside* cr_π are definitely not in CR_π. The points shown to be in both cr_π and CR_π are all on or below the line $L_\pi(s)$. In this section, we show that for any $1 < k \leq m + 1$, all points above the line between (s_1, s_1) and (s_k, S_k) are not in CR_π. We do this by finding, for any point $(\hat{s}, \hat{S})$ above the line between (s_1, s_1) and (s_k, S_k), a specific a uniform heterogeneous multiprocessor π' with $s_1(\pi') = \hat{s}$ and $S(\pi') = \hat{S}$, and a specific real-time instance I such that I is feasible on π' but is not f-EDF-schedulable on π.

Theorem 18 *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor and let $s_k < \hat{s} \leq s_1$ for some $1 < k \leq m + 1$. If $(\hat{s}, \hat{S})$ is above the line between (s_1, s_1) and (s_k, S_k) then $(\hat{s}, \hat{S}) \notin CR_\pi$.*

Proof: The proof is by construction. For any point $(\hat{s}, \hat{S})$ above the given line, we find a real-time instance I and a uniform heterogeneous multiprocessor π' with $s_1(\pi') = \hat{s}$ and $S(\pi') = \hat{S}$ such that I is feasible on π' but is not f-EDF-schedulable on π.

The equation of the line between (s_1, s_1) and (s_k, S_k) is

$$f(s) = s_1 + \frac{S_k - s_1}{s_1 - s_k} \cdot (s_1 - s).$$

Define ρ as follows

$$\rho \overset{\text{def}}{=} \left\lceil \frac{s_1(\hat{S} - \hat{s})}{\hat{s} S_{k-1}} \right\rceil .$$

Let π' be the uniform heterogeneous multiprocessor consisting of $((\rho - 1)m + k)$ processors with the following speeds:

1 processor with speed $\hat{s}$,

1 processor with speed $\frac{s_i(\hat{S}-\hat{s})}{\rho S_{k-1}}$ for each $1 \leq i \leq k-1$,

$(\rho - 1)$ processors with speed $\frac{s_i(\hat{S}-\hat{s})}{\rho S_m}$ for each $1 \leq i \leq m$.

Claim: $u_{max}(\pi') = \hat{s}$ and $U_{sum}(\pi') = \hat{S}$.

Proof of claim: By construction,

$$u_{max}(\pi') = \max\left\{ \hat{s}, \max_{1 \leq i < k} \left\{ \frac{s_i(\hat{S} - \hat{s})}{\rho S_{k-1}} \right\}, \max_{1 \leq i \leq m} \left\{ \frac{s_i(\hat{S} - \hat{s})}{\rho S_m} \right\} \right\} .$$

Since $s_1 \geq s_i$ for each $i = 1, 2, ..., m$, this gives

$$u_{max}(\pi') = \max\left\{ \hat{s}, \frac{s_1(\hat{S} - \hat{s})}{\rho S_{k-1}}, \frac{s_1(\hat{S} - \hat{s})}{\rho S_m} \right\} .$$

By our choice of ρ, the maximum of these three values is $\hat{s}$. This proves the first half the claim.

By construction,

$$\begin{aligned} U_{sum}(\pi') &= \hat{s} + \sum_{i=1}^{k-1} \frac{s_i(\hat{S} - \hat{s})}{\rho S_{k-1}} + (\rho - 1) \sum_{i=1}^{m} \frac{s_i(\hat{S} - \hat{s})}{\rho S_m} \\ &= \hat{s} + \frac{S_{k-1}(\hat{S} - \hat{s})}{\rho S_{k-1}} + (\rho - 1) \frac{S_m(\hat{S} - \hat{s})}{\rho S_m} \\ &= \hat{s} + \frac{\hat{S} - \hat{s}}{\rho} + (\rho - 1) \frac{\hat{S} - \hat{s}}{\rho} \\ &= \hat{s} \; . \end{aligned}$$

Therefore, the second half of the claim also holds.

Choose any positive numbers ϵ and $\delta_{h,i}$, where $1 \leq h \leq \rho$ and $1 \leq i \leq m$, such that

$$\begin{aligned}
\epsilon &< (\hat{S} - f(\hat{s}))\frac{s_1 - s_k}{\rho S_{k-1}(s_1 - \hat{s})}, \\
\delta_{h,i} &< \delta_{h,i+1} \text{ for } 1 \leq h \leq \rho \text{ and } 1 \leq i < m, \\
\delta_{h,m} &< \delta_{h+1,1} \text{ for } 1 \leq h < \rho, \text{ and} \\
\delta_{\rho,k-1} &< \epsilon \ .
\end{aligned}$$

Define $e_{h,i}$ and $d_{h,i}$ as follows

$$e_{h,i} \stackrel{\text{def}}{=} \begin{cases} \frac{s_i(\hat{S}-\hat{s})}{\rho S_m} & \text{if } 1 \leq h < \rho \text{ and } 1 \leq i \leq m \\ \frac{s_i(\hat{S}-\hat{s})}{\rho S_{k-1}} & \text{if } h = \rho \text{ and } 1 \leq i < k \\ \hat{s}(1+\epsilon) & \text{if } h = \rho \text{ and } i = k \end{cases}$$

and

$$d_{h,i} \stackrel{\text{def}}{=} \begin{cases} 1 + \delta_{h,i} & \text{if } (1 \leq h < \rho \text{ and } 1 \leq i \leq m) \text{ or if } (h = \rho \text{ and } 1 \leq i < k) \\ 1 + \epsilon & \text{if } h = \rho \text{ and } i = k \end{cases}$$

Let I be the real-time instance comprised of $((\rho - 1)m + k)$ jobs

$$I = \left\{J_{h,i} = (0, e_{h,i}, d_{h,i}) \mid 1 \leq h < \rho, 1 \leq i \leq m\right\} \bigcup \left\{J_{\rho,i} = (0, e_{h,i}, d_{h,i}) \mid 1 \leq i \leq k\right\} .$$

Note that if $\rho = 1$ then π' consists of k processors and I consists of k jobs.

To show that I is feasible on π' consider the schedule where $J_{\rho,k}$ is scheduled on the processor with speed $\hat{s}$ and the remaining jobs are scheduled on the processors with speed equal to their execution requirement. Thus, $J_{\rho,k}$ completes at $t = (1 + \epsilon)$ and the remaining jobs complete at $t = 1$.

Now consider the f-EDF schedule of I on π. The jobs in I are listed in increasing order of deadlines. Therefore, assuming $\rho > 1$, jobs $J_{1,1}, J_{1,2}, \ldots, J_{1,m}$ will be scheduled on processor with speeds $s_1, s_2, \ldots, s_m$, respectively. Furthermore, all these jobs will complete simultaneously at $t = (\hat{S} - \hat{s})/(\rho S_m)$, at which point the next m jobs will be scheduled. This will repeat $(\rho - 1)$ times with the final group of m jobs completing at $t = (\rho - 1)(\hat{S} - \hat{s})/(\rho S_m)$. Note if $\rho = 1$, this expression evaluates to 0 and will not affect the subsequent calculations.

Once the $(\rho - 1)m$ jobs with the earliest deadlines have executed, the final k jobs are scheduled in a similar manner — job $J_{\rho,i}$ will be scheduled on the processor with speed s_i for $1 \leq i \leq k$. Jobs $J_{\rho,1}, J_{\rho,2}, \ldots, J_{\rho,k-1}$ will all complete simultaneously after $(\hat{S} - \hat{s})/(\rho S_{k-1})$ time units at which point $J_{\rho,k}$ will migrate to the fastest processor. Assuming $J_{\rho,k}$ meets its deadline, we have the following inequality:

$$\frac{(\rho-1)(\hat{S}-\hat{s})}{\rho S_m}+\frac{(\hat{S}-\hat{s})}{\rho S_{k-1}}+\frac{\hat{s}(1+\epsilon)-\frac{\hat{S}-\hat{s}}{\rho S_{k-1}}s_k}{s_1}\leq 1+\epsilon\ .$$

Subtracting the leftmost term from both sides and multiplying by ρS_{k-1} gives

$$\hat{S}-\hat{s}+\frac{\rho S_{k-1}}{s_1}\hat{s}(1+\epsilon)-\frac{s_k}{s_1}(\hat{S}-\hat{s})\leq \rho S_{k-1}(1+\epsilon)-\frac{(\rho-1)S_{k-1}(\hat{S}-\hat{s})}{S_m}\ .$$

Combining the two $\hat{S}$ terms on the left-hand side of the inequality and subtracting $\frac{\rho S_{k-1}}{s_1}\hat{s}(1+\epsilon)-\hat{s}(1-\frac{s_k}{s_1})$ from both sides gives

$$\hat{S}\left(1-\frac{s_k}{s_1}\right)\leq \rho S_{k-1}(1+\epsilon)-\frac{\rho S_{k-1}}{s_1}\hat{s}(1+\epsilon)+\hat{s}\left(1-\frac{s_k}{s_1}\right)-\frac{(\rho-1)S_{k-1}(\hat{S}-\hat{s})}{S_m}\ .$$

Multiplying by $s_1/(s_1-s_k)$ gives

$$\hat{S}\leq \rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}+\hat{s}+\epsilon\rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}-\frac{(\rho-1)s_1 S_{k-1}(\hat{S}-\hat{s})}{S_m(s_1-s_k)}\ .$$

Noting that $\rho=1+(\rho-1)$ gives

$$\hat{S}\leq S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}+\hat{s}+\epsilon\rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}-(\rho-1)\left(\frac{s_1 S_{k-1}(\hat{S}-\hat{s})}{S_m(s_1-s_k)}+S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}\right)\ .$$

Combining terms gives

$$\hat{S}\leq S_{k-1}\frac{s_1}{s_1-s_k}+\hat{s}\left(1-\frac{S_{k-1}}{s_1-s_k}\right)+\epsilon\rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}-\frac{(\rho-1)S_{k-1}}{s_1-s_k}\left(s_1-\hat{s}+\frac{s_1(\hat{S}-\hat{s})}{S_m}\right)\ .$$

Noting that $S_{k-1}\frac{s_1}{s_1-s_k}=s_1+s_1\left(\frac{S_{k-1}}{s_1-s_k}-1\right)$ and that the last term of the above inequality cannot be negative, we have

$$\begin{aligned}\hat{S}\ &\leq\ s_1+s_1\left(\frac{S_{k-1}}{s_1-s_k}-1\right)+\hat{s}\left(1-\frac{S_{k-1}}{s_1-s_k}\right)+\epsilon\rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}\\ &=\ s_1+\frac{S_k-s_1}{s_1-s_k}(s_1-\hat{s})+\epsilon\rho S_{k-1}\frac{s_1-\hat{s}}{s_1-s_k}\ .\end{aligned}$$

The first two terms of the right-hand side of the final inequality equal $f(\hat{s})$. Therefore, we have

$$\left(\hat{S}-f(\hat{s})\right)\frac{s_1-s_k}{\rho S_{k-1}(s_1-\hat{s})}\leq\epsilon.$$

This contradicts our choice of ϵ. Therefore, $J_{\rho,k}$ cannot meet its deadline. ■

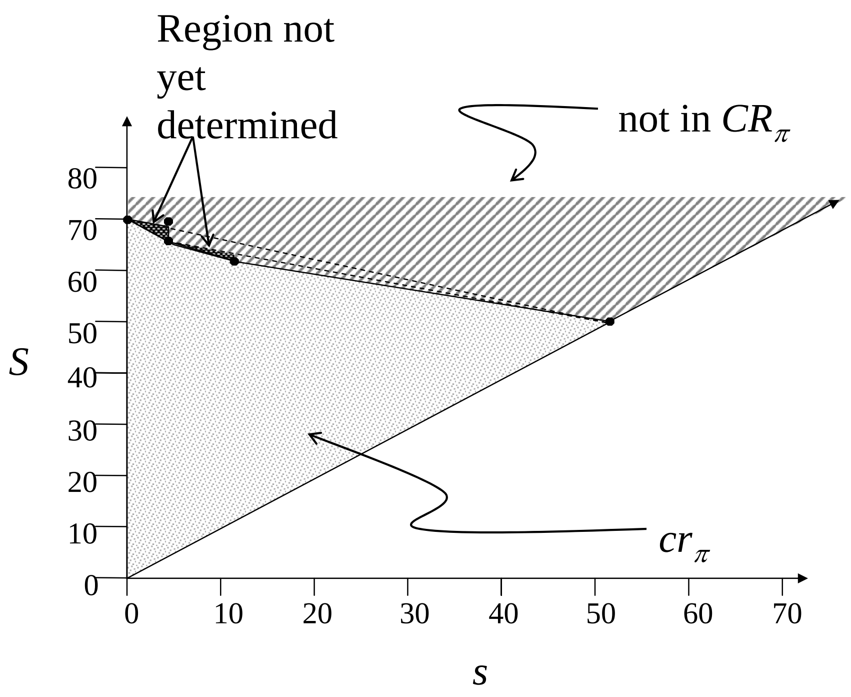


Figure 3.5: Points inside and outside of CR_π for $\pi = [50, 11, 4, 4]$.

Each line can be found in constant time and there are at most m lines, therefore the region that is definitely outside CR_π can also be found in $O(m)$ time.

3.5 Identifying points whose membership in CR_π has not been determined

We have seen that all points on or below the piecewise linear function $L_\pi(s)$ are in CR_π and that all points above any line segment connecting (s_1, s_1) to (s_k, S_k) are not in CR_π. Therefore, the remaining points are those that are above $L_\pi(s)$ and below every line segment between (s_1, s_1) and (s_k, S_k). For simplicity, we will call this region M. Figure 3.5 illustrates this region for the multiprocessor $\pi = [50, 11, 4, 4]$. We shall see that this region is always comprised of a group of triangles.

Use H_π to partition $0 < s \leq s_1$ and consider the points in M one partition at a time, beginning with $s_{h_2} \leq s \leq s_{h_1}$. By Theorem 17, we know that every point on or below the line segment between (s_{h_1}, S_{h_1}) and (s_{h_2}, S_{h_2}) is in CR_π. Furthermore, by Theorem 18, every point above this line segment is proved not to be in CR_π. Therefore, all points $(s, S) \in M$ have the property that $s < s_{h_2}$.

Now consider $s_{h_i} \leq s \leq s_{h_{i-1}}$ for some $3 \leq i \leq m'$. By Theorem 17, we know that every point below the line segment between $(s_{h_{i-1}}, S_{h_{i-1}})$ and (s_{h_i}, S_{h_i}) is in CR_π. Also, by Theorem 18, every point above the line segment between $(s_{h_{i-1}}, S_{h_{i-1}})$ and (s_1, s_1) is proved not to be in CR_π. Furthermore, this is the lowest segment that borders points not in CR_π because H_π is the lower edge of the convex hull. Therefore, if we let $g_k(s)$ be the line between (s_1, s_1) and (s_k, S_k), the region M is the set of points above the line segment between $(s_{h_{i-1}}, S_{h_{i-1}})$ and (s_{h_i}, S_{h_i}) and below or on $g_{h_i}(s)$ for $3 \leq i \leq m'$. For each i, the region between these two lines is the triangle defined by the three points $(s_{h_{i-1}}, S_{h_{i-1}})$, (s_{h_i}, S_{h_i}), and $(s_{h_{i-1}}, g_{h_i}(s_{h_{i-1}}))$.

Notice that M is empty when $|H_\pi| = 2$ and in this case $CR_\pi = cr_\pi$. Whenever $\lambda(\pi) = \sum_{i=2}^{m} s_i/s_1$, $CR_\pi = cr_\pi$ and hence CR_π is fully determined. For example, the characteristic region of $\pi = [2, 2, 2, 1]$ (with $\lambda(\pi) = 5/2$) is completely identified. Similarly, if π is an identical multiprocessor, then CR_π is completely identified.

3.6 Scheduling task sets on uniform heterogeneous multiprocessors using f-EDF

This section applies the theory developed in the previous sections to study the deadline-based scheduling of task sets on uniform heterogeneous multiprocessors. The results of the previous section can easily be applied to periodic and sporadic task sets to determine a sufficient condition for ensuring f-EDF-schedulability on any uniform heterogeneous multiprocessor π.

Theorem 19 *Let τ be any periodic or sporadic task set and let π be any uniform heterogeneous multiprocessor. If $(u_{max}(\tau), U_{sum}(\tau)) \in CR_\pi$, then τ is* f-EDF*-schedulable on π.*

Proof: Let π' be the n-processor uniform heterogeneous multiprocessor whose $i'th$ processor has speed u_i, the utilization of the i'th task of τ. Clearly, τ is feasible on π' — simply schedule all jobs generated by a given task on the processor whose speed matches that task's utilization. By construction, $s_1(\pi') = u_{max}$ and $S(\pi') = U_{sum}$. Thus, $(s_1(\pi'), S(\pi')) \in CR_\pi$ and by the definition of CR_π, τ is f-EDF-schedulable on π. ■

Computational complexity. Evaluating $(u_{max}(\tau), U_{sum}(\tau))$ takes $\mathcal{O}(n)$ time. Finding the segment of the piecewise function $L_\pi(s)$ that corresponds to u_{max} takes $\mathcal{O}(\log \cdot |H_\pi|) = \mathcal{O}(\log m)$ time. Therefore, evaluating $L_\pi(u_{max})$ takes $\mathcal{O}(\log m)$. Hence, if $L_\pi(s)$ has been calculated ahead of time the test takes $\mathcal{O}(n + \log m)$ time, otherwise it takes $\mathcal{O}(n + m)$ time.

3.7 Summary

This chapter has presented an f-EDF-schedulability test for any uniform heterogeneous multiprocessor π. This test was determined by considering feasibility on a less powerful platform π'. In many cases, feasibility is easier to determine than schedulability using a specific scheduling algorithm. Therefore, this chapter takes the difficult question of whether f-EDF can be used to schedule a real-time instance I on π and reduces it to the relatively easier question of whether *any* scheduling algorithm can be used to schedule I on a different (less powerful) platform π'.

In $O(m)$ time, one can divide all classes of uniform multiprocessors into three regions: those that are definitely in CR_π, those that are definitely not in CR_π, and those whose membership in CR_π is currently not determined. Moreover, in many cases the third region is empty.

Chapter 4

Partitioned EDF (p-EDF)

Section 2.2.2 in Chapter 2 described a polynomial-time approximation scheme (PTAS) developed by Hochbaum and Shmoys [HS86, HS87, HS88] for partitioning a task set τ onto a uniform heterogeneous multiprocessor π. Given any constant $\delta < 1$, this method attempts to schedule τ onto a uniform heterogeneous multiprocessor π' in which the speed of each processor of π' is $(1 + \delta)$ times as fast as the speed of the corresponding processor of π. If τ can be successfully partitioned onto π, this method will successfully partition τ onto π'. On the other hand, if this method cannot successfully partition τ onto π', then we can conclude that τ cannot be partitioned onto π.

This PTAS resolves the important theoretical question of whether an approximate solution to the problem of determining partitioned EDF-schedulability can be determined in polynomial time. However, it evaluates an expensive calculation for *each* task set τ. Instead, we would like to determine a test that tells us if *all* task sets with maximum utilization u_{max} and total utilization U_{sum} can be partitioned onto π. To this end, we take an alternative approach to partitioning task sets onto uniform heterogeneous multiprocessors. The approach in this dissertation sorts the tasks by utilization prior to assigning them to processors and uses uniprocessor EDF to schedule the individual processors.

First-fit- and any-fit-decreasing task assignment. This dissertation considers two task assignment heuristics — Any Fit Decreasing (AFD) and First Fit Decreasing (FFD). Both heuristics sort the tasks and assign them to processors in decreasing order of utilization. A task can be assigned to processor s_i if its utilization is no greater than $(s_i - U_i)$, where U_i is the total utilization of tasks already assigned to the processor — in this case, we say the task *fits* on the processor. Once the tasks are assigned to the processors, they are scheduled using uniprocessor EDF. These scheduling algorithms are called AFD-EDF and FFD-EDF. Figure 4.1 shows the FFD-EDF task assignment algorithm in which each task is assigned to the fastest processor upon which it will fit. The variable $\mathsf{gap}(j)$ denotes the amount of remaining capacity available on s_i. According to the FFD-EDF task assignment algorithm, each task T_i is assigned to processor s_j, where j is the smallest-indexed processor with $\mathsf{gap}(j)$

```
FFD-EDF task assignment $(\tau, \pi)$

Let $\tau = \{T_1, T_2, \ldots, T_n\}$ denote the tasks, with $e_i/p_i \geq e_{i+1}/p_{i+1}$ for all $i$
Let $\pi = \{s_1, s_2, \ldots, s_m\}$ denote the processors, with $s_j \geq s_{j+1}$ for all $j$

for $j \leftarrow 1$ to $m$ do $\mathsf{gap}(j) := s_j$

for $i \leftarrow 1$ to $n$ do
    Let $j_o$ denote the smallest index such that $\mathsf{gap}(j_o) \geq e_i/p_i$
    If no such $j_o$ exists declare $\tau$ FFD-EDF-infeasible on $\pi$; return
    Assign $T_i$ to processor $j_o$
    $\mathsf{gap}(j_o) := \mathsf{gap}(j_o) - e_i/p_i$
od
```

Figure 4.1: The FFD-EDF task-assignment algorithm.

at least as large as u_i, the utilization of T_i. If FFD-EDF attempts to assign task T_i to a processor and all the processors' gaps are smaller than u_i, then τ is said to be FFD-EDF-*infeasible* on π. Otherwise, τ is said to be FFD-EDF-*feasible* on π. The only difference between the AFD-EDF and FFD-EDF algorithms is that AFD-EDF assigns each task to *any* processor upon which it will fit, while FFD-EDF must assign the task to the fastest processor. The run-time computational complexity of FFD-EDF task assignment is $\mathcal{O}(n \log n)$ (for sorting the tasks in non-increasing order of utilizations) + $\mathcal{O}(m \log m)$ (for sorting the processors in non-increasing order of capacities) + $\mathcal{O}(n \times m)$ (for doing the actual assignment of tasks to processors), for an overall computational complexity of $\mathcal{O}(n \cdot (\log n + m))$ assuming the number of processors does not exceed the number of tasks.

These task partitioning algorithms are a scheduling application of the bin packing algorithms FFD and AFD. While both AFD-EDF and FFD-EDF have been studied in detail for identical bins, less attention has been paid to these algorithms for the variable bin packing problem. These are known to be superior bin-packing algorithms for identical bins (e.g., they have the best known competitive ratio for identical bin sizes [Joh73]).

4.1 The utilization bound for FFD-EDF and AFD-EDF

In this section, we will estimate the utilization bound for partitioned EDF scheduling on uniform heterogeneous multiprocessors. More specifically, given any uniform heterogeneous multiprocessor π and any value u, we will find a value U such that every task set τ with $u_{max}(\tau) \leq u$ and $U_{sum}(\tau) \leq U$ can be partitioned onto π. Of course the bound will vary depending on what method we use to assign tasks to processors. The expression $\mathcal{U}_{\pi}^{A}(u)$ denotes the bound associated with task assignment algorithm A:

$$\mathcal{U}_\pi^A(u) \stackrel{\text{def}}{=} \max\{U \mid \text{ every } \tau \text{ with } U_{sum}(\tau) \leq U \wedge u_{max}(\tau) \leq u \text{ is A-schedulable on } \pi\}.$$

The notation $\mathcal{U}_\pi^A$ refers to the utilization bound over the full range of maximum utilization values — i.e., this refers to the entire function. We will consider the algorithms FFD-EDF and AFD-EDF — the p-EDF scheduling algorithm using the partitioning heuristics FFD and AFD, respectively. In the remainder of this section, we will determine a method of approximating $\mathcal{U}_\pi^{\textsf{AFD-EDF}}$. Since FFD-EDF is a special case of AFD-EDF, $\mathcal{U}_\pi^{\textsf{FFD-EDF}}(u) \geq \mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$ for all u. Therefore, the approximation we find in this chapter will also provide a utilization bound for FFD-EDF, but it is not guaranteed to be an approximation of $\mathcal{U}_\pi^{\textsf{FFD-EDF}}$.

The approximation algorithm presented in this chapter is not very efficient — it takes time exponential in the number of processors in π. However, it needs to be run only once for each uniform heterogeneous multiprocessor π. The utilization bound can be provided with any uniform heterogeneous multiprocessor along with other system parameters such as the individual processor speeds, interprocessor migration costs, and voltage information. Once the bound is determined, performing a sufficient schedulability test for task set τ reduces to the linear-time computation of $U_{sum}(\tau)$ and $u_{max}(\tau)$, followed by table look-up to determine whether $U_{sum}(\tau) \leq \mathcal{U}_\pi^{\textsf{AFD-EDF}}(u_{max}(\tau))$. Figure 4.9 on page 88 illustrates an approximate utilization bound for the uniform multiprocessor $\pi = [2.5, 2, 1.5, 1]$. Given any task set τ, if $(u_{max}(\tau), U(\tau))$ is below the line shown in the graph, then τ is AFD-EDF-feasible on π. This example is discussed in more detail in Section 4.2

The remainder of this chapter is organized as follows. We begin by exploring properties of task sets that are AFD-EDF-schedulable. Based on insights gained from this exploration, we modify the problem so as to greatly reduce the complexity of the problem. Finally, we present an approximation algorithm for the utilization bound of the modified problem.

Consider the AFD-EDF task-assignment of any task set τ that is AFD-EDF-*in*feasible on π. We observe the following about the AFD-EDF task-assignment of τ prior to attempting to assign the first unschedulable task to a processor.

Observation 1 *Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be any task set that is* AFD-EDF*-infeasible on π. Assume the tasks of τ are indexed so that $u_i \geq u_{i+1}$ for $1 \leq i < n$ and let T_k be the first task of τ that cannot fit on any processor. Thus, there is some feasible task assignment of $\tau' = \{T_1, T_2, \ldots, T_{k-1}\}$ on π. Let* gap(j) *denote the gap on the j'th processor after this feasible* AFD-EDF *task-assignment of τ'. Then*

$$u_i > \max\{\textsf{gap}(j) \mid 1 \leq j \leq m\} \textit{ for all } i \textit{ such that } 1 \leq i \leq k\ . \tag{4.1}$$

Proof: Since T_k cannot fit on any of the processors, u_k must be strictly greater than the maximum gap. By assumption, $u_k \leq u_i$ for all $i < k$. Therefore, the condition holds. ■

This important property is used to develop our approximation algorithm. Based on this observation, we convert any FFD-EDF-infeasible task set τ to another task set τ' with a very specific structure — this structure can be exploited to approximate $\mathcal{U}_{\pi}^{\mathsf{AFD\text{-}EDF}}$. We call these highly structured task sets *modular on* π.

Definition 11 (modular on π) *Let $\pi = [s_1, \ldots, s_m]$ denote an m-processor uniform heterogeneous multiprocessor and let $\tau = \{T_1, T_2, \ldots, T_n\}$ denote a task set. Then τ is said to be* modular on π *if the tasks of τ have at most $(m+1)$ distinct utilizations $v_1, v_2, \ldots, v_{m+1}$ and there exists a partitioning of the tasks in τ among the m processors of π that satisfies the following properties.*

1. *For all $i, j < n$, if T_i and T_j are both assigned to the k'th processor then $u_i = u_j$ — we denote this utilization v_k,*
2. *for all $k \leq m$, there are $\lfloor s_k / v_k \rfloor$ tasks of utilization v_k assigned to the k'th processor, and*
3. *T_n fits on the processor with the largest gap — i.e., $u_n = v_{m+1} = \max_{1 \leq i \leq m}\{s_i \bmod v_i\}$.*

The m-tuple $(v_1, v_2, \ldots, v_m)$ represents this task set. The utilization of task T_n is implicitly included in this m-tuple since it is a function of the $v_1, v_2, \ldots, v_m$.

Prior to assigning task T_n to a processor, the gap on the i'th processor is $(s_i \bmod v_i)$ for each $i = 1, 2, \ldots, m$. Thus, $S(\pi) - \sum_{i=1}^{m}(s_i \bmod v_i)$ is the total utilization of the tasks in $\tau \backslash \{T_n\}$. By the third property in Definition 11 above, the utilization of T_n is $\max\{s_i \bmod v_i \mid 1 \leq i \leq m\}$. Therefore, the utilization of the modular task set $(v_1, v_2, \ldots, v_m)$, is given by the following equation:

$$\mathtt{ModUtil}(v_1, v_2, \ldots, v_m) = S(\pi) - \sum_{i=1}^{m}(s_i \bmod v_i) + \max_{1 \leq i \leq m}\{s_i \bmod v_i\}. \tag{4.2}$$

Example 4.1 *Let $\pi = [7, 6, 3]$ and let τ be a task set with $u_1 = 4$, $u_2 = u_3 = 3$, and $u_4 = u_5 = 2$. Then τ is modular on π. Figure 4.2 illustrates an* AFD-EDF *partitioning of τ onto π that satisfies the condition of Definition 11. Throughout this chapter, illustrations of partitioned assignments reflect the correspondence between the partitioning problem and the bin packing problem — processors are represented as bins with their size reflecting the processor speed, and tasks are represented as items with their size reflecting their utilization. Unused processor capacity is drawn hatched.*

The total utilization of τ is $4 + 2 \cdot 3 + 2 \cdot 2 = 14$, which satisfies Equation (4.2),

$$14 = 16 - ((7 \bmod 3) + (6 \bmod 4) + (3 \bmod 2)) + \max\{(7 \bmod 3), (6 \bmod 4), (3 \bmod 2)\}.$$

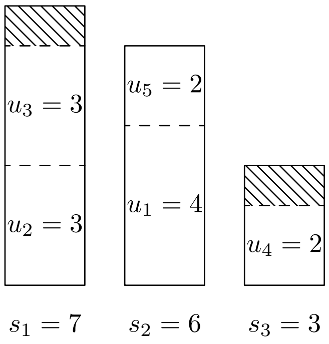


Figure 4.2: A modular task set.

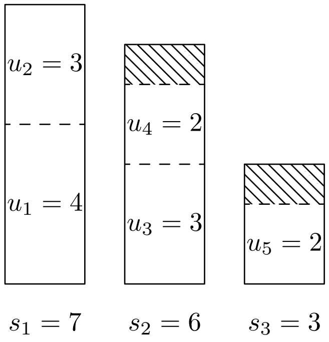


Figure 4.3: FFD-EDF may not generate a modular schedule.

While the modular partition can be produced by an AFD-EDF assignment, the FFD-EDF assignment may not satisfy the conditions of Definition 11. For example, Figure 4.3 illustrates the FFD-EDF task assignment of the system discussed in Example 4.1. In this example, both s_1 and s_2 have tasks assigned to them with different utilizations. Furthermore, both of the tasks with utilization 2 are assigned to processors that have slack. In a modular task assignment, there is no slack on the processor to which the final task is assigned. Nevertheless, modular task sets are useful for finding the approximation of $\mathcal{U}_\pi^{\mathsf{AFD\text{-}EDF}}$ — for small m, the modular tasks sets can be exhaustively searched to approximate a utilization bound. Below we will see that all AFD-EDF-infeasible tasks sets have a corresponding AFD-EDF-feasible task set that is modular on π.

Notice that Property 3 of the modular task set definition bears a close resemblance to the observation about infeasible task sets (Equation (4.1)). Taking note of this similarity, we can find a modular task set τ' that corresponds to any AFD-EDF-infeasible task set τ by first reducing the utilization of τ appropriately until the task set is "just about feasible," and then "averaging" the feasible task set to make it modular on π. Both these steps are described in

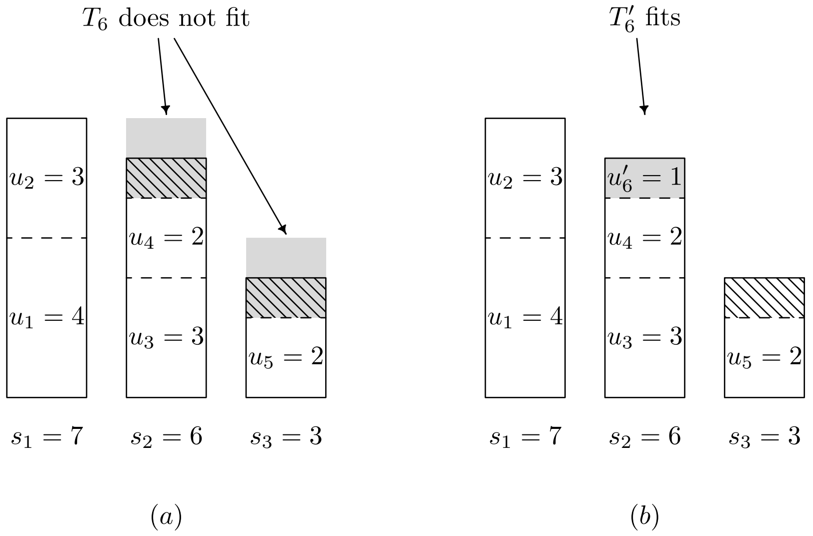


Figure 4.4: A feasible reduction.

more detail below.

Definition 12 (feasible reduction) *Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be* AFD-EDF*-infeasible on π. Assume the tasks of τ are indexed so that $u_i \geq u_{i+1}$ for all $1 \leq i < n$ and let T_k be the task of τ upon which the* AFD-EDF *task-assignment algorithm reports failure. Let* $\mathsf{gap}(j)$ *denote the gap on the j'th processor after performing the* AFD-EDF *task-partitioning algorithm on the first $(k-1)$ tasks of τ and let $g \overset{def}{=} \max\{\mathsf{gap}(j) \mid 1 \leq j \leq m\}$. Then the* AFD-EDF***-feasible reduction*** *of τ, denoted τ', is defined as follows:*

- *If $g = 0$, $\tau' = \{T_1, T_2, \ldots, T_{k-1}\}$,*
- *otherwise $\tau' = \{T_1, T_2, \ldots, T_{k-1}, T_g\}$, where T_g is any task with utilization equal to g.*

Thus, in a feasible reduction, tasks $T_1, T_2, \ldots, T_{k-1}$ are unchanged, tasks $T_{k+1}, T_{k+2}, \ldots, T_n$ are removed from the task set, and the utilization of T_k is reduced to fit into the largest gap, if any gap exists, or removed altogether if no processors have any gap.

The following example illustrates an AFD-EDF-infeasible task set and its feasible reduction.

Example 4.2 *Let $\pi = [7, 6, 3]$ and let τ be a task set with $u_1 = 4$, $u_2 = u_3 = 3$, and $u_4 = u_5 = u_6 = 2$. Then τ is* AFD-EDF*-infeasible on π. Inset (a) of Figure 4.4 illustrates the* AFD-EDF *partitioning of τ onto π and inset (b) illustrates its feasible reduction. When* AFD-EDF *attempts to assign T_6, there is no processor with a large enough gap. In the feasible reduction, the utilization of T_6' is 1, the maximum gap when* AFD-EDF *failed to assign T_6.*

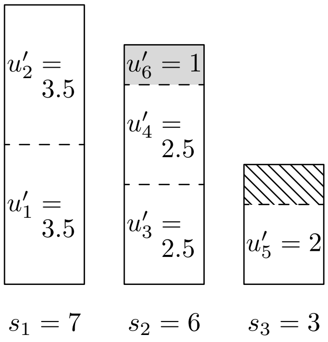


Figure 4.5: A modularized feasible reduction.

The AFD-EDF-feasible reduction of τ relates Equation (4.1), our observation about infeasible task sets, to Property 3 of modular task sets. However, the AFD-EDF-feasible reduction of τ may not be a modular task set since tasks on the same processor do not necessarily have the same utilization. In such cases, we will "modularize" the task set as follows.

Definition 13 (modularization of τ) *Let $\pi = [s_1, s_2, \ldots, s_m]$ be a uniform heterogeneous multiprocessor and let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a feasible reduction of some* AFD-EDF*-infeasible task set on π with $u_{i+1} \leq u_i$ for all $i = 1, 2, ..., n-1$. Let* A *be an assignment of τ onto π in which the largest gap remaining on any processor is u_n after assigning $T_1, T_2, \ldots, T_{n-1}$ — i.e., T_n exactly fits onto the processor to which it is assigned. Define τ_i as follows:*

$$\tau_i \overset{def}{=} \begin{cases} \{T_j \mid 1 \leq j \leq n \wedge \mathsf{A} \textit{ assigns } T_j \textit{ to } s_i\} & \textit{if } U_{sum}(\tau) = S(\pi) \\ \{T_j \mid 1 \leq j < n \wedge \mathsf{A} \textit{ assigns } T_j \textit{ to } s_i\} & \textit{if } U_{sum}(\tau) < S(\pi) \end{cases}$$

Let $v_i \overset{def}{=} U_{sum}(\tau_i)/|\tau_i|$. Thus, v_i is the average utilization of the tasks of τ (if $U_{sum}(\tau) = S(\pi)$) or of $\tau \setminus \{T_n\}$ (if $U_{sum}(\tau) < S(\pi)$) that are assigned to the i'th processor of π. Then the modularization of τ is represented by the m-tuple $(v_1, v_2, \ldots, v_m)$.

Notice that if $U_{sum}(\tau) = S(\pi)$ then all the tasks of τ assigned to each processor will have the same utilization. On the other hand, if $U_{sum}(\tau) < S(\pi)$ then all the tasks of $\tau \setminus \{T_n\}$ assigned to each processor will have the same utilization and T_n's utilization will be exactly equal to the largest gap. Figure 4.5 illustrates the "modularization" of the task set illustrated in inset (*b*) of Figure 4.4.

The following lemma states that modularizing any AFD-EDF-feasible reduction in this way will not change the total utilization, nor will it increase the maximum utilization. In this lemma, we restrict our attention to those task sets whose total utilization is strictly less than $S(\pi)$. A feasible reduction can have total utilization equal to $S(\pi)$ if AFD-EDF is able to assign tasks to π until all the processors have no gap. In this case, T_k in Definition 12 is removed from the task set and the feasible reduction is $\{T_1, T_2, \ldots, T_{k-1}\}$. This type of feasible reduction occurs only when $g = 0$. Since we know that no partitioning algorithm is optimal on multiprocessors, it must be the case that $\mathcal{U}_\pi^{\textsf{AFD-EDF}} < S(\pi)$. Therefore, we can ignore any AFD-EDF-feasible reduction whose utilization equals $S(\pi)$ without affecting the accuracy of our analysis.

Lemma 5 *Let $\pi = [s_1, s_2, \ldots, s_m]$ be a uniform heterogeneous multiprocessor and let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a feasible reduction of some* AFD-EDF*-infeasible task set on π with $u_{i+1} \leq u_i$ for all $i = 1, 2, ..., n-1$ and $U_{sum}(\tau) < S(\pi)$. Let $\tau' = (v_1, v_2, \ldots, v_m)$ be the modularization of τ as described in Definition 13. Then the following three conditions hold:*

- $u_n = \max\{s_i \bmod v_i | 1 \leq i \leq m\}$,
- $U_{sum}(\tau') = U_{sum}(\tau)$, *and*
- $u_{max}(\tau') \leq u_{max}(\tau)$.

Proof: By the definition of feasible reduction, if there is a gap on any processor when AFD-EDF reports failure while trying to assign task T_k to a processor, the utilization of T_k is reduced so it will fit onto the processor with the largest gap. Therefore, task T_n was found by reducing the utilization of some task and u_n is exactly equal to the largest gap when T_n is assigned to a processor. This implies u_n is *strictly* smaller than the utilization of any other task in τ. Moreover, since T_n fits exactly into the maximum gap, it follows that $u_k > \max\{gap(j) \mid 1 \leq j \leq m\}$ for every $k < n$.

We now show that the number of tasks on each processor is not changed by modularizing τ. Consider the tasks of $\tau \setminus \{T_n\}$ assigned to processor s_i. Recall these tasks are denoted τ_i. In τ', these tasks are replaced by $\lfloor s_i/v_i \rfloor$ tasks with utilization $v_i = U_{sum}(\tau_i)/|\tau_i|$. Therefore, we need to demonstrate that $\lfloor s_i/v_i \rfloor = |\tau_i|$.

Let g_i be the gap on the i'th processor prior to assigning task T_n to a processor — i.e., $g_i \stackrel{\text{def}}{=} s_i - U_{sum}(\tau_i)$. Therefore, $g_i < u_j$ for each $T_j \in \tau_i$. Since v_i is the average utilization of the tasks in τ_i, we have $g_i < v_i$. Furthermore,

$$s_i - g_i = U_{sum}(\tau_i) = \frac{U_{sum}(\tau_i)}{|\tau_i|} \cdot |\tau_i| = v_i \cdot |\tau_i| \ .$$

Therefore,

$$v_i(|\tau_i| + 1) = s_i - g_i + v_i > s_i \geq v_i|\tau_i| \ ,$$

so $|\tau_i| = \lfloor s_i/v_i \rfloor$ and $U_{sum}(\tau_i) = v_i \cdot |\tau_i|$.

By the above argument, $U_{sum}(\tau_i) = s_i - (s_i \bmod v_i)$ for each $i = 1, 2, \ldots, m$, so the gap on each processor is unchanged by the process of modularization. Therefore, both the first and the second conditions must hold.

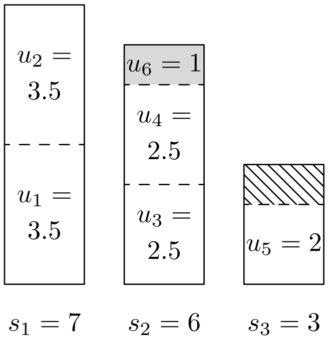


Figure 4.6: A modularized system.

It remains to demonstrate the third condition holds — i.e., $u_{max}(\tau) \geq u_{max}(\tau') = \max\{v_i \mid 1 \leq i \leq m\}$. This must hold since the utilizations of τ' were found by averaging utilization values of τ and the result of an averaging operation will always fall between the minimum and maximum values averaged. ■

Therefore, any feasible reduction on π can be "modularized." Furthermore, the process of modularizing a feasible reduction τ will not change the total utilization and will not increase the maximum utilization. Figure 4.6 illustrates the "modularization" of the system discussed in Example 4.2.

Theorem 20 states these modular task sets can be used to find $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$.

Theorem 20 *Let $\pi = [s_1, s_2, \ldots, s_m]$ be any uniform heterogeneous multiprocessor and let*

$$\mathcal{M}_\pi(u) \overset{def}{=} \min\{U(\tau) \mid \tau \textit{ is modular on } \pi \textit{ and } u_{max}(\tau) \leq u\}. \tag{4.3}$$

Then $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u) = \mathcal{M}_\pi(u)$ for all $u \in (0, s_1]$.

Proof: By Lemma 5 every AFD-EDF-feasible reduction on π has a corresponding modular task set with the same total utilization and with the same or smaller maximum utilization. Furthermore, every modular task set is an AFD-EDF-feasible reduction of some infeasible task set — increasing the utilization of the smallest task set slightly will cause the modular task set to become AFD-EDF-infeasible. Therefore, there can be no AFD-EDF-infeasible task set τ with $u_{max}(\tau) \leq u$ and $U_{sum}(\tau) < \mathcal{M}_\pi(u)$. The result follows. ■

This section has demonstrated that we can find $\mathcal{U}_\pi^{\textsf{AFD-EDF}}$ by considering only modular task sets. The following section describes how to accurately estimate $\mathcal{M}_\pi$ (thereby estimating $\mathcal{U}_\pi^{\textsf{AFD-EDF}}$) by considering discrete utilization values.

4.2 Estimating the utilization bound

Because modular task sets have a limited number of distinct utilizations and they have such a rigid structure, for small values of m we can approximate $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$ by doing a thorough search of valid m-tuples $(u_1, u_2, \ldots, u_m)$ with $u_i \in (0, s_i]$. Since this is a continuous region, we cannot search *every* possibility within this range. Instead, we allow the utilizations to be chosen from a finite set V. Figure 4.7 illustrates the approximation algorithm that considers all the valid utilization combinations in the set V^m (the set of m-tuples of elements in V). The function $\texttt{MinUtilGraph}(\pi, V)$ finds the graph of $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$ for $u \in (0, s_1]$. It calls the function $\texttt{MinUtilLeqA}(\pi, V, u)$, which returns the value of $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$ for a specific value of u by considering all valid m-tuples in the subset V bounded by u. $\texttt{MinUtilLeqA}(\pi, V, u)$ chooses a utilization for processor s_1 and calls $\texttt{ConsiderAnotherProc}(\pi, max_{gap}, max_{sum}, min_{util})$, which recursively chooses utilizations between max_{gap} and u_{max} for the remaining processors.

Of course, the values in V must be chosen carefully to ensure the V contains the fewest possible number of elements that will approximate $\mathcal{U}_\pi^{\textsf{AFD-EDF}}$ within a reasonable margin of the actual bound. In particular, given any $\epsilon > 0$ we want to find V_ϵ so that for all $u \in (0, s_1]$, the difference between $\texttt{MinUtilLeqA}(\pi, V_\epsilon, u)$ and $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u)$ is at most ϵ.

Assume that OPT$= (u_1, u_2, \ldots, u_m)$ is a modular set whose total utilization equals the bound: $U_{sum}(\text{OPT}) = \mathcal{U}_\pi^{\textsf{AFD-EDF}}(u_{max}(\text{OPT}))$. We wish to choose V_ϵ so as to ensure $\texttt{MinUtilLeqA}$ will consider some m-tuple $(u'_1, u'_2, \ldots, u'_m)$ with maximum utilization no more than u_{max} and $|\texttt{ModUtil}(u'_i, u'_2, \ldots, u'_m) - \texttt{ModUtil}(\text{OPT})| \leq \epsilon$. The remainder of this section first describes a condition that guarantees that there is an m-tuple $(u'_i, u'_2, \ldots, u'_m) \in V_\epsilon$ with $|\texttt{ModUtil}(u'_i, u'_2, \ldots, u'_m) - \texttt{ModUtil}(\text{OPT})| \leq \epsilon$ and then finds a set V_ϵ satisfying the condition. The condition states that for every two consecutive elements v_k and v_{k+1} of V_ϵ , if $v_k < v \leq v_{k+1}$ then $(s_i \bmod v)$ must be within $\epsilon/(m-1)$ of $(s_i \bmod v_k)$ for each processor s_i.

Lemma 6 *Consider any uniform heterogeneous multiprocessor $\pi = [s_1, s_2, \ldots, s_m]$ and any $\epsilon > 0$. Let $(u_1, u_2, \ldots, u_m)$ represent a modular task set τ with $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u_{max}(\tau)) = U_{sum}(\tau)$. Choose a set $V_\epsilon = \{v_1, v_2, \ldots, v_r\}$ with $v_i < v_{i+1}$ for all $i < r$ and $v_r = s_1$. If the minimum utilization of τ at least $\epsilon/(m-1)$ and the following two conditions hold*

$$v_1 \leq \frac{\epsilon}{m-1}, \text{ and} \tag{4.4}$$

$$v - v_i \leq \frac{\epsilon}{m-1} \text{ for all } v \in (v_{i-1} - v_i] \text{ where } i = 2, 3, \ldots, r, \tag{4.5}$$

then there exists a modular task set represented by $(u'_1, u'_2, \ldots, u'_m) \in V_\epsilon^m$ with maximum utilization at most u_{max} such that

$$\texttt{ModUtil}(u'_1, u'_2, \ldots, u'_m) - \texttt{ModUtil}(u_1, u_2, \ldots, u_m) \leq \epsilon.$$

```
function MinUtilGraph($\pi, V$)
    % $\pi = [s_1, s_2, ..., s_m]$ is the uniform multiprocessor
    % $V = \{v_1, v_2, ..., v_r\}$ is the set of allowable utilizations in decreasing order
    $\mathcal{U} = S(\pi)$
    while $u_{max} > v_r$
        % $min\_util$ is the minimum total utilization
        % $min\_a$ is the minimum of the set $\{u_{max}(\tau) \mid U(\tau) = \mathcal{U}$ and $u_{max}(\tau) \leq u_{max}\}$
        $(min\_util, min\_u)$ =MinUtilLeqA($\pi, V, u_{max}$)
        if $min\_util > \mathcal{U}$
            draw a line from $(u_{max}, \mathcal{U})$ to $(min\_u, \mathcal{U})$
            $u_{max} = min\_u; \mathcal{U} = min\_util$

function MinUtilLeqA($\pi, V, u_{max}$)
    % uses a recursive function to find the minimum feasible utilization
    $min\_util = S(\pi); min\_u = s_1$ % global variables for the recursion
    for $((u_1$ in $V)$ and $(u_1 \leq u_{max}))$
        % call ConsiderNextProc recursively
        ConsiderNextProc($2, (s_1 \bmod u_1), (s_1 \bmod u_1), u_1$)
return $(min\_util, min\_u)$

function ConsiderNextProc($proc, , max\_gap, gap\_sum, min\_util$)
    % there are two exit conditions
    %  — $proc = m + 1$ or none of the current tasks fit on $proc$
    if $((proc = m+1)$ or $((s_{proc} < min\_util)$ and $((s_{proc} < max\_gap)$ or $(max\_gap = 0))))$
        $util = \sum_{i=1}^{proc-1} s_i - gap\_sum + max\_gap$
        $a = \max\{s_i \mid 1 \leq i < proc\}$
        if $((util < min\_util)$ or $((util = min\_util)$ and $(u < min\_u)))$
            $min\_util = util; min\_u = u$
        return
      % the exit conditions are not yet, consider next processor $(proc + 1)$
      for $((u_{proc}$ in $V)$ and $(u_{proc} \leq \max\{u_{max}, s_{proc}\}))$
          ConsiderNextProc($proc + 1, max\{max\_gap, (s_{proc} \bmod u_{proc})\}$,
                          $(gap\_sum + (s_{proc} \bmod u_{proc})), \min\{min\_util, u_{proc}\}$)
```

Figure 4.7: Approximating the minimum utilization bound of modular task sets.

Proof: Let $u'_i = \max\{v \mid v \in V_\epsilon \wedge v \leq u_i\}$. Since $u_i \geq \frac{\epsilon}{m-1} \geq v_1$, u'_i exists for each $i = 1, 2, \ldots, m$. Assume s_j is the processor with the maximum gap prior to assigning task T_n — i.e., $(s_j \bmod u_j) \geq (s_i \bmod u_i)$ for each $i = 1, 2, \ldots, m$. Then

$$\texttt{ModUtil}(u_1, u_2, \ldots, u_m) = S(\pi) - \sum_{i=1}^{m}(s_i \bmod u_i) + (s_j \bmod u_j) = S(\pi) - \sum_{i \neq j}(s_i \bmod u_i).$$

And

$$\begin{aligned}\texttt{ModUtil}(u'_1, u'_2, \ldots, u'_m) &= S(\pi) - \sum_{i=1}^{m}(s_i \bmod u'_i) + \max_{1 \leq i \leq m}(s_i \bmod u'_i) \\ &\leq S(\pi) - \sum_{i \neq j}(s_i \bmod u'_i).\end{aligned}$$

Therefore,

$$\begin{aligned}&\texttt{ModUtil}(u'_1, u'_2, \ldots, u'_m) - \texttt{ModUtil}(u_1, u_2, \ldots, u_m) \\ &\leq \sum_{i \neq j}[(s_i \bmod u'_i) - (s_i \bmod u_i)] \\ &\leq (m-1)\frac{\epsilon}{m-1} \qquad (\textit{by Condition } (4.5) \textit{ above}) \\ &= \epsilon.\end{aligned}$$

■

Therefore, we need to construct V_ϵ so that Conditions (4.4) and (4.5) hold. In order to determine the values in V_ϵ, we need to consider the function $(s_i \bmod x)$ more carefully. Figure 4.8 illustrates the graph of $y = (6 \bmod x)$. Consider the value of y as x increases. If x does not divide 6, y decreases as x increases until x divides 6, at which point $(6 \bmod x) = 0$. Once x increases beyond this point, the function suffers a discontinuity — the value increases to the line $y = x$ and then begins decreasing once again. Thus, this graph is a series of straight lines between the points $(6/(k+1), 6/(k+1))$ and $(6/k, 0)$ — each with slope $(-k)$. It is easy to see that the graph of $(s_i \bmod x)$ follows this general pattern for every value s_i.

Using this understanding of the function $(s_i \bmod x)$, we will construct a set V_ϵ satisfying Conditions (4.4) and (4.5) above.

Since the slope of $(s_i \bmod x)$ is $(-k)$ when $s_i/(k+1) \leq x \leq s_i/k$ and we want to ensure Condition (4.5) holds, the elements of V_ϵ must be at most $\epsilon/(k \cdot (m-1))$ units apart. This value is minimized when k is maximized — i.e., when considering the function $(s_1 \bmod x)$. Therefore, the following must hold for V_ϵ

$$v_{i+1} - v_i \leq \frac{\epsilon}{k(m-1)} = \frac{\epsilon}{\lceil s_1/v_i \rceil (m-1)}\ .$$

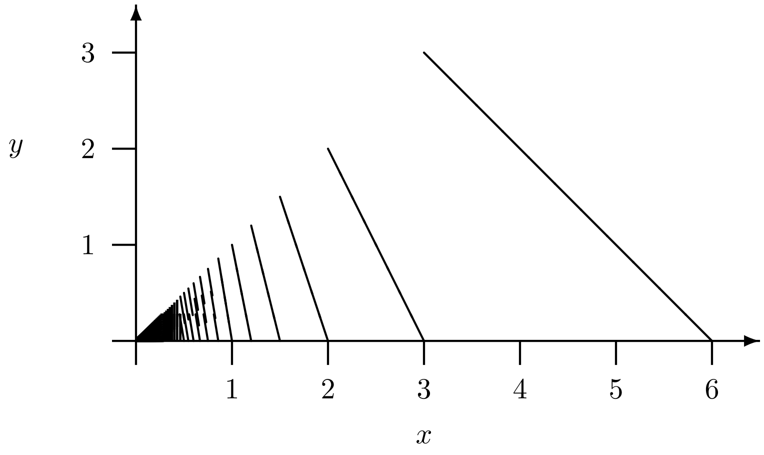


Figure 4.8: The graph of $y = 6 \bmod x$

In addition, V_ϵ must contain all the points where ($s_i \bmod x$) jumps from the line $y = x$ to the x-axis for some processor s_i. If these points were not included, Condition (4.5) would be violated.

Given this, we will we construct V_ϵ as follows:

$$V_\epsilon \stackrel{\text{def}}{=} \left\{ \frac{s_i}{k} \middle| 1 \le i \le m \quad \wedge \quad 1 \le k \le \left\lfloor \frac{s_i(m-1)}{\epsilon} \right\rfloor \right\} \bigcup$$
$$\left\{ \frac{s_1}{k} - j \cdot \frac{\epsilon}{k(m-1)} \middle| 1 \le k \le \left\lfloor \frac{s_1(m-1)}{\epsilon} \right\rfloor \quad \wedge \quad 1 \le j < \left\lfloor \frac{s_1(m-1)}{\epsilon(k+1)} \right\rfloor \right\}. \tag{4.6}$$

With this construction of V_ϵ, we can ensure our estimate of $\mathcal{U}_\pi^{\textsf{AFD-EDF}}$ is sufficiently close to the actual bound.

Theorem 21 *Let $\pi = (s_1, s_2, \ldots, s_m)$ be any uniform multiprocessor. Then for any ϵ, there exists a set V_ϵ such that* $\texttt{MinUtilLeqA}(\pi, V_\epsilon, u) \le \mathcal{U}_\pi^{\textsf{AFD-EDF}}(u) + \epsilon$ *for all $u > 0$. Moreover,*

$$|V_\epsilon| \le \frac{m-1}{\epsilon} \left[S(\pi) + s_1 \cdot \left(H_{(s_1(m-1)/\epsilon)+1} - 1 \right) \right],$$

where H_i is the i'th harmonic number — $H_i \stackrel{def}{=} \sum_{k=1}^{i} (i/k)$.

Proof: Equation (4.6) defines a set V_ϵ that satisfies the conditions of the theorem. This contains **(i)** all points of the form (s_i/j) between $\epsilon/(m-1)$ and s_1, and **(ii)** all points that are $\epsilon/(k(m-1))$ distance apart in the same range, where $(-k)$ is the slope of the function $(s_1 \bmod u)$.

Consider the following two cases.

Case 1: $\mathcal{U}_\pi^{\textsf{AFD-EDF}}(u) > S(\pi) - \epsilon$.

In this case $\mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u) + \epsilon > S(\pi)$. The value of $\texttt{MinUtilLeqA}(\pi, V_\epsilon, u)$ never exceeds $S(\pi)$ since `MinUtilLeqA` returns the total utilization of some task set that is feasible on π. Therefore, $\texttt{MinUtilLeqA}(\pi, V_\epsilon, u) \leq \mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u) + \epsilon$ for all u such that $\mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u) > S(\pi) - \epsilon$.

Case 2: $\mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u) \leq S(\pi) - \epsilon$.

Let $\tau = (u_1, u_2, \ldots, u_m)$ be a modular task set such that $U_{sum}(\tau) = \mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u)$ and $u_{max}(\tau) \leq u$. By assumption, $U_{sum}(\tau) \leq S(\pi) - \epsilon$. Also, modular task sets always have at least one processor with no gap. Therefore, the average gap on the remaining processors is at least $\epsilon/(m-1)$ so $u_i > \epsilon/(m-1)$ for all $i = 1, 2, \ldots, m$.

By construction of V_ϵ, we know **(i)** $v_1 \leq \epsilon/(m-1)$, and **(ii)** for all $i = 2, 3, \ldots, r$, if $v \in (v_{i-1} - v_i]$ then $v - v_i \leq \epsilon/(m-1)$. Therefore, Lemma 6 applies and there exists a modular task set represented by $(u'_1, u'_2, \ldots, u'_m) \in V_\epsilon^m$ with maximum utilization at most $u_{max}(\tau)$ such that $\texttt{ModUtil}(u'_1, u'_2, \ldots, u'_m) - \texttt{ModUtil}(u_1, u_2, \ldots, u_m) \leq \epsilon$. Moreover, since `MinUtilLeqA` will consider this m-tuple in its search for $\mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(u)$, we know $\texttt{MinUtilLeqA}(\pi, V_\epsilon, u) \leq \texttt{ModUtil}(u'_1, u'_2, \ldots, u'_m)$. Therefore,

$$\texttt{MinUtilLeqA}(\pi, V_\epsilon, u) \leq \texttt{ModUtil}(u_1, u_2, \ldots, u_m) + \epsilon \ .$$

By assumption, $\texttt{ModUtil}(u_1, u_2, \ldots, u_m) = \mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(j)$ so we must have

$$\texttt{MinUtilLeqA}(\pi, V_\epsilon, u) \leq \mathcal{U}_{\pi}^{\textsf{AFD-EDF}}(j) + \epsilon \ .$$

It remains to find a bound on the size of V_ϵ. By definition, the first subset of V_ϵ contains $\lfloor s_i(m-1)/\epsilon \rfloor$ elements for each $i = 1, 2, \ldots m$ and the second subset of V_ϵ contains $\lfloor s_1(m-1)/(\epsilon(k+1)) \rfloor$ elements for each $k = 1, 2, \ldots \lfloor s_1(m-1)/\epsilon \rfloor$. Therefore the union of these two subsets contains at most the sum of their sizes:

$$|V_\epsilon| \leq \sum_{i=1}^{m} \left\lfloor \frac{s_i(m-1)}{\epsilon} \right\rfloor + \sum_{i=1}^{\lfloor s_1(m-1)/\epsilon \rfloor} \left\lfloor \frac{s_1(m-1)}{\epsilon(i+1)} \right\rfloor \ .$$

Dropping the floors in these sums can only increase the right-hand side of this inequality. Therefore,

$$|V_\epsilon| \leq \sum_{i=1}^{m} \left(\frac{s_i(m-1)}{\epsilon} \right) + \sum_{i=1}^{s_1(m-1)/\epsilon} \left(\frac{s_1(m-1)}{\epsilon(i+1)} \right) \ .$$

The first sum above is equal to $(m-1)/\epsilon \cdot \sum_{i=1}^{m} s_i = (m-1)/\epsilon \cdot S(\pi)$. The second sum above contains the constant expression $s_1(m-1)/\epsilon$. Therefore, the inequality above can be simplified as follows.

$$|V_\epsilon| \leq \frac{m-1}{\epsilon} S(\pi) + \frac{s_1(m-1)}{\epsilon} \sum_{i=1}^{s_1(m-1)/\epsilon} \frac{1}{i+1} \ .$$

The sum above is the sum of $1/i$ for $i = 2, 3, \ldots, s_1(m-1)/\epsilon + 1$, which is 1 less than the harmonic number of $s_1(m-1)/\epsilon + 1$. Substituting this in for the sum and factoring $(m-1)/\epsilon$ from the two addends gives

$$|V_\epsilon| \leq \frac{m-1}{\epsilon} \left[S(\pi) + s_1 \left(H_{(s_1(m-1)/\epsilon)+1} - 1 \right) \right] \ ,$$

so the theorem holds. ■

Since AFD-EDF is a special AFD-EDF bin-packing heuristic, the following corollary holds.

Corollary 2 *Let π be any uniform multiprocessor and τ be any task set. For any $\epsilon > 0$, let V_ϵ be the set specified in Theorem 21. If $U_{sum}(\tau) \leq$ `MinUtilLeqA`$(\pi, V_\epsilon, u_{max}(\tau)) - \epsilon$ then τ is* FFD-EDF*-schedulable on π.*

The size of V_ϵ may be reduced somewhat without sacrificing the accuracy of `MinUtilLeqA`. First, the number of processor-speed divisors is not necessarily as much as $S(\pi)(m-1)/\epsilon$. Since the divisors of s_1 are included in the second set, these values do not need to be included in the first set as well. Also, some of these divisors may be double counted if several processor speeds have divisors in common. For example, if π contains a 12- and a 15-speed processor, then the divisor 3 is double counted. In the extreme case, all of π's processor speeds divide s_1 and the first set adds no new points to V_ϵ. If there are processors whose speeds do not divide s_1, the size of the second set can be reduced slightly. Instead of finding all the points that are sufficiently close together and then adding the divisors of the other processor speeds, we can first find the divisors of the processor speeds and then add points to ensure all points are sufficiently close together, as the following example illustrates.

Example 4.3 *Consider the uniform heterogeneous multiprocessor $\pi = [2.5, 2, 1.5, 1]$. Assume we want to estimate the* AFD-EDF *utilization bound with $\epsilon = 0.4$. Then between 2.5 and 1.25, the distance between elements of V_ϵ must be at least $0.4/3 = 0.133$. If we first find all the points $(2.5 - 0.233k)$ and then insert the two points 2 and 1.5 (the only divisors of some speed between 2.5 and 1.25), we get the the following twelve elements: 2.5, 2.367, 2.233, 2.1, 2, 1.967, 1.833, 1.7, 1.567, 1.5 1.433 and 1.3. If, on the other hand, we start with the three points 2.5, 2 and 1.5, and then insert points to insure consecutive points are at most 0.13 units apart, we get the following ten elements: 2.5, 2.3667, 2.233, 2.1, 2, 1.8667, 1.733, 1.6, 1.5 and 1.3667.*

The subroutine `MinUtilLeqA` examines all valid modular task sets whose task utilizations

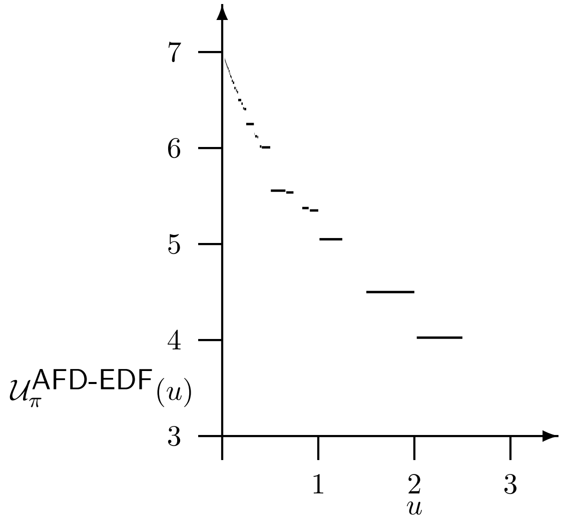


Figure 4.9: The graph of $\mathcal{U}_\pi^{\mathsf{AFD\text{-}EDF}}(u)$ for $\pi = [2.5, 2, 1.5, 1]$ with error bound $\epsilon = 0.1$.

are in V_ϵ. Thus the complexity of this subroutine is $O(|V_\epsilon|^m)$. The algorithm `MinUtilGraph` calls `MinUtilLeqA` at most $|V_\epsilon|$ times. Therefore, the complexity of `MinModUtil` is

$$O(|V_\epsilon|^{m+1}) = O\left(\left(\frac{m-1}{\epsilon}\left[S(\pi) + s_1\left(\sum_{i=1}^{(s_1(m-1)/\epsilon)+1}\left(\frac{1}{i}\right) - 1\right)\right]\right)^{(m+1)}\right).$$

Example 4.4 *Figure 4.9 illustrates the graph that results from executing* `MinUtilGraph` *on the uniform multiprocessor* $\pi = [2.5, 2, 1.5, 1]$ *with a maximum error of* $\epsilon = 0.1$. *For* $u \in (2.025, 2.5]$, *the value of* `MinUtilGraph`(π, V_ϵ) *is* 4.025. *By Theorem 21,*

$$3.925 \leq \mathcal{U}_\pi^{\mathsf{AFD\text{-}EDF}}(u) \leq 4.025 \text{ for all } u \in (2.025, 2.5].$$

Thus, if τ *is a task set with* $u_{max}(\tau) \in (2.025, 2.5]$ *and* $U(\tau) \leq 3.925$, τ *is* AFD-EDF*-feasible on* π. *As* u *approaches 0, the value returned by* `MinUtilGraph` *approaches* $S(\pi)$ *in a near-linear fashion. This is because when* u *is small, the gaps on each processor must also be small (recall that the gaps are smaller than the task utilizations).*

4.3 Summary

This chapter has explored partitioned scheduling on uniform heterogeneous multiprocessors. We have developed a method of approximating the AFD-EDF utilization bound for any uniform heterogeneous multiprocessor. While this method has time complexity exponential in the number of processors, it needs to be executed only once for any given uniform heterogeneous multiprocessor. Once the bound is determined, a task set τ can be tested for

AFD-EDF-feasibility by a linear calculation to determine $u_{max}(\tau)$ and $U_{sum}(\tau)$ and a lookup to verify if $U_{sum}(\tau)$ is below the bound associated with $u_{max}(\tau)$.

Partitioned EDF scheduling can be efficiently scheduled by executing a uniprocessor EDF scheduler on each processor. It has the advantage of incurring no migration overhead. However, there are several disadvantages to partitioning. First, we cannot schedule dynamic task sets using the methods discussed in this chapter. If a new task arrives, we may need to repartition the entire task set, which would cause several migrations. Second, partitioning does not allow us to adjust the load on individual processors. When a task generates a job, that job must execute on a specific processor. In the next section, we discuss restricted migration, which keeps migration overhead down while still permitting flexibility in the schedule and allowing tasks to arrive dynamically.

Chapter 5

Restricted migration EDF (r-EDF)

In the previous two chapters, we determined two EDF schedulability tests — one allowing full migration and another allowing no migration. In this chapter, we will determine a schedulability test for EDF scheduling with restricted migration on uniform multiprocessors, denoted r-EDF. Recall that the r-EDF scheduling algorithm allows tasks to migrate only at job boundaries. Thus, a global scheduler assigns jobs to processors and each processor uses a uniprocessor EDF algorithm to schedule jobs. Figure 5.1 illustrates this algorithm.

The global scheduler keeps track of the slack on each processor. For each k, with $1 \leq k \leq m$, the quantity $slack_i$ is the available capacity on processor s_k. Any job generated by a task of utilization u can be assigned to any processor that has a slack of at least u. If a job is generated by a task with utilization u at a time when each of the processors of π have slack less than u, then the schedule is invalid *even if* enough capacity may become available to accommodate the job at a later point in time. Initially, $slack_k = s_k$ for all k, with $1 \leq k \leq m$. When the global scheduler assigns a job $T_{i,j}$ to processor s_k at time t, the value of $slack_k$ is immediately reduced by u_i. In addition to decreasing the slack, the global scheduler also schedules an interrupt to increase the slack by u_i units at the job's deadline — i.e., at time $t + d_i$.

In the special case when a job completes execution at a point when there are no jobs waiting to execute on processor s_k, the value or $slack_k$ is reset to s_k. This can be done because EDF will only miss deadlines if there is no processor idle time between the earliest release time and the deadline miss [LL73]. Of course, in this case, all of the interrupts scheduled to increase the slack on s_k should be disregarded. This can be done either by cancelling all scheduled increases or, alternatively, by keeping track of when the last reset occurred and ignoring increases associated with jobs that arrived before the most recent reset time. Allowing the slack to be reset also requires a minor change to the local EDF schedulers: When the local EDF scheduler attempts to schedule a new job and the job queue is empty, it must send a "reset slack" signal to the global scheduler. Note that by this definition, r-EDF may generate many different valid r-EDF schedules for the same system. This is illustrated in the following example.

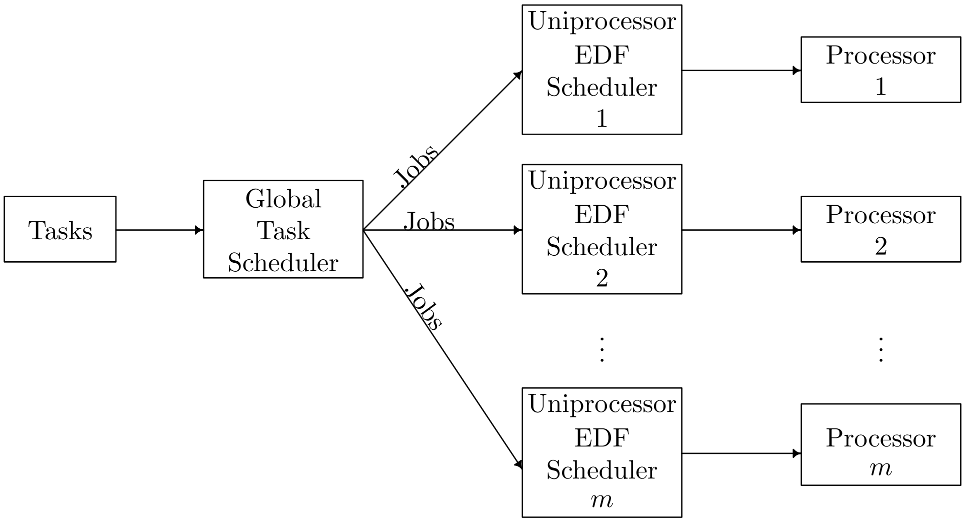

Tasks
Global Task Scheduler
Jobs
Jobs
Jobs
Uniprocessor EDF Scheduler 1
Uniprocessor EDF Scheduler 2
Uniprocessor EDF Scheduler m
Processor 1
Processor 2
Processor m

Figure 5.1: EDFwith restricted migration (r-EDF).

Example 5.1 *Let $\tau = \{T_1 = (1, 2, 3), T_2 = (1, 3, 4), T_3 = (0, 6, 8)\}$ be a periodic task set and let $\pi = [2, 1]$ be a uniform heterogeneous multiprocessor. Insets (b) and (c) of Figures 5.2 illustrate two valid* r-EDF *schedules of τ on π. Inset (a) of Figure 5.2 illustrates the slack of processor s_1 during the schedule illustrated in inset (b). Recall the up arrows in insets (b) and (c) indicate both the arrival of a job and the deadline of the previous job. Also, the height of the rectangle indicates the speed of execution. Thus, taller rectangles indicate the task is executing on processor s_1 and shorter rectangles indicate the task is executing on processor s_2.*

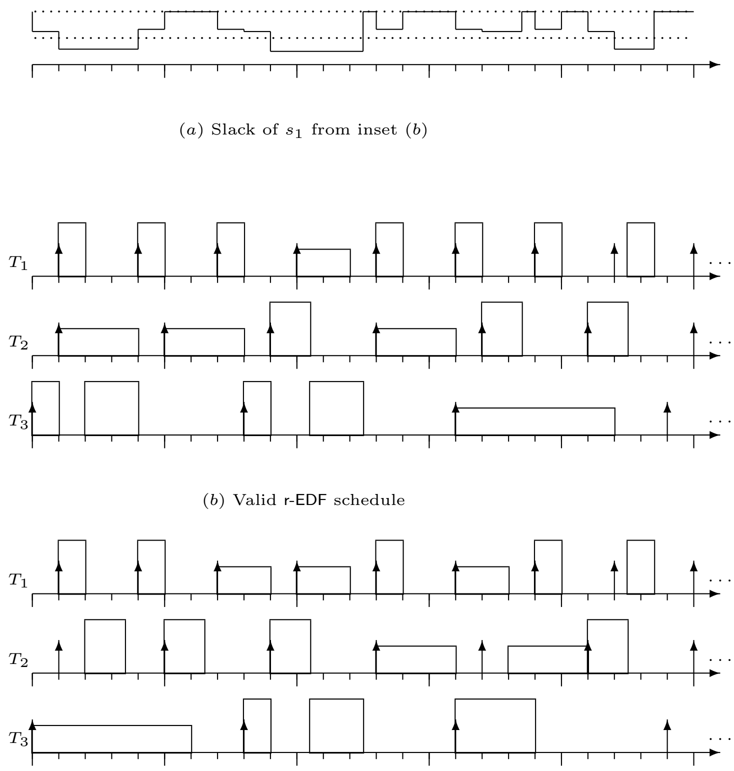
Since $s_1 = 2$, the slack shown in inset (a) is initially 2. At $t = 0$, T_3 has a job scheduled on s_1, so the slack is immediately reduced by the utilization of T_3 to $1\frac{3}{4}$. At $t = 1$, task T_1 has a job scheduled on s_1, so the slack is once again reduced, this time by the utilization of T_1, to $\frac{7}{12}$. The slack increases back to $1\frac{3}{4}$ at $t = 4$, when task T_1's deadline occurs. Notice that at $t = 12\frac{1}{2}$ the slack increases to 2 even though there are no deadlines at that time. This is because the processor becomes idle at that point and the slack is reset.

A task T_i is said to be *present* on processor s_k at time t if that task's utilization contributes to the calculation of $slack_k$ — i.e., if there exists a job $T_{i,j} = (r_{i,j}, e_i, d_{i,j})$ assigned to processor s_k such that $r_{i,j} \leq t < d_{i,j}$ and the most recent reset of $slack_k$ occurred at or before $r_{i,j}$.

Lemma 7 *Let π be any m-processor uniform heterogeneous multiprocessor and let τ be any periodic or sporadic task set such that*

$$U_{sum}(\tau) \leq S(\pi) - (m - 1) \cdot u_{max}(\tau) \ . \tag{5.1}$$

Then whenever any task $T_i \in \tau$ generates a job, there exists at least one processor s_k of π



(a) Slack of s_1 from inset (b)

(b) Valid r-EDF schedule

(c) Valid r-EDF schedule

Figure 5.2: Two valid r-EDF schedules of task set $\tau = \{T_1 = (1, 2, 3), T_2 = (1, 3, 4), T_3 = (0, 6, 8)\}$ on $\pi = [2, 1]$.

such that $slack_k \ \geq \ u_i$.

Proof: (By contradiction.) Let t_o be the earliest time at which some task T_i generates a job $T_{i,j}$ and each of the processors has slack less than u_i. Therefore, $T_{i,j}$ cannot be assigned to any processor at t_o. By the optimality of EDF on uniprocessors all jobs with deadlines at or before t_o meet their deadlines.

Let $P(t_o)$ be the set of tasks that are present on some processor at time t_o. Since T_i has a job that has just arrived and all earlier jobs of T_i meet their deadlines, T_i cannot be in $P(t_o)$. On the other hand, every task of τ other than T_i may be in $P(t_o)$. Therefore,

$$\sum_{T_j \in P(t_o)} u_j \leq U_{sum}(\tau) - u_i \ . \tag{5.2}$$

Since $slack_k < u_i$ for all $k, 1 \leq k \leq m$,

$$\sum_{k=1}^{m} slack_k < m \cdot u_i \ . \tag{5.3}$$

Since the total slack is equal to the total speed of π minus the utilization of all active tasks, we can replace the right-hand side of Inequality (5.3) as follows,

$$S(\pi) - \sum_{T_j \in P(t_o)} u_j < m \cdot u_i \ .$$

Subtracting $S(\pi)$ from both sides and multiplying by (-1) gives

$$\Rightarrow \sum_{T_j \in P(t_o)} u_j > S(\pi) - m \cdot u_i \ . \tag{5.4}$$

Combining Inequalities (5.2) and (5.4) gives

$$U_{sum}(\tau) - u_i > S(\pi) - m \cdot u_i \ .$$

Adding u_i to both sides gives

$$\Rightarrow U_{sum}(\tau) > S(\pi) - (m-1) \cdot u_i$$

Since $u_i \leq u_{max}(\tau)$, we have

$$\Rightarrow U_{sum}(\tau) > S(\pi) - (m-1) \cdot u_{max}(\tau) \ ,$$

which contradicts the condition of the lemma. ■

Notice that if $u_{max}(\tau) > s_m$,

$$S(\pi) - (m-1)u_{max}(\tau) > S_{m-1}(\pi) - (m-2)u_{max}(\tau) \ ,$$

and it may be possible that Lemma 7 applies to the r-EDF schedule of τ on the $(m-1)$ fastest processors of π, but not to the r-EDF schedule of τ on π. In this case, it is beneficial to consider only a *subset* of the processors of π when applying the Lemma 7. Theorem 22 below takes note of this improvement and also shows that the resulting test is a tight one.

First some notation:

Definition 14 ($m_u(\pi)$) *Let π be any uniform heterogeneous multiprocessor and let u be any value. Then $m_u(\pi)$ is the index of the slowest processor of π whose speed is at least u. Specifically,*

$$m_u(\pi) \overset{def}{=} \begin{cases} \max\{j \mid 1 \le j \le m \wedge s_j \ge u\} & \textit{if } u \le s_1(\pi), \textit{ and} \\ \infty & \textit{otherwise.} \end{cases}$$

Theorem 22 *Let π be any m-processor uniform heterogeneous multiprocessor. Let τ be any task set and let $m' = m_{u_{max}}(\pi)$. Then if*

$$U_{sum}(\tau) \le S_{m'}(\pi) - (m'-1)u_{max}(\tau) \tag{5.5}$$

τ is r-EDF*-schedulable on π. Moreover, this is a tight bound — i.e., for any $\hat{u}$ there exists a task set τ' with $u_{max}(\tau') = \hat{u}$ and $m_{\hat{u}}(\pi) = m''$ such that*

$$U_{sum}(\tau') = S_{m''}(\pi) - (m''-1)\hat{u} + \epsilon \tag{5.6}$$

that is not r-EDF*-schedulable on π, where ϵ is an arbitrarily small value.*

Proof: Let S be some r-EDF schedule of τ. Let t_o be any time at which some task T_i generates a job $T_{i,j}$ and every job generated before t_o was able to fit on some processor. It suffices to show that at least one processor s_k has $slack_k \ge u_i$ at time t_o.

Let $\pi' = [s_1, s_2, \ldots, s_{m'}]$ and consider the set of jobs that S schedules on π'. These jobs could be generated by a sporadic task set, τ', containing tasks with the same execution requirements and periods as the tasks in τ — the only difference between τ and τ' is that interarrival times may be longer in τ'. Let τ' be this sporadic task set and let S′ be the schedule of S on π' during the interval $[0, t_o)$. Clearly, S′ is a valid r-EDF schedule of τ' on π'.

Since the tasks of τ' have the same parameters as the tasks of τ, the utilizations $U_{sum}(\tau') = U_{sum}(\tau)$ and $u_{max}(\tau') = u_{max}(\tau)$. Therefore,

$$U_{sum}(\tau') = U_{sum}(\tau) \le S_{m'}(\pi') - (m'-1)u_{max}(\tau) = S(\pi') - (m(\pi')-1)u_{max}(\tau'),$$

so Lemma 7 applies to any r-EDF schedule of τ' on π'. Therefore, whenever a job of τ' is generated, there will always be at least one processor of π' with enough slack to successfully schedule that job. In particular, any r-EDF schedule of τ' on π' (such as S$'$) will be able to successfully schedule $T_{i,j}$. Since the slack on the processors of π' is always the same during the two schedules S and S$'$, $T_{i,j}$ will also fit on at least one processor of π at time $t = t_o$.

It remains to show that the bound is tight. Consider any arbitrary value $\hat{u}$ and let $m'' = m_{\hat{u}}(\pi)$. If $\hat{u} > s_1$, then no task set τ' with $u_{max}(\tau') = \hat{u}$ can be scheduled on π regardless of the algorithm chosen.

Assume $\hat{u} \le s_1$ and define k_j as follows:

$$k_j = \begin{cases} \frac{\lceil s_j - \hat{u} \rceil}{\hat{u}} + 1 & \text{if } 1 \le j \le m'', \\ 1 & \text{if } j = m'' + 1. \end{cases}$$

Clearly $0 < \frac{s_j - \hat{u}}{k_j} < \hat{u}$ for all j, where $1 \le j \le m''$. Choose any values ϵ and δ such that

$$0 < \epsilon \le \min \left\{ m'' \cdot \left(\hat{u} - \frac{s_j - \hat{u}}{k_j} \right) \mid 1 \le j \le m'' \right\}$$

and

$$\delta = \frac{\epsilon}{m''} \ .$$

Define $e_{j,\ell}$ as follows:

$$e_{j,\ell} = \begin{cases} \frac{s_j - \hat{u}}{k_j} & \text{if } 1 \le j \le m'' \text{ and } \ell < k_j, \\ \frac{s_j - \hat{u}}{k_j} + \delta & \text{if } 1 \le j \le m'' \text{ and } \ell = k_j, \\ \hat{u} & \text{if } j = m'' + 1\ . \end{cases}$$

Let τ' be comprised of tasks $T_{j,\ell}$, where $1 \le j \le m'' + 1$, and $1 \le \ell \le k_j$, each with execution requirement equal to $e_{j,\ell}$ and period equal to 1.

Claim. $u_{max}(\tau') = \hat{u}$ **and** $U_{sum}(\tau') = S_{m''}(\pi) - (m'' - 1)\hat{u} + \epsilon$.

Proof of claim. Since all the tasks in τ' have a period of 1, their utilizations are equal to their execution requirements. Hence,

$$u_{max}(\tau') \quad = \quad \max \left\{ \max_{1 \le j \le m''} \left\{ \frac{s_j - \hat{u}}{k_j} \right\}, \max_{1 \le j \le m''} \left\{ \frac{s_j - \hat{u}}{k_j} + \delta \right\}, \hat{u} \right\} \ ,$$

so it suffices to show that $\hat{u} \ge \frac{s_j - \hat{u}}{k_j} + \delta$ for all j, where $1 \le j \le m''$.

By definition of δ,

$$\frac{s_j - \hat{u}}{k_j} + \delta = \frac{s_j - \hat{u}}{k_j} + \frac{\epsilon}{m''} \ .$$

Combining this with the bound on ϵ gives

$$\frac{s_j - \hat{u}}{k_j} + \delta \leq \frac{s_j - \hat{u}}{k_j} + \frac{m'' \cdot \left(\hat{u} - \frac{s_j - \hat{u}}{k_j}\right)}{m''} = \hat{u} \ .$$

Also, by definition of τ',

$$\begin{aligned} U_{sum}(\tau') &= (k_j - 1)\sum_{j=1}^{m''}\sum_{\ell=1}^{k_j} e_{j,\ell} \\ &= (k_j - 1)\sum_{j=1}^{m''}\frac{s_j - \hat{u}}{k_j} + \sum_{j=1}^{m''}(\frac{s_j - \hat{u}}{k_j} + \delta) + \hat{u} \\ &= k_j \sum_{j=1}^{m''}\frac{s_j - \hat{u}}{k_j} + m''\delta + \hat{u} \\ &= S_{m''}(\pi) - m'' \cdot \hat{u} + m'' \cdot \delta + \hat{u} \\ &= S_{m''}(\pi) - (m'' - 1)\hat{u} + \epsilon \ , \end{aligned}$$

where the last step follows because $\delta = \epsilon / m''$. Therefore, the claim is proved.

By the claim, Condition (5.6) of the theorem is satisfied. Assume each task other than $T_{m''+1,1}$ generates a job at $t = 0$ and that the global scheduler assigns the jobs generated by tasks $T_{j,1}, T_{j,2}, \ldots, T_{j,k_j}$ to processor s_j. Then the spare capacity on processor $s_j, j = 1, 2, \ldots, m''$ is $\hat{u} - \delta$. If task $T_{m''+1,1}$ generates a job before any other jobs become inactive, the job will miss its deadline since no processor of π has enough spare capacity. Therefore, τ' is not r-EDF-schedulable. This proves that Condition (5.5) is a tight bound. ■

The result of Baruah and Carpenter [BC03] concerning r-EDF-scheduling on identical multiprocessors is an immediate corollary to Theorem 22 above since the total speed of m unit-speed processors is m.

Corollary 3 *Any periodic task set τ satisfying*

$$U_{sum}(\tau) \leq m - (m-1)u_{max}(\tau)$$

will meet all its deadlines when scheduled on m unit-speed processors using r-EDF.

There are cases where the utilization bound proven in Theorem 22 may be too restrictive. For example, if τ consists of a few tasks with large utilization and several tasks with significantly smaller utilization, it may make more sense to reserve the fastest processors for the higher-utilization tasks and allow the lower-utilization tasks to execute only on slower processors.

Example 5.2 *Consider the uniform heterogeneous multiprocessor* $\pi = [8, 3, 3]$ *and the task set* τ *comprised of 21 tasks with* $u_1 = 4, u_2 = u_3 = 1, u_4 = u_5 = \ldots = u_{11} = 0.5$*, and* $u_{12} = u_{13} = \ldots = u_{21} = 0.1$*. Thus,* $U_{sum}(\tau) = 11$ *and* $u_{max}(\tau) = 4$*. Notice that* $m_{u_{max}} = 1$ *and* $S_1(\pi) = 8$*, so Condition* (5.5) *does not hold* $(11 > 8 - 0 \cdot 4)$ *and this system may not be* r-EDF*-schedulable. However, if we let* τ_1 *be the task set containing the three highest-utilization tasks of* τ*, the system is schedulable since the utilization of* τ_1 *is 6 so these jobs can be* r-EDF *scheduled on the fastest processor of* π*. Also, applying Theorem 22 to* $\pi' = [3, 3]$ *and* τ_2 *with* $u_{max}(\tau') = 0.5$ *and* $U_{sum}(\tau_2) = 5$*, we see that* $(5 \leq 6 - 1 \cdot 0.5)$*, therefore the 18 low-utilization tasks of* τ_2 *can be* r-EDF *scheduled on the two slowest processors of* π*.*

A task set that fails Condition (5.5) may become schedulable by using a minor modification of r-EDF which takes this observation into account. We call this modification *semi-partitioning.*

5.1 Semi-partitioning

Since Condition (5.5) may not hold for systems where the maximum and minimum utilizations of τ are significantly different, we may wish to consider a scheme whereby jobs generated by a given task may be assigned to only a subset of the processors of π. This is not a fully partitioned system because each task may be assigned to several processors. In this modification of r-EDF, called r-EDF(τ_1, ℓ), τ is divided into two disjoint groups, τ_1 and τ_2. All jobs generated by τ_1 are scheduled on the ℓ fastest processors of π and the jobs generated by τ_2 are scheduled on the remaining processors. In general, τ_1 will contain the higher-utilization tasks of τ, since these ar the tasks that require the faster processors, and τ_2 will contain the the lower-utilization tasks of τ.

Example 5.2 illustrates that a system which fails Condition (5.5) may be schedulable using semi-partitioning. Notice that r-EDF(k, ℓ)-schedulability can be tested by simply applying Condition (5.5) twice. The following theorem formalizes this observation.

Theorem 23 *Let* π *be any uniform heterogeneous multiprocessor and let* τ *be any task set. Assume* $\tau = \tau_1 \cup \tau_2$*, where* τ_1 *and* τ_2 *are disjoint. If*

$$\begin{aligned} U_{sum}(\tau_1) &\leq S_\ell(\pi) - (\ell - 1) \cdot u_{max}(\tau_1) \text{ and} \\ U_{sum}(\tau_2) &\leq (S(\pi) - S_\ell(\pi)) - (m - \ell - 1) \cdot u_{max}(\tau_2) \ , \end{aligned}$$

then τ *is* r-EDF(τ_1, ℓ)*-schedulable on* π*.*

If the two conditions in Theorem 23 are added together, the resulting condition is

$$U_{sum}(\tau) \leq S(\pi) - (\ell - 1)u_{max}(\tau_1) - (m - \ell - 1)u_{max}(\tau_2) \ .$$

Thus, Theorem 23 always imposes a looser bound on $U_{sum}(\tau)$ than Theorem 22 since

$$\begin{aligned} S(\pi) - (\ell - 1)u_{max}(\tau_1) - (m - \ell - 1)u_{max}(\tau_2) &\geq S(\pi) - (m - 2\ell - 2)u_{max}(\tau) \\ &> S(\pi) - (m - 1)u_{max}(\tau) \ . \end{aligned}$$

The difference between these two bounds may be significant if $u_{k+1} \ll u_{max}(\tau)$ or if ℓ is large. Also, if $u_{max}(\tau)$ is larger than the speed of one or more processors of π some of the tasks are defacto semi-partitioned. For example, since the task of τ with highest utilization in Example 5.2 has utilization larger than the speed of the two slower processors of π, we can see that it can only be executed on the fastest processor of π even if the system had not been semi-partitioned. Thus, task sets with widely divergent utilizations may benefit from semi-partitioning — particularly if $u_{max}(\tau) > s_m$. These two observations lead us to the following two heuristics for determining how to semi-partition a system, which are applied to the system after τ is sorted by utilization.

- If $u_{max}(\tau) > s_m$, determine the index ℓ such that $s_\ell \geq u_{max}(\tau) > s_{\ell+1}$ and determine the largest index k for which the first condition of Theorem 23 holds:

  $$k = \max\{j \mid 1 \leq j \leq n \wedge \sum_{i=1}^{j} u_i \leq S_\ell(\pi) - (\ell - 1)u_{max}\} \ .$$

  Let $\tau_1 = \{T_1, T_2, \ldots, T_k\}$.

- Otherwise, consider the tasks of τ. If $u_{max}(\tau) \gg u_{min}(\tau)$, then semi-partitioning may be useful because of the widely divergent utilizations of τ. If there is some point where the utilization decreases significantly, let T_k be the highest-utilization task before this decrease. (For example, let A equal the average of (u_i/u_{i+1}) for all $i, i = 1, 2, \ldots, n-1$, and choose the smallest k such that $u_k/u_{k+1} > thresh \times A$ for some threshold $thresh$.) If there is no point at which the utilization decreases significantly, then let $k = \lfloor \frac{n}{2} \rfloor$. Let $\tau_1 = \{T_1, T_2, \ldots, T_k\}$ and determine the smallest index ℓ for which the first condition of Theorem 23 holds:

  $$\ell = \min\{j \mid 1 \leq j \leq m \wedge U_{sum}(\tau_1) \leq S_j(\pi) - (j - 1)u_{max}(\tau_1)\} \ .$$

Once an appropriate τ_1 and ℓ have been chosen, test if the the second condition of Theorem 23 holds. Of course, it is possible that the second condition does not hold. In this case, semi-partitioning can be done several times. The system can be semi-partitioned as many as $(m-1)$ times, where $(m-1)$ semi-partitions results in a fully partitioned system. The notation r-EDF$(\tau_1, m_1; \tau_2, m_2; \ldots; \tau_r, m_r)$ denotes the r-EDF schedule with the following restrictions

$$\begin{gathered}
\tau_1 \text{ executes only on processors } s_1, s_2, \ldots, s_{m_1} \\
\tau_2 \text{ executes only on processors } s_{m_1+1}, s_{m_1+2}, \ldots, s_{m_2} \\
\vdots \\
\text{Tasks } \tau_{r+1} \text{ execute only on processors } s_{m_r+1}, s_{m_r+2}, \ldots, s_m.
\end{gathered}$$

Thus, the test for r-EDF$(\tau_1, m_1; \tau_2, m_2; \ldots; \tau_r, m_r)$-schedulability would involve $(r+1)$ applications of Theorem 22, each of which must be satisfied.

Theorem 24 *Let π be any uniform heterogeneous multiprocessor and let τ be any task set. Assume $\tau_1, \tau_2, \ldots, \tau_{r+1}$ are disjoint and $\tau_1 \cup \tau_2 \cup \ldots \cup \tau_{r+1} = \tau$. If*

$$\begin{aligned}
U_{sum}(\tau_1) &\leq S_{m_1}(\pi) - (m_1 - 1) \cdot u_{max}(\tau_1) \ , \\
U_{sum}(\tau_2) &\leq (S_{m_2}(\pi) - S_{m_1}(\pi)) - (m_2 - m_1 - 1) \cdot u_{max}(\tau_2) \ \textit{and} \\
&\vdots \\
U_{sum}(\tau_{r+1}) &\leq (S(\pi) - S_{m_r}(\pi)) - (m - m_r - 1) \cdot u_{max}(\tau_{r+1}) \ .
\end{aligned}$$

then τ is r-EDF$(\tau_1, m_1; \tau_2, m_2; \ldots; \tau_r, m_r)$*-schedulable on π.*

5.2 Virtual processors

Example 5.2 in the previous section illustrates that a system that does not satisfy Condition (5.5) of Theorem 22 can still be schedulable using a minor modification of r-EDF. Notice that the first partition in this example schedules the three highest-utilization tasks on a single processor of speed 8. Since the total utilization of these three tasks is 6, there are two units of spare capacity available which can be "loaned" to the system (π', τ_2) if necessary. For example, if 6 more tasks with execution utilization 0.1 were added to τ, $U_{sum}(\tau_2)$ becomes 5.6 and Condition (5.5) fails on the system (π', τ_2) since $(5.6 > 6 - 1 \cdot 0.5)$. If π' "borrows" capacity from the faster partition, the fastest processor of π would be divided into 2 virtual processors — one processor of speed 6 devoted to the highest-utilization tasks of τ and one processor of speed 2 devoted to τ_2. Once the 2 units of capacity are borrowed, Condition (5.5) is satisfied since $5.6 \leq (6+2) - 2 \cdot 0.5$. (We now subtract $2 \cdot 0.5$ instead of $1 \cdot 0.5$ because π' has an extra processor with speed $= 2$.) The implementation of virtual processors only affects the *global* scheduling algorithm. Once a job is assigned to a certain processor, the local EDF scheduling algorithm is used regardless of which partition the job belongs to.

If capacity can be borrowed, we have more flexibility in determining the semi-partitioning (i.e., in determining τ_1 and ℓ). For example, since the highest-utilization task of τ in Example 5.2 can only be executed on the fastest processor of π and all the remaining tasks can be executed on any of the processors of π, a natural semi-partition would be to let $\tau_1 = \{T_1\}$ and $\ell = 1$. Therefore, T_1 would be restricted to the fastest processor and the remaining tasks would be restricted to the slower two processors. Applying the conditions of Theorem 23 to this semi-partition gives $(4 \leq 8 - 0)$ and $(7 \leq 6 - 1)$. The second test clearly fails. However, if the four units of spare capacity remaining from the first test are given to the second partition, we have $(4 \leq 4 - 0)$ and $(7 \leq 10 - 2)$. Therefore, this system can be scheduled with the given semi-partition provided the fastest processor is divided into two virtual processors. We use the notation r-SVP(τ_1, ℓ, b) to denote r-EDF scheduling with semi-partitioning and virtual processors, where τ_1 are the tasks restricted to the ℓ fastest processors and the remaining tasks, executing on the $m - \ell$ slower processors, can borrow as much as b units of capacity. Thus, the semi-partition with virtual processor discussed above is denoted r-SVP$(\{T_1\}, 1, 4)$.

By Theorem 22, a set τ' of k tasks scheduled on m' processors using r-EDF requires at least $R = U_{sum}(\tau') + (m' - 1)u_{max}(\tau')$ units of capacity to ensure all deadlines are met. If S is the total capacity of the m' processors and $S > R$, there will be $S - R$ units of wasted capacity in this system. Therefore, this capacity can be loaned to a semi-partition with lower-utilization tasks. The following lemma states that there will always be at least one processor with enough capacity to loan to the τ_2 provided $u_{max}(\tau_1) \geq u_{max}(\tau_2)$ and no more than $S - R$ units are ever loaned to τ_2 at one time.

Lemma 8 *Let π be any uniform heterogeneous multiprocessor and let τ be any task set. Let τ_1 and τ_2 be two disjoint task sets such that $\tau = \tau_1 \cup \tau_2$ and $u_{max}(\tau_1) \geq u_{max}(\tau_2)$. Choose ℓ and b such that*

$$0 \leq b \leq S_\ell(\pi) - U_{sum}(\tau_1) - (\ell - 1)u_{max}(\tau_1) \ ,$$

and schedule τ on π using algorithm r-SVP(τ_1, ℓ, b). *Assume a job $T_{i,j}$ arrives at time t_o, where $T_i \in \tau_2$ and let $P_i(t_o)$, $i = 1, 2$, be the set of tasks in τ_i that are present on some processor s_k, $1 \leq k \leq \ell$. If the following condition holds:*

$$b - \sum_{T_j \in P_2(t_o)} u_j \geq u_i \tag{5.7}$$

then there exists at least one processor $s_k, 1 \leq k \leq \ell$, with $slack_k \geq u_i$.

Proof: (By contradiction.) Assume $slack_k < u_i$ for all $1 \leq k \leq \ell$ at time t_o. Therefore,

$$\sum_{r=1}^{\ell} slack_r < \ell \cdot u_i \ . \tag{5.8}$$

Since $P_1(t_o) \subseteq \tau_1$, we know

$$\sum_{T_j \in P_1(t_o)} u_j \leq U_{sum}(\tau_1) \ . \tag{5.9}$$

Rearranging Condition (5.7) gives

$$\sum_{T_j \in P_2(t_o)} u_j \leq b - u_i \tag{5.10}$$

Since all the jobs executing on the processors of $S_\ell(\pi)$ at time t_o are generated either by tasks in $P_1(t_o)$ or in $P_2(t_o)$, we have

$$S_\ell(\pi) = \sum_{T_j \in P_1(t_o)} u_j + \sum_{T_j \in P_2(t_o)} u_i + \sum_{r=1}^{\ell} slack_r \ . \tag{5.11}$$

Substituting Conditions (5.9) and (5.10) into Equation 5.11 gives

$$S_\ell(\pi) \leq U_{sum}(\tau_1) + (b - u_i) + \sum_{r=1}^{\ell} slack_r \ .$$

Substituting Condition (5.8) gives

$$S_\ell(\pi) < U_{sum}(\tau_1) + (b - u_i) + \ell \cdot u_i \ .$$

Subtracting $(U_{sum}(\tau_1) + b)$ from both sides gives

$$(\ell - 1)u_i > S_\ell(\pi) - U_{sum}(\tau_1) - b \ .$$

Recall, $b \leq S_\ell(\pi) - U_{sum}(\tau_1) - (\ell - 1)u_{max}(\tau_1)$. Substituting this into the inequality above gives

$$(\ell - 1)u_i > b + U_{sum}(\tau_1) + (\ell - 1)u_{max}(\tau_1) - U_{sum}(\tau_1) - b \ .$$

Notice that the right-hand side of the above inequality equals $(\ell - 1)u_{max}(\tau_1)$. Therefore, we have $u_i > u_{max}(\tau_1)$, which contradicts the assumption that $u_{max}(\tau_2) \leq u_{max}(\tau_1)$. ■

By Lemma 8, a task $T_i \in \tau_2$ will not fit on any of the ℓ fastest processors *only if* assigning $T_{i,j}$ to the first semi-partition would cause the borrowed capacity to exceed b. Therefore, we can think of the total capacity that gets loaned to the lower-utilization semi-partition as a single processor *even if* the loaned capacity is spread over several processors.

Definition 15 (r-SVP(τ_1, ℓ, b)) *Let π be any uniform multiprocessor and let $\tau = \tau_1 \cup \tau_2$ be any task set such that $u_{max}(\tau_1) \geq u_{max}(\tau_2)$ and τ_1 and τ_2 are disjoint. Then* r-SVP(τ_1, ℓ, b) *indicates the scheduling algorithm in which the jobs generated by τ_1 are* r-EDF-*scheduled on processors $s_1, s_2, \ldots, s_\ell$ and the jobs generated on τ_2 are scheduled on any processor* provided that *the total utilization of tasks in τ_2 that are present on the ℓ fastest processors of π never exceeds b.*

Determining appropriate semi-partitions with virtual processors is similar to determining appropriate semi-partitions without virtual processors, though one step is added to the process. Once an appropriate τ_1 and ℓ have been determined, let

$$b = S_\ell(\pi) - U_{sum}(\tau_1) - (\ell - 1)u_{max}(\tau_1).$$

Then b is the spare capacity in the first semi-partition which can be loaned to the second semi-partition. Test if τ_2 is r-EDF-schedulable a multiprocessor π', where $S(\pi') = S(\pi) - S_\ell(\pi)$ and $m(\pi') = m - \ell + 1$. (Since the heuristics are applied to sorted task sets, it is clear that $u_{max}(\tau_1) \geq u_{max}(\tau_2)$.)

The following theorem formalizes these steps for finding allowable semi-partitions.

Theorem 25 *Let π be any uniform heterogeneous multiprocessor and let $\tau = \tau_1 \cup \tau_2$ be any task set such that $u_{max}(\tau_1) \geq u_{max}(\tau_2)$ and τ_1 and τ_2 are disjoint. If*

$$\begin{aligned} 0 \leq b &\leq S_\ell(\pi) - U_{sum}(\tau_1) - (\ell - 1) \cdot u_{max}(\tau_1), \ \textit{and} \\ U_{sum}(\tau_2) &\leq S(\pi) - S_\ell(\pi) + b - (m - \ell) \cdot u_{max}(\tau_2), \end{aligned}$$

then τ is r-SVP(τ_1, ℓ, b)*-schedulable on π.*

Proof: The conditions on b ensure that there will always be enough capacity for the higher-utilization tasks to meet their deadlines. The second condition is an application of Theorem 22 on the $(m - \ell)$ slowest processors plus the virtual processor of capacity b. ■

Semi-partitioning with virtual processors can be repeated several times in the same manner as semi-partitioning without virtual processors. Allowing more semi-partitions with virtual processors will allow some task sets to be successfully scheduled on π even though they would be unschedulable if fewer semi-partitions are used. The main difference between having several semi-partitions with virtual processors as opposed to without virtual processors is that while $(m - 1)$ semi-partitions without virtual processors results in a fully partitioned system, the same does not hold for semi-partitioning with virtual processors. Thus, the scheduling algorithm r-SVP$(\tau_1, m_1, b_1; \tau_2, m_2, b_2; \ldots; \tau_r, m_r, b_r)$ generates an r-EDF schedule with the following restrictions

The task sets $\tau_1, \tau_2, \ldots, \tau_{r+1}$ are disjoint
τ_1 can execute only on processors $s_1, s_2, \ldots, s_{m_1}$
τ_2 can execute only on processors $s_1, s_2, \ldots, s_{m_2}$

$$\text{At most } b_1 \text{ units of } \tau_2 \text{ can be present on } s_1, s_2, \ldots, s_{m_1}$$

$$\vdots$$

$$\tau_r \text{ can execute only on processors } s_{m_{r-2}+1}, s_{m_{r-2}+2}, \ldots, s_{m_r}$$
$$\text{At most } b_{r-1} \text{ units of } \tau_r \text{ can be present on } s_{m_{r-2}+1}, s_{m_{r-2}+2}, \ldots, s_{m_{r-1}}$$
$$\tau_{r+1} \text{ can execute only on processors } s_{m_{r-1}+1}, s_{m_{r-1}+2}, \ldots, s_m$$
$$\text{At most } b_r \text{ units of } \tau_{r+1} \text{ can be present on } s_{m_{r-1}+1}, s_{m_{r-1}+2}, \ldots, s_{m_r}$$

Thus, the test for r-SVP$(\tau_1, m_1, b_1; \tau_2, m_2, b_2; \ldots; \tau_r, m_r, b_r)$-schedulability would involve $(r+1)$ applications of Theorem 22, each of which must be satisfied.

Theorem 26 *Let π be any uniform heterogeneous multiprocessor and let τ be any task set. Assume τ is comprised of $(r+1)$ disjoint task sets $\tau_1, \tau_2, \ldots, \tau_{r+1}$ with $u_{max}(\tau_i) \geq u_{max}(\tau_{i+1})$ for all $1 \leq i \leq r$. If*

$$\begin{aligned}
0 \leq b_1 &\leq S_{m_1}(\pi) - U_{sum}(\tau_1) - (m_1 - 1) \cdot u_{max}(\tau_1), \\
0 \leq b_2 &\leq S_{m_2}(\pi) - S_{m_1}(\pi) + b_1 - U_{sum}(\tau_2) - (m_2 - m_1) \cdot u_{max}(\tau_2), \\
&\vdots \\
0 \leq b_j &\leq S_{m_j}(\pi) - S_{m_{j-1}}(\pi) + b_{j-1} - U_{sum}(\tau_j) - (m_j - m_{j-1}) \cdot u_{max}(\tau_j), \\
&\vdots \\
0 &\leq S_m(\pi) - S_{m_r}(\pi) + b_r - U_{sum}(\tau_1) - (m - m_r) \cdot u_{max}(\tau_{r+1})
\end{aligned}$$

then τ is r-SVP$(\tau_1, m_1, b_1; \tau_2, m_2, b_2; \ldots; \tau_r, m_r, b_r)$*-schedulable on π.*

Notice that the i'th semi-partition can only borrow from the $(i-1)$'th semi-partition. This is because the b_k units of spare capacity in the k'th semi-partition are reserved for τ_{k+1} — if some other task uses some of that capacity then τ_{k+1} is no longer guaranteed to meet all of its deadlines.

5.3 The r-SVP scheduling algorithm

The previous two sections introduce three variations of r-EDF — one with no partitioning, one with semi-partitioning and one with semi-partitioning and virtual processors (r-SVP). The difference between these variations of r-EDF is the global scheduler they use to assign jobs to processors as the tasks generate them. This section introduces r-SVP — the restricted-migration, semi-partitioning, global scheduler with virtual processors. The other variations of global schedulers can be derived by removing the implementation of the semi-partitioning and/or the virtual processor from the r-SVP scheduler. This scheduling algorithm can successfully schedule any task sets satisfying the conditions of Theorem 26.

The r-SVP scheduler is illustrated in Figure 5.3. This algorithm maintains two variables, SemiPartMap and Slack. SemiPartMap is an $(s \times 3)$ array, where s is the number of semi-partitions in the system. It maintains the minimum and maximum processor indices associated with each semi-partition, denoted SemiPartMap$[*, min]$ and SemiPartMap$[*, max]$, respectively, and the capacity available to loan to the next lower-utilization semi-partition, SemiPartMap$[*, loan]$. Slack is an array of m elements, where m is the number of processors in the system. Slack$[k]$ is the slack on the k'th processor of π. Initially, Slack$[k] = s_k$ for each $k, 1 \leq k \leq m$. Slack$[k]$ is modified whenever the set of tasks present on s_k changes — i.e., when a job is assigned to s_k, a job's deadline elapses, or s_k becomes idle. In the first and second cases, Slack$[k]$ is reduced or increased, respectively, by the utilization associated with the job. In the third case, Slack$[k]$ is reset to s_k.

When a job is generated by a task and submitted to the r-SVP global scheduler, it must provide three parameters: the semi-partition to which the task belongs, sp, the task's utilization, $util$, and the job's deadline d. r-SVP begins in lines 7 and 8 by finding the processors in the semi-partition sp. It then determines which processor to assign the job to. In line 9 r-SVP finds the processor s_k in semi-partition sp with the most slack. If this processor has enough slack (i.e., if $slack_k \geq util$), then the job will be assigned to s_k on line 25. Otherwise, r-SVP will search for a processor in semi-partition $\mathsf{sp} - 1$ to assign to job to. In lines 12 through 14, r-SVP finds the processor in the prior semi-partition with the largest slack. If this slack is at least $util$, then r-SVP selects this processor, decreases SemiPartMap$[\mathsf{sp} - 1, loan]$ by $util$, and sets an interrupt to increase SemiPartMap$[\mathsf{sp} - 1, loan]$ when the job's deadline elapses on lines 16 and 17. If there was no processor with enough slack to fit the job in semi-partition sp-1 or if sp=1 and there is not enough slack to fit the job in semi-partition sp, r-SVP reports an error on line 19 or 22. Assuming a processor is found on which the job will fit, the slack of this processor is decreased by the task's utilization (line 26) and an interrupt is set to increase the utilization once the task's next deadline arrives (line 27). Since r-SVP uses the values in Slack[*] to determine the processor assignments, it is essential that the interrupts that occur at job deadlines have higher priority than the r-SVP algorithm. Otherwise, if a job arrives at another job's deadline, r-SVP may erroneously determine that the arriving job can not fit onto any processor.

The version of r-SVP shown in Figure 5.3 can be varied in several ways — either to address different system requirements or to optimize the algorithm. For example, in Figure 5.3, r-SVP assigns jobs to the processor with the most slack. However, any method may be used to select among the eligible processors (e.g., first fit, best fit, worst fit, random fit). The choice depends on the load-balancing goal for the given system.

```
function r-SVP($job, sp, util, d$)
    %Submit $job$ with utilization = $util$ and
    % deadline = $d$ to some processor in
    % semi-partition $sp$
    if $sp < 1$ or $sp > maxsp$
        **error: invalid $sp$ value**
    fi
    $minproc \leftarrow$ SemiPartMap[$sp, min$]
    $maxproc \leftarrow$ SemiPartMap[$sp, max$]
    $j \leftarrow$index in [$minproc, maxproc$] with maximum Slack
    if Slack[$j$] $< util$
        if $sp > 1$
            $minproc \leftarrow$ SemiPartMap[$sp - 1, min$]
            $maxproc \leftarrow$ SemiPartMap[$sp - 1, max$]
            $j \leftarrow$index in [$minproc, maxproc$] with maximum Slack
            if Slack[$j$] $\geq util$
                SemiPartMap[$sp - 1, loan$] $\leftarrow$ SemiPartMap[$sp - 1, loan$] $- util$
                set interrupt to increase SemiPartMap[$sp - 1, loan$] by $util$ at time $d$
            else
                **error**
            fi
        else
            **error**
        fi
    fi
    submit $job$ to processor $j$
    Slack[$j$] $\leftarrow$ Slack[$j$] $- util$
    set interrupt to increase Slack[$j$] by $util$ at time $d$
return
```

Figure 5.3: The r-SVP global scheduler.

5.4 Summary

This chapter presents the r-EDF scheduling algorithm and its associated utilization bound. It then presents conditions under which the utilization bound is too restrictive, making r-EDF impractical in the sense that an excessive amount of computing power is forced to remain idle if we are to ensure that all deadlines will be met. This shortcoming is addressed by introducing the concept of semi-partitioning and virtual processors. Finally, an algorithm is presented for scheduling r-SVP, the global r-EDF scheduler with semi-partitions and virtual processors.

Chapter 6

Conclusions and future work

Real-time systems can now be found in our cars, our cell phones — even some of our standard home appliances. As the uses for real-time systems become more diverse, these systems require a larger variety of platforms. This dissertation has provided a first step in availing uniform heterogeneous multiprocessors to the real-time community. To date, research on real-time systems has focussed primarily on uniprocessors and identical multiprocessors. Since multiprocessors are currently being designed with processors that operate at different speeds, it behooves the real-time community to take advantage of uniform heterogeneous multiprocessor technology.

This dissertation presents EDF-schedulability tests for uniform heterogeneous multiprocessors under three different migration strategies — full migration, partitioning, and restricted migration. Table 6.1 illustrates the context in which this research has been developed. While EDF-schedulability tests using each of these three migration strategies have been developed for identical multiprocessors, no such tests existed for uniform heterogeneous multiprocessors. This dissertation provides tests for each of the migration strategies, thereby remedying this deficiency in the current research.

Both the full and restricted migration tests can be performed in polynomial time. The partitioning test requires exponential time. We anticipate this test will be executed by the manufacturers and the results will be provided with other parameters of the system upon delivery, such as the clock speed, bus speed and cache size. After performing this expensive calculation, it is a simple polynomial time calculation to confirm whether a task set is guaranteed to meet all deadlines using partitioned EDF. The research presented in this dissertation is intended to provide a theoretical foundation for the real-time community to use this new type of computing platform.

The remainder of this chapter is divided into 2 sections. Section 6.1 discusses the three EDF-schedulability tests presented in this dissertation in more detail. Section 6.2 discusses some future work.

	Identical	**Uniform Heterogeneous**	**Unrelated Heterogeneous**
Full	Phillips, et al. [PSTW97]	This Research	Future Research
Restricted	Baruah and Carpenter [BC03, BC]		
Partitioning	Lopez, et al. [LGDG00]		

Table 6.1: Context for this research and future research.

6.1 The EDF-schedulability tests

Chapter 3 discusses the full migration EDF test (f-EDF) in detail. Developing this test requires two fundamental techniques. First, resource augmentation [KP98, PSTW97] is a technique whereby the behavior of a real-time instance I on one multiprocessor is evaluated by considering the behavior of the same real-time instance on a *less powerful* multiprocessor. In particular, this chapter infers the f-EDF-schedulability of a real-time instance on a processor π by considering the feasibility of the same real-time instance on a less powerful processor π'. If the following condition is satisfied:

$$S(\pi) \geq S(\pi') + \lambda(\pi) \cdot s_1(\pi') \ ,$$

then any real-time instance that is feasible on π' will be f-EDF-schedulable on π. Unfortunately, resource augmentation does not identify all multiprocessors π' for which feasibility on π' guarantees f-EDF-schedulability on π. We applied the robustness of f-EDF on uniform heterogeneous multiprocessors [Bar02] to find the remaining processors. Using these two techniques, we were able to find all points (s, S) for which feasibility on some multiprocessor with total speed S and fastest speed s implies f-EDF-schedulability on π. Finally, we applied these results to tasks sets to find a utilization-based f-EDF-schedulability test.

Unfortunately, f-EDF is not always a practical scheduling algorithm. For example, in systems with high migration overhead, migration must be either restricted or prohibited altogether. With this in mind, we developed two other schedulability tests for EDF on uniform heterogeneous multiprocessors.

Chapter 4 develops a schedulability test for partitioned EDF on uniform heterogeneous multiprocessors. We show that determining whether a task set τ can be partitioned onto a uniform heterogeneous multiprocessor π is equivalent to the variable-sized bin packing problem, which is known to be NP-complete in the strong sense. Given this observation, we explore methods for approximating the p-EDF schedulability test for uniform heterogeneous multiprocessors. We first sort the tasks by utilization before partitioning. We show that if the tasks are sorted, we can estimate the bound by considering highly simplified tasks sets and discrete utilization values. The complexity of finding the p-EDF utilization bounds is

$$\mathcal{O}\left(m^{\frac{S(\pi)+\log_2(\lfloor s_1(\pi)/\epsilon\rfloor)}{\epsilon}}\right),$$

where ϵ is the maximum error in the estimated bound. This is admittedly an expensive calculation. However, the bound need only be evaluated once for a given uniform heterogeneous multiprocessor π. Once the bound is known, a task set can be checked in polynomial time to verify whether it meets the bound.

p-EDF has advantages and disadvantages. One advantage is that is is fairly easy to implement — simply execute the uniprocessor algorithm in parallel. Furthermore, it clearly reduces the migration overhead. One *dis*advantage of using p-EDF is that it can only schedule static task sets. If tasks join the system, all tasks may need to be repartitioned, which would violate the policy of no migrations. Another disadvantage of p-EDF is that the scheduler has no control over processor loads. If the scheduler could migrate jobs, it could make decisions with an overall system goal in mind. For example, jobs may be assigned to processors in an attempt to balance loads as much as possible. Alternatively, the scheduler may try to keep one of the processors free to execute non-real-time jobs. Either way, these goals cannot be achieved using p-EDF. Instead, we may choose to restrict migration so that the migration costs are kept low while the system still can have dynamic tasks and direct processor assignments according to a specific load profile.

Chapter 5 developed a schedulability test for EDF with restricted migration. We developed this test by considering the worst possible configuration of job assignments that could cause a job to miss its deadline. Based on this analysis, we found that if

$$U_{sum}(\tau) \leq S(\pi) - (m-1) \cdot u_{max}(\tau)$$

then τ is r-EDF-schedulable on π. Because we found that this test may force us to leave a large portion of the system idle, we considered *semi-partitioning* — dividing both the processors and the tasks into two or more groups and restricting all jobs generated by a task to execute only on processors in the task's group. The schedulability test for semi-partitioned r-EDF is a repeated application of the original r-EDF-schedulability test. Finally, we considered *virtual processors*, which allow the excess idle capacity in one partition to be used by another semi-partition and we presented the algorithm r-SVP, which implements r-EDF with semi-partitioning and virtual processors.

6.2 Future work

The work presented in this dissertation generalizes known results for EDF scheduling on identical multiprocessors. For both the full and restricted migration, the generalized tests reduce to the known tests when applied to identical multiprocessors. For partitioning, the uniform heterogeneous multiprocessor test cannot be expressed as a simple mathematical expression so the two tests cannot be compared.

Every generalization of the model expands the number of systems that can be used by the real-time community. Therefore, further extensions of this work include generalizing the processing model and the job (or task) model. Furthermore, many of these generalized models will require new scheduling algorithms. The remainder of this section discusses each of these future goals.

6.2.1 Generalizing the processing model

There are a variety of ways in which the processor model can be generalized. For example, processing speeds can vary by processor *and* job or they can vary over time. In addition the system speeds can refer to communication speeds as opposed to processing speeds.

Unrelated multiprocessors

Just as uniform heterogeneous multiprocessors are a generalization of the identical multiprocessor model, unrelated multiprocessors are a generalization of the uniform heterogeneous multiprocessor model. Recall that in unrelated multiprocessors, the rate of execution depends upon both the processor and the job. These systems are comprised of processors with different architectures — oftentimes, some of the processors, such as graphics co-processors or digital signal processors, are specialized for specific operations. Extending the EDF-schedulability tests to apply to this type of system would be an important contribution to the real-time community. While some of the techniques employed in this dissertation may be extended to apply to this more general model, the extensions are not straightforward because the processors cannot be sorted according to speed — the sort order would depend on both the job and the processor. Much of the analysis performed in this dissertation relied on the sorting of the processors according to speed.

Variable voltage processors

Rather than having processing speeds vary over jobs, we could instead have the processing speeds vary over time. In this case, the model is still uniform in the sense that processing speeds apply uniformly to all jobs. However, these speeds are no longer fixed. This kind of model arises for *variable voltage processors*, in which the adjusting the voltage available to the processor adjusts its speed. These processors are often used in mobile devices such as laptops. When the voltage is increased, both the processor speed and the power consumption are also increased. There are a variety of reasons why a system designer may choose to reduce the power consumption at the expense of reducing the processing speed.

- If battery power dictates design decisions, the system designer may choose to reduce the voltage in order to extend the battery life.

- If heat generation dictates design decisions, the system designer may choose to reduce the voltage whenever possible. Since faster clock speeds generate more heat, reducing the voltage would reduce clock speed, which in turn reduces the heat generated by the system.

- If heat generation dictates design decisions, the system designer may choose to reduce the power consumption in order to reduce the system operating costs.

- Finally, if environmental factors dictate design decisions, the system designer may choose to reduce the power consumption in order to reduce the impact to the environment.

It is easy to see how the research presented in this dissertation extends to *fixed* variable voltage multiprocessors — i.e., systems comprised of variable voltage multiprocessors in which the voltage level of each processor is set before the system begins executing and is not changed thereafter. The tricky question is what voltage settings minimize power consumption while still ensuring jobs meet their deadlines. Another interesting question is how to analyze these systems if the voltage levels can change dynamically over time. The results presented in this dissertation assume fixed processing speeds. Extending this results to account for dynamically changing speeds would be an interesting and important problem.

Storage area networks

Another generalization of the processor model is to reinterpret the processing speed as *communication* speed. In this case, all the jobs would be managing data stored in a storage area network — i.e., a subnetwork of storage devices. Instead of scheduling jobs to ensure they complete before their deadline, this interpretation would be scheduling data retrievals to ensure they complete in a timely manner. However, the data is written as well as read. Therefore, the scheduling may also benefit from moving regularly-accessed data to the fastest locations. While this problem is significantly different than the hard-real-time scheduling problem, the two problems do share similarities. In particular, the techniques employed in this dissertation such as resource augmentation for full migration, finding the worst-case scenario for restricted migration, and modularizing utilization per processor for partitioning may translate into analysis of storage area networks by replacing processing speeds with communication costs. Of course, some of the analysis would have to be modified to reflect the different application. For example, data retrieval jobs cannot be preempted at arbitrary times so the analysis would have to account for a delay before a preemption can occur.

6.2.2 Generalizing the job model

All of the results presented in this dissertation assume a preemptive model in which jobs are independent. There are many problems in which independence is an unreasonable assumption. Also, in some cases preemption may not be allowed.

Dependent jobs

Jobs may have dependencies for a variety of reasons. Two types of job dependencies are *resource sharing* and *precedence constraints.* If jobs J_1 and J_2 share an exclusive resource, they cannot execute simultaneously. Once one job accesses the resource, the other job will stall until the first job stops using it. If job J_1 has precedence over J_2, then J_2 cannot be scheduled until J_1 has completed executing even if J_2 has already been released. Thus, in addition to not being allowed to execute simultaneously, jobs with a precedence constraint must execute in a particular order. Job J_1 is generally given precedence over J_2 when J_2's correct execution depends on values generated by job J_1.

Incorporating resource sharing and precedence constraints into the results presented in this dissertation would be a natural and important extension. However, both of these extensions can cause *priority inversions* — situations where a higher priority job is blocked while a lower priority job executes. The research presented in this dissertation assumes that a job can execute whenever it is active and has high enough priority. Priority inversions may violate this assumption. Therefore, the results in this dissertation do not account for either type of job dependence.

Non-preemptive systems

While preemption is a common assumption in real-time systems, there are situations when jobs may not be preempted. For example, communication devices often have nonpreemptable jobs. Removing the ability to preempt jobs may cause priority inversions. Moreover, non-preemptive systems can also suffer from *scheduling anomalies* whereby a feasible system may miss deadlines if one or more jobs are removed from the system or complete executing early. Accounting for priority inversions and scheduling anomalies is another important extension to the research presented in this dissertation.

6.2.3 Algorithm development

This dissertation considers several variations of the EDF scheduling algorithm. Even though there are a variety of reasons why EDF is a reasonable algorithm to consider, there are times when a job's urgency is not reflected by its deadline alone. For example, a high-utilization job may be more critical than a low-utilization job even if the high-utilization job has a later deadline. While EDF cannot account for such a situation when setting priorities, there may be other algorithms that can address these issues. Of course, any such algorithm would have to either be efficient or the schedule would have to be determined off-line. Whether the new algorithm is online or off-line, developing an algorithm tailored to take advantage of the different processing speeds available in uniform heterogeneous multiprocessors would be a valuable contribution to the field.

6.2.4 Combining models

While each of the areas of research mentioned above valid and challenging, combining two or more models could also result in valuable contributions. One possible combination of the challenges listed here would be to develop an algorithm for scheduling non-preemptive real-time systems with precedence constraints on variable voltage processors. Even if each of the three components — an algorithm for scheduling non-preemptive real-time systems on uniform heterogeneous multiprocessors, analysis of systems with precedence constraints, and strategy for effectively using variable voltage processors in the multiprocessor context — had already been developed combining the three contributions would be another significant development. Furthermore, the combination would very likely not be an incremental extension of any of the three original results. Real-time systems have very complex characteristics. Changing one aspect of the system can result in a very different problem. While the approaches used to find a schedulability test within each of the models would be helpful in determining a schedulability test in the combined model, developing the final schedulability test that takes all the constraints into consideration may involve developing new analysis techniques as well.

6.3 Summary

This dissertation presents several EDF-schedulability tests for real-time scheduling on uniform heterogeneous multiprocessors. We have seen that changing the migration strategy while leaving all other aspects of the system the same results in a very different problem which requires a completely different approach to analyze. Allowing processors to execute at different speeds gives the systems designer more flexibility to develop a system tailored specifically to the application at hand — particularly if the application has tasks with widely different utilization characteristics. Lifting further assumptions can provide even more flexibility for the system designer.

Just as varying migration strategies requires different analysis techniques, changing any of the parameters discussed in this section could have a major impact on the best approach for finding a good schedulability test. The goal of this dissertation and the goal of all future work is to extend the types of systems available to designers of real-time systems. As real-time systems become more common in industry and computing platforms become more diverse, any of the extensions mentioned here can provide important contributions to the real-time community.

INDEX

BIBLIOGRAPHY

[Bar02] Sanjoy K. Baruah. Robustness results concerning EDF scheduling upon uniform multiprocessors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 95–102, Vienna, Austria, June 2002. IEEE Computer Society Press.

[Bar04] Sanjoy K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[BC] Sanjoy K. Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *Journal of Embedded Computing*. Accepted for publication.

[BC03] Sanjoy K. Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In *Proceedings of the Euromicro Conference on Real-time Systems*, Porto, Portugal, 2003. IEEE Computer Society Press.

[BCPV96] Sanjoy K. Baruah, Neil Cohen, C. Greg Plaxton, and Don Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[BFG03] Sanjoy K. Baruah, Shelby Funk, and Joël Goossens. Robustness results concerning EDF scheduling upon uniform multiprocessors. *IEEE Transactions on Computers*, 52(9):1185–1195, 2003.

[BFS89] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31. ACM Press, 1989.

[BG03a] Sanjoy K. Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, 2003.

[BG03b] Sanjoy K. Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. In *Proceeding of the 23rd International Conference on Distributed Computing Systems*, Providence, RI, April 2003. IEEE Computer Society Press.

[CFH$^+$03] John Carpenter, Shelby Funk, Phil Holman, Anand Srinivasan, Jim Anderson, and Sanjoy K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y.-T Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. /, 2 edition, 2001.

[Del04] Dell, Inc. Dell PowerEdge 2600 Server. February 2004. http://www.dell.com/downloads/global/products/pedge/en/2600_specs.pdf.

[Der74] Michael Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[DM89] Michael Dertouzos and Aloysius K. Mok. Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[HL88] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 244–250, Huntsville, Alabama, December 1988. IEEE.

[HL94] Rhan Ha and Jane W.-S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, pages 162–171, Poznan, Poland, June 1994.

[HLS77] Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, 1977.

[HS86] Dorit S. Hochbaum and David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM*, 33:533–550, 1986.

[HS87] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162, January 1987.

[HS88] Dorit S. Hochbaum and David B. Shmoys. A polynomial time approximation scheme for scheduling on uniform processors using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, June 1988.

[Joh73] David S. Johnson. *Near-optimal Bin Packing Algorithms.* PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[KP95] Bala Kalyanasundaram and Kirk R. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 214–223, Los Alamitos, October 1995. IEEE Computer Society Press.

[KP98] Bala Kalyanasundaram and Kirk R. Pruhs. Maximizing job completions online. *Lecture Notes in Computer Science*, 1461:235–246, 1998.

[LGDG00] Jose Maria López, M. Garcia, José Luis Diaz, and Daniel F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 25–34, Stockholm, Sweden, June 2000. IEEE Computer Society Press.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM Press, 1986.

[LL73] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LL74] Jand W.-S. Liu and Chung Laung Liu. Bounds on scheduling algorithms for heterogeneous computing platforms. In *Proceedings of the IFIP Congress*, volume 30, pages 483–485, Stockholm, Sweden, 1974. North-Holland Publishing Company.

[LLG$^+$92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[LT99] Tak Wah Lam and Kar Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623 – 632, Baltimore, Maryland, 1999.

[McN59] R. McNaughton. Scheduling with deadlines and loss functions. *Machine Science*, 6(1):1–12, October 1959.

[Mok88] Aloysius K. Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proc. 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, Washington D.C., May 1988.

[PSTW97] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, 4–6 May 1997.

[SA] Anand Srinivasan and James Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*. Scheduled for publication.