END-TO-END SCHEDULING TO MEET DEADLINES
IN DISTRIBUTED SYSTEMS

BY

RICCARDO BETTATI

Diploma, Swiss Federal Institute of Technology, Zürich, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

1994

END-TO-END SCHEDULING TO MEET DEADLINES IN DISTRIBUTED SYSTEMS

Riccardo Bettati, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
Professor Jane W.S. Liu, Advisor

In a distributed real-time system or communication network, tasks may need to be executed on more than one processor. For time-critical tasks, the timing constraints are typically given as end-to-end release times and deadlines. This thesis describes algorithms to schedule a class of systems where all the tasks execute on different processors in turn in the same order. This end-to-end scheduling problem is known as the flow-shop problem. We present several cases where the problem is tractable and evaluate two heuristic algorithms for the NP-hard general case. We generalize the traditional flow-shop model in two directions. First, we present two algorithms for scheduling flow shops where tasks can be serviced more than once by some processors. Second, we describe a technique to schedule flow shops that consist of periodic tasks and to analyze their schedulability. We generalize this technique and describe how it can be used to schedule distributed systems that can not be modeled by flow shops. We then describe how to combine local or global resource access protocols and end-to-end scheduling. Finally, we show that by using end-to-end scheduling we can simplify resource access protocols and thus increase the utilization of resources.

To my parents, Aldo and Maria-Rosa Bettati

Acknowledgements

My deepest gratitude goes to Prof. Jane Liu, who five years ago took upon her to teach an intimidated student from overseas the tools of the trade of a researcher. She did this as a great teacher, a fabulous advisor, and an invaluable role model. Her continuous encouragement was crucial in helping over the periods of doubt and the occasionally arid portions of the work, and her enthusiasm and energy are always contagious.

I would like to thank the other members of my committee, Professors Andrew Chien, Kwei-Jay Lin, Dave Liu, Tony Ng, and Ben Wah, for having agreed to scrutinize and comment on the contents of my thesis.

A big "Thank you" goes to all the members of the Real-Time Systems Laboratory. This is a very special group, because joining it not only means acquiring new colleagues, but tying very dear friendships. First I would like to thank Don Gillies, with whom I had the honor to share the office. His encyclopedic memory has saved me many hours of literature search. Don perhaps remembers that I sill owe him the Magic Jacket. The third cohabitant of our office for many years was Carol Song, the person to ask for many little tools and tricks needed for experimentation, measurement, and writing. My gratitude goes also to the remaining members of the group: Infan Cheong, Zhong Deng, Wu Feng, Rhan Ha, Changwen Liu, Victor Lopez-Millan, Pilar Manzano, Joseph Ng, Luis Redondo, Arjun Shankar, Wei-Kuan Shih, Ami Silberman, Matthew Storch, Too-Seng Tia, Susan Vrbsky, and Xin Wang.

Over the years I learned to appreciate Champaign-Urbana as a very special place to make friends: I met Maher Hakim the very first day I arrived and cherish our friendship very much. Marc Najork and Sam Abassi have been very good friends over the years. Sam has always had a tender hart for starving graduate students. Let me acknowledge Hoichi Cheong, Nany Hasan, Ran Libeskind-Hadas, Ernst Mücke, Roman Waupotitsch, and many others, with whom I share many fond memories.

I would especially like to thank Bill and Joyce Krahling and their daughter Heidi for having welcomed me as part of their family. Joyce and Bill have become my American Mom and Dad, showing me many a precious side of living in America.

No spouse has ever given more support to a graduate student than my wife Andrea gave to me. She went very far out of her way (five continents, actually) to let me focus on my work, but managed to never make me miss her warmth and affection. One life time may not be enough for me to make up for all the hardship Andrea had to endure in these years.

This work was supported by the U.S. Navy ONR contract No. N00014-89-J-1181.

Table of Contents

Chapter

Appendix

List of Tables

List of Figures

Chapter 1
Introduction

In distributed real-time systems, tasks often are decomposed into chains of subtasks. In order to execute, each subtask requires the exclusive use of some resource, referred to here as a processor. We use the terms task and subtask to mean individual units of work that are allocated resources and then processed, that is, executed. For example, a subtask may be a granule of computation, a disk access, or a data transmission. Its execution may require a computer, a disk controller, or a data link, all modeled as processors. A task typically consists of many subtasks, which do some computations, access files, and transmit messages in turn. If such a task is time-critical, its timing constraints, derived naturally from its timing requirements, are typically given by its end-to-end release time and deadline. The end-to-end release time of a task is the time instant after which its first subtask can begin execution. Its end-to-end deadline is the time instant by which its last subtask must be completed. As long as these end-to-end constraints are met, when the individual subtasks are executed is not important.

1.1 Problem Statement

This thesis is concerned with scheduling tasks that execute in turn on different processors and have end-to-end release-time and deadline constraints. It focuses on the case where the system of tasks to be scheduled can be characterized as a *flow shop* or a variation of a flow shop. A flow shop models a distributed system or communication network in which tasks execute on different processors, devices, and communication links (all modeled as processors) in turn, following the same order.

An example of the type of systems studies here is a distributed multimedia system. This system consists of a video server that accesses and delivers video data, one or more hosts that receive and display the data, and a communication network that connects the display hosts to the video server. The video data consists of streams of frames. We model the access, delivery, and display of each stream of data frames as a task, which consists of three subtasks. The first subtask is the access and delivery of the data frame on the video server; the second subtask is the transmission of the data on the network from the video server to the requesting host; the third subtask is the acquisition and display on the host. These subtasks must be scheduled on the corresponding processors. The timing constraints of the task as whole are given by the maximum end-to-end delay between the request for and the display of a single frame of data. We study here how to schedule such a task to meet its deadline in the presence of similar tasks.

We note that each subtask in above example may need to be further decomposed into subtasks and, hence it is itself a task with end-to-end timing constraints. The task running on the video server for instance may consist of a subtask running on the disk controller and a second subtask that acquires buffer space and delivers the data to the underlying network. The communication network that delivers the data frames to the display hosts may be a multi-hop (generally a packet-switched) network. The tasks, modeling the transmission of real-time messages along a virtual circuit in such a network, consist of chains of subtasks, each subtask forwarding the message through one hop. The timing constraints for the transmission of the messages are end-to-end.

In a similar example, we consider a distributed system containing an input computer, an input link, a computation server, an output link, and an output computer. The input computer reads all sensors and preprocesses the sensor data. The processed sensor data is transmitted over the input link that connects the input computer to the computation server. The computation server computes the control law and generates commands. The commands are transmitted over its output link to the output computer, which performs D/A conversion and delivers the commands to the actuators. In our model, each tracker and controller is a task. Its release time and deadline arise from its required response. Each task consists of five subtasks: the first subtask is the processing of the sensor data on the input computer; the second subtask is the transmission of sensor data on the input link; and so on. These subtasks must be scheduled on the corresponding processors to meet the overall deadline. Again, how the individual subtasks are scheduled is not important as long as this overall deadline is met.

End-to-end timing constraints are not necessarily limited to distributed systems. In many instances we can think of a task accessing a non-sharable resource as temporarily executing on that resource. After the task releases the resource, it returns to execute on the processor. Such a task can be modeled as a chain of three subtasks with end-to-end timing constraints. This example shows how variations of flow shops can be used to model certain forms of resource access, such as the integrated processor and I/O scheduling described in Sha et al. [46] or the access to global resources in a multiprocessor described by Rajkumar et al. [42].

A system may contain many classes of tasks. Tasks in each class execute on different processors in the same order, but tasks in different classes execute on different processors in different orders. (In queuing modeling terms, the system contains many task-routing chains.) One straightforward way to schedule multiple classes of tasks is to partition the system resources and assign them statically to task classes. This allows the tasks in each class to be scheduled according to a scheduling algorithm suited for the class. The effect of this partitioning is that we now have as many virtual processors as there are classes of tasks sharing each processor. A distributed system that contains N classes of tasks and uses such a static resource partition and allocation strategy can be modeled as a system containing N flow shops. Static partitioning of resources may lead to low utilization. Unfortunately, traditional techniques to increase utilization, such as statistical multiplexing or dynamic allocation of the resources, are not suited in hard real-time systems, because the effect of conflicts between subtasks of different classes is often unpredictable.

1.2 Summary of Results

In this thesis we first focus on the problem of scheduling tasks in traditional flow shops to meet their end-to-end timing constraints. The problem of scheduling tasks in a flow shop to meet deadlines is $\mathcal{NP}$-hard, except for a few special cases [11, 25]. We have developed optimal efficient algorithms for scheduling sets of tasks that have identical processing times on all processors, called *identical-length* task sets, and sets of tasks that have identical processing times on each of the processors but have different processing times on different processors, called *homogeneous* task sets. An algorithm is *optimal* if it always finds a *feasible* schedule, that is, a schedule in which all tasks meet their deadlines whenever such a schedule exists. These optimal algorithms are used as the basis for a heuristic algorithm, called Algorithm $\mathcal{H}$, for scheduling tasks with arbitrary processing times. The results of our evaluation show that Algorithm $\mathcal{H}$ performs well for tasks that have similar processing times. When the difference between processing times of tasks becomes large (for instance when there are long tasks and short tasks), the performance of this heuristic degrades.

In order to complement Algorithm $\mathcal{H}$, we developed an algorithm for scheduling tasks belonging to two different task classes; tasks in each class form an identical-length task set. The processing times of tasks in the sets are τ and $p\tau$, where p is a positive integer. This algorithm is optimal when the release times are integer multiples of τ. We extended this optimal algorithm into a second heuristic algorithm, called Algorithm $\mathcal{HPCC}$, to schedule tasks with arbitrary processing times. As our evaluations show, Algorithm $\mathcal{HPCC}$ performs much better than Algorithm $\mathcal{H}$ when tasks have widely different processing times.

We also consider two variations of the traditional flow-shop model, called *flow shop with recurrence* and *periodic flow-shop*. In a flow shop with recurrence each task executes more than once on one or more processors. A flow shop with recurrence models a system that has limited resources and, hence, does not have a dedicated processor for every function. As an example, suppose that the three computers in the control system mentioned earlier are connected by a bus, not by two dedicated links. The system can no longer be modeled as a traditional flow shop. However, we can model the bus as a processor and the system as a flow shop with recurrence. Each task executes first on the input computer, then on the bus, on the computation server, on the bus again, and finally on the output computer. Similarly, a task that accesses a serially reusable resource can be thought of as first executing on a processor, then temporarily executing on the resource, and eventually migrating back to the processor. A system containing such tasks can also be modeled as a flow shop with recurrence.

In analyzing flow shops with recurrence, we focused our attention on a simple recurrence pattern, namely where one processor, or one sequence of processors, is visited more than once. We developed a $\mathcal{O}(n\ log\ n)$ algorithm to schedule tasks in flow shops with such recurrence patterns (where n is the number of tasks in the flow shop). When tasks have identical processing times and release times, this algorithm is optimal. We also developed a $\mathcal{O}(n^3)$ algorithm to schedule tasks with individual release times in flow shops with simple recurrence patterns. The $\mathcal{O}(n^3)$ algorithm is optimal as long as the release times and deadlines are multiples of $q\tau$, where τ is the processing time of the subtasks and q is the number of subtasks between the first and the second visit of tasks to a revisited processor.

In a periodic flow shop each task is a periodic sequence of requests for the same computation. Hence, a sequence of requests for a periodic computation in a traditional flow shop is represented as a single task in a periodic flow shop. Several analytical techniques exist to determine whether given algorithms can feasibly schedule periodic tasks on a single processor [27, 31]. *Schedulability analysis* is the process of determining whether a given algorithm always generates a feasible schedule for a given system of tasks under all conditions. We extended the techniques for single-processor schedulability analysis to end-to-end schedulability analysis of systems where tasks execute on more than one processor. We developed a technique that makes use of results in single-processor schedulability analysis to determine whether end-to-end timing constraints are met while the dependencies between periodic subtasks are preserved. This technique, called *phase modification*, proved to be very flexible. We show how it can be used in combination with arbitrary fixed-priority scheduling algorithms, even algorithms that assign different priorities to different subtasks of the same task. Moreover, phase modification is not restricted to systems that can be modeled by flow shops. We showed how this technique can be applied to collections of flow shops that share one or more processors, or even to systems with arbitrary dependencies between periodic subtasks on different processors. Although phase modification in its basic form is based on the assumption that the executions of the tasks on the different processors are synchronized according to a common clock, we showed that this assumption can be relaxed. We present two techniques to account for the lack of a common clock. In the first technique we simply separate the executions of subtasks so that clock drift can not cause dependencies to be violated. In the second technique we use results in the scheduling of aperiodic tasks to allow for asynchronous execution of the subtasks.

One area that has attracted a lot of attention in the schedulability analysis for single-processor systems are real-time resource access protocols [3, 42, 47]. We could show that these protocols can be naturally included in the phase modification technique to analyze systems with end-to-end timing constraints and access to resources, both local and global. Models of access to global resources typically assume that tasks temporarily execute on a special processor when they are holding a global resource. In such systems, each access to a global resource can be viewed as a sequence of three subtasks with end-to-end timing constraints. The task first executes on its processor until it accesses the global resource. When it is granted the resource, it executes on another processor, and returns to the first processor after it releases the resource.

This decomposition allows us to replace the expensive global-resource access protocols with access protocols to local resources in combination with end-to-end schedulability analysis. We could show that this typically allows for higher resource utilizations with the tasks still meeting the timing constraints.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 describes the underlying model of systems with end-to-end timing constraints. We describe the traditional flow-shop model, as well as the flow shop with recurrence and periodic flow-shop models. Chapter 3 summarizes existing work in end-to-end scheduling and related areas.

In Chapter 4 we present algorithms for scheduling in traditional flow shops to meet end-to-end release times and deadlines. In particular, we describe Algorithm $\mathcal{H}$ and Algorithm $\mathcal{HPCC}$, the two heuristic algorithms for scheduling arbitrary task sets in traditional flow shops, and evaluate their performance.

In Chapter 5 we present two algorithms, Algorithm $\mathcal{R}$ and Algorithm $\mathcal{RR}$, to schedule identical-length task sets on flow shops with recurrence. Both algorithms work for recurrence patterns in which one processor, or a sequence of processors, is visited more than once. We show that Algorithm $\mathcal{R}$ is optimal for such task sets with identical release times and arbitrary deadlines. Algorithm $\mathcal{RR}$ is optimal for task sets in which the release times and the deadlines are integer multiples of the distance between the first and the second visit of tasks to the revisited processor.

In Chapter 6 we return to the periodic flow-shop model. We present the *phase modification* technique to assign intermediate deadlines to subtasks in periodic flow shops and to determine their schedulability. We introduce the technique in combination with the rate-monotonic scheduling algorithm, but show that it can be used in conjunction with arbitrary fixed-priority scheduling algorithms. We describe how the techniques developed in this chapter can be extended to handle systems without a common clock or systems that are more general than ordinary flow shops. We show in fact how the phase-modification method can be used to schedule of a wide class of end-to-end systems.

Chapter 7 describes how to incorporate resource access protocols for local and global resources into the phase-modification scheme. We describe how the phase modification technique can be combined with local-resource access protocols to handle systems with global resources and compare this approach against the multiprocessor priority-ceiling protocol [42], a well known real-time multiprocessor-synchronization protocol.

Chapter 8 gives a summary and an overview of open questions, and concludes the thesis.

Chapter 2
The Flow-Shop Model

In this chapter we describe a general model of distributed systems with end-to-end constraints: the flow shop. This model is used throughout this thesis. The notation and the terminology used in the successive chapters are introduced here.

2.1 The Flow Shop

In a traditional flow shop, there are m different processors $P_1, P_2, \cdots, P_m$. We are given a set $\mathcal{T}$ of n tasks $T_1, T_2, \cdots, T_n$ that execute on the m processors. Specifically, each task T_i consists of m subtasks $T_{i1}, T_{i2}, \cdots, T_{im}$. These subtasks execute in order; first T_{i1} on processor P_1, then T_{i2} on processor P_2, and so on. In other words, every task passes through the processors in the same order. Let τ_{ij} denote the time required for the subtask T_{ij} to complete its execution on processor P_j; τ_{ij} is the *processing time* of the subtask T_{ij}. Let τ_i denote the sum of the processing times of all the subtasks of the task T_i. We refer to τ_i as the *total processing time* of the task T_i. Each task T_i is ready for execution at or after its *release time* r_i and must be completed by its *deadline* d_i. For sake of simplicity, we will occasionally refer to the totality of release times, deadlines, and processing times as the *task parameters*.

In the following we will make use of the *effective deadlines* of subtasks. The effective deadline d_{ij} of the subtask T_{ij} is the point in time by which the execution of the subtask T_{ij} must be completed to allow the later subtasks, and the task T_i, as a whole, to complete by the deadline d_i. d_{ij} is computed as follows:

$$d_{ij} = d_i - \sum_{k=j+1}^{m} \tau_{ik}. \tag{2.1}$$

Similarly, we define the *effective release time* r_{ij} of a subtask T_{ij} to be the earliest point in time at which the subtask can be scheduled. Since T_{ij} cannot be scheduled until earlier subtasks are completed, r_{ij} is given by

$$r_{ij} = r_i + \sum_{k=1}^{j-1} \tau_{ik}. \tag{2.2}$$

For scheduling purposes we distinguish two classes of tasks, depending on whether or not the tasks can be temporarily interrupted during their execution. A task is *preemptable* if its execution can be interrupted and, at a later point in time, resumed from where it was interrupted. On the other hand, a *non-preemptable* task cannot be interrupted. Schedulers that make use of the fact that tasks are preemptable are called *preemptive* schedulers. For many scheduling problems, there are no polynomial-time optimal schedulers when tasks are non-preemptable. But efficient optimal preemptive schedulers are possible (e.g. see McNaughton's rule [36]). Unfortunately, with arbitrary task parameters, the traditional flow-shop problem is $\mathcal{NP}$-hard, even where the subtasks are preemptable, as the following theorem shows:

Theorem 1. The flow-shop problem on $m > 2$ processors with arbitrary task parameters is $\mathcal{NP}$-complete.

Proof. We consider two cases: when preemption is not allowed and when preemption is allowed. To show that this problem is $\mathcal{NP}$-complete when preemption is not allowed, we restrict it to the case of one processor with arbitrary job parameters by letting the processing times of subtasks on all but one processor be zero. The problem of scheduling non-preemptable tasks with arbitrary job parameters to meet individual deadlines is known to be $\mathcal{NP}$-complete [9, 10]. Similarly, when preemption is allowed, we restrict the flow-shop problem to the case of identical deadlines. The preemptive flow-shop problem to meet an overall deadline is known to be $\mathcal{NP}$-complete [14]. □

Two special cases of the flow-shop scheduling problem are tractable, however. The first is the case of *identical-length task sets*, where the processing times τ_{ij} of all subtasks on all processors are equal to τ. In the second case the processing times τ_{ij} of all subtasks are identical for any one processor, but may vary between different processors. In other words, all the subtasks T_{ip} on any processor P_p have the same processing time τ_p, but subtasks T_{ip} and T_{iq} on processors P_p and P_q may have different processing times, that is, $\tau_p \neq \tau_q$ when $p \neq q$. We call task sets of this sort *homogeneous task sets*. In Chapter 4 we will describe algorithms to optimally schedule identical-length and homogeneous task sets. We will also describe a heuristic, Algorithm $\mathcal{H}$, built on these algorithms to schedule task sets with arbitrary task parameters.

2.2 The Flow Shop with Recurrence

In the more general flow-shop-with-recurrence model, each task T_i has k subtasks, and $k > m$. Without loss of generality, we let the subtasks be executed in the order $T_{i1}, T_{i2}, \cdots, T_{ik}$ for all tasks T_i, that is, T_{i1} followed by T_{i2}, followed by T_{i3}, and so on. We specify the processors on which the subtasks execute by a sequence $V = (v_1, v_2, \cdots, v_k)$ of integers, where v_j is one of the integers in the set $\{1, 2, \cdots, m\}$. v_j being l means that the subtasks T_{ij} (for all i) are executed on processor P_l. For example, suppose that we have a set of tasks each of which has 5 subtasks, and they are to be executed on 4 processors. The sequence $V =$ (1, 2, 3, 2, 4) means that all tasks first execute on P_1, then on P_2, P_3, again on P_2, and then P_4, in this order. We call this sequence the *visit sequence* of the tasks. If an integer l appears more than once in the visit sequence, the corresponding processor P_l is a *reused processor.* In this example P_2 is reused, and each task visits it twice. This flow shop with recurrence models the distributed control system described in Chapter 1: P_2 models the bus that is used as both the input link and the output link. The traditional flow-shop model is therefore a special case of the flow-shop-with-recurrence model in which the visit sequence is $(1, 2, \cdots, m)$, and every processors is visited only once.

Any visit sequence can be represented by a graph, called a *visit graph.* A visit graph $\mathcal{G}$ is a directed graph whose vertices P_i's represent the processors in the system. There is a directed edge e_{ij} from P_i to P_j with label a if and only if in the visit sequence $V = (v_1, v_2, \cdots, v_a, v_{a+1}, \cdots, v_k)$ $v_a = i$ and $v_{a+1} = j$. A visit sequence can therefore be represented by a path with increasing edge labels in the visit graph. The visit graph for the distributed control system with visit sequence $V = (1, 2, 3, 2, 4)$ is shown in Figure 2.1.

We confine our attention here to a class of visit sequences that contain simple recurrence patterns: the recurrence pattern in the visit sequence is a *loop.* The notion of loops becomes intuitively clear when we look at the representation of such a loop in the visit graph. In the example shown in Figure 2.2, the labeled path that represents the visit sequence contains a loop. The sub-sequence $(2, 3)$ in $(1, 2, 3, 4, 5, 2, 3, 6)$ occurs twice and therefore makes the sequence $(4, 5, 2, 3)$ following the first occurrence of $(2, 3)$ a loop. P_2 and P_3 are the reused processors in this example. Loops can be *simple* or *compound.* A simple loop contains no subloop. The visit sequence in Figure 2.2 contains a simple loop. A compound loop contains other loops as

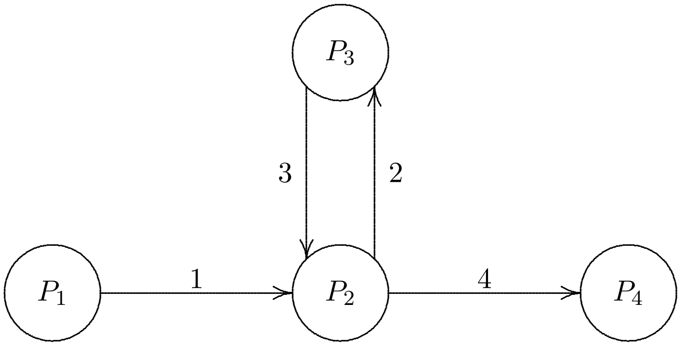


Figure 2.1: Visit Graph for Visit Sequence $V = (1, 2, 3, 2, 4)$.

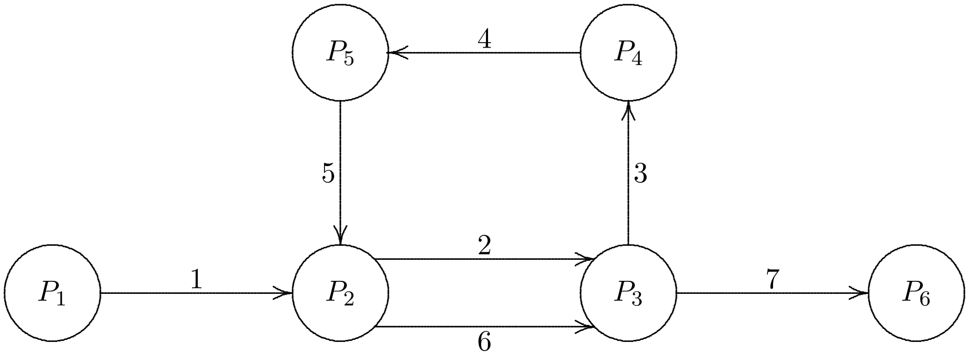


Figure 2.2: Visit Graph for Visit Sequence $V = (1, 2, 3, 4, 5, 2, 3, 6)$.

parts of the path forming the loop. The *length* of a loop is the number of vertices in the visit graph that are on the loop. The loop in Figure 2.2 therefore has length 4. Loops can be used to model systems where a specific processor (or a sequence of processors) is used before and after a certain service is attained. An example is a database that is queried before and updated after a specific operation.

The flow-shop-with-recurrence problem on $m > 2$ processors with arbitrary task parameters is $\mathcal{NP}$-hard. This follows as a corollary from Theorem 1 since the traditional flow shop is a special case of the flow shop with recurrence.

2.3 The Periodic Flow Shop

The periodic flow-shop model is a generalization of both the traditional flow-shop model and the traditional periodic-job model [26, 31, 41]. As in the traditional periodic-job model, the periodic *job system* $\mathcal{J}$ to be scheduled in a flow shop consists of n periodic jobs. Each job consists of a periodic sequence of requests for the same computation. In our previous terms, each request is a *task*. In an m-processor flow shop, each task consists of m subtasks that are to be executed on the m processors in turn following the same order. The processing time of the subtask on processor P_j of each task in the job J_i is τ_{ij}. The *period* p_i of a job J_i in $\mathcal{J}$ is the length of the time interval between the ready times of two consecutive tasks in the job. The ready time of the task in every period is the beginning of the period. The deadline of a task is some time D_i after its ready time, for some constant $D_i \leq m\,p_i$. The *utilization factor* u_{ij} of subjob J_{ij} on processor P_j is $u_{ij} = \tau_{ij}/p_i$, the fraction of time J_{ij} keeps processor P_j busy. The *total utilization factor* u_j on processor P_j is the sum of the utilization factors of the subjobs running on that processor, that is, $u_j = \sum_{i=1}^{n} \tau_{ij}/p_i$.

In the following, by flow shop we mean specifically non-periodic flow shop without recurrence. By the flow-shop (or the flow-shop-with-recurrence) problem we mean the problem of scheduling in a flow shop (or a flow shop with recurrence) to meet deadlines.

Chapter 3
Related Work

The end-to-end scheduling problems addressed in this thesis are closely related to problems of machine-scheduling, pipeline scheduling, and real-time communication. Here, we summarize the similarities and differences in the approaches and objectives of our work and earlier work on the related problems.

3.1 Classical Machine-Scheduling Theory

Past efforts in flow-shop scheduling have focused on the minimization of completion time, that is, the total time required to complete a given set of tasks [9, 12, 25, 53]. Johnson [9] showed that tasks in a two-processor flow shop can be scheduled to minimize completion time in $\mathcal{O}(n \log n)$ time. Beyond the two-processor flow shop, however, almost every flow-shop scheduling problem is $\mathcal{NP}$-complete. For example, the general problem of scheduling to minimize completion time on three processors is strictly $\mathcal{NP}$-hard [12]. Consequently, most studies of flow-shop problems were concerned with restrictions of the problem that make it tractable.

Other efforts focused on heuristic algorithms and enumerative methods that yield optimal and suboptimal schedules in reasonable amounts of time. Hoitomt et al. [21] use integer-programming techniques and Lagrangian relaxation to schedule tasks with deadlines to minimize the weighted tardiness of tasks in job shops where precedence constraints are simple. (In a *job shop*, tasks execute on different processors in arbitrary orders.)

Much research has focused on enumerative methods for permutation flow shops [18, 35]. In a *permutation flow shop* the tasks execute in the same order on all the processors. In other words, if the execution of a task T_i precedes the execution of another task T_j on one processor, T_i must precede T_j on all the processors. A schedule for which this condition holds is called a *permutation schedule*. In contrast, in a non-permutation schedule the sequence of tasks executed on different processors may vary.

Grabowski et al. [17] extend their enumerative approach described in [16] to schedule flow shops with arbitrary release times and deadlines to minimize lateness. In *no-wait flow shops* the tasks are not buffered between processors. In other words, when a task is started in a no-wait flow shop, it has to run through all processors until it terminates. The problem of scheduling no-wait flow shops is reviewed in [15].

The general problem of scheduling to meet deadlines on identical multiprocessor systems is also $\mathcal{NP}$-hard [11, 25]. However, polynomial algorithms for optimally scheduling tasks with identical processing times on one or two processors exist [10, 13]. Our algorithms in Chapter 4 and Chapter 5 make use of one of them.

3.2 Pipeline Scheduling

Lawler et al. [24] identify three differences between the pipeline-scheduling and the flow-shop scheduling problems:

(1) As opposed to flow-shops, no buffering of subtasks is allowed between stages in pipelines. We note that the buffering of subtasks allows for non-permutation schedules on flow shops. Consequently the flow-shop scheduling problem is more difficult.

(2) When two tasks are dependent, it is usually assumed in the flow-shop model that the last subtask of the predecessor task is the immediate predecessor of the first subtask of the successor task. In the pipeline model, arbitrary dependencies between internal stages are allowed.

(3) In flow-shop scheduling, there is a single sequence of processors, whereas pipeline scheduling has to deal with multiple, perhaps specialized, pipelines. In this way, the pipeline scheduling problem is more complex.

The main objective in traditional pipeline scheduling is the generation of schedules that maximize throughput. Polynomial-time algorithms are known for scheduling pipelines of length two with arbitrary precedence constraints and pipelines of length k with precedence constraints that are trees [4]. Most generalizations of these two cases are $\mathcal{NP}$-complete. For example, the general pipeline-scheduling problem to minimize completion time on a single pipeline of length k for arbitrary dependency graphs is $\mathcal{NP}$-complete [29].

Palem and Simons [39] describe a priority-driven heuristic algorithm to schedule pipelines with timing constraints. Their pipeline model is generalized to consider inter-instructional latencies. Dependencies can be weighted to represent the minimal delay between the termination of one subtask and the execution of the descendent subtask. Inter-instructional latencies are similar to minimum-distance constraints described by Han in [19, 20]. According to the heuristic in [39], the priority assignments are based on weighted path lengths in the dependency graph. Timing constraints are then added in the form of additional such latencies.

3.3 Real-Time Communication

In a real-time communication system [23], there is a time constraint on the delay of each message, that is, the length of the time interval between the time instants when the source generates the message and when the destination receives it. The message is discarded when its delay exceeds this constraint. A typical application is packetized audio or video transmission, where the signal is digitized and packetized at the sender and reconstructed at the destination. The delay of the packets is constrained to be below a limit. Packets that do not arrive at the destination within the maximally allowed delay are discarded. The encoding and packeting schemes are designed so that the occasionally loss of packets has a minimal effect on the quality of the received audio or video. Recently, a considerable amount of effort has been directed toward how to ensure messages with time constraints are delivered over multiple-access networks [1, 23, 38, 55] in time.

When the underlying network has a point-to-point interconnection structure, such as in wide-area networks [7], it is more complicated to guarantee timely message delivery. Since only the end-to-end delay is constrained to be under some limit, there are more choices in scheduling message transmissions through the multiple links in the network. The increase in the number of choices results in either more complicated algorithms or heuristics with uncertain performance. Kandlur et al. [22] describe a method to guarantee time-constrained, sequenced message delivery over unidirectional virtual connections, called (real-time) *channels*. A channel consists of a sequence of *communication links*. The communication links in each channel are selected at channel-establishment time. During channel-establishment time, it is necessary to ensure that the new channel does not affect the guaranteed delivery times of existing channels.

Moreover, on each link along the channel, an upper bound on the message delay, called *worst-case message delay*, is determined by a schedulability analysis of the channels on that link. The channel can be established if the sum of the worst-case delays on all the links along the channel is less than the end-to-end delay allowed for the entire channel. The run-time scheduling on each link is done according to a variation of the effective-earliest-deadline-first algorithm that is described in Chapter 4. The worst-case message delays along the links in a channel are computed at channel-establishment time and are used to determine the effective deadlines of message transmissions on each link.

3.4 Task Assignments

Burns et al. [5] discuss the problem of allocating and scheduling periodic tasks, called *transactions*, that consist of chains of subtasks (called *subtransactions*) in DIA (Data Interaction Architecture) [49]. Each transaction consist of a collection of precedence related processes that are modeled as subtasks in our periodic flow-shop model. The processors in DIA are linked via dual-port memories. The network is not fully connected and hence the placement of tasks and the routing of messages cannot be considered separately. Burns et al. propose a *simulated annealing* approach that allocates subjobs to processors and assigns priorities to subjobs to satisfy end-to-end timing constraints. In Chapter 6 we prove that the problem of assigning priorities to subjobs to meet deadlines is indeed $\mathcal{NP}$-hard, even for a single processor. We have simple and efficient heuristics that can solve this problem effectively without having to use expensive optimization techniques.

Chapter 4

End-To-End Scheduling of Flow Shops

The traditional flow-shop model is the basic model used in this thesis. In this chapter, we present two special cases of flow shops for which polynomial-time scheduling algorithms exist. We describe how we extend these algorithms into heuristic algorithms to schedule flow shops with arbitrary task parameters.

4.1 Scheduling Identical-Length Task Sets on Flow Shops

Sometimes, tasks have regular structures. In the simplest case, the processing times τ_{ij} of all subtasks are identical for all i and j, that is, the task set is of identical length. As an example, suppose that we are to schedule a set of 64 kbps voice data streams over a n-hop virtual circuit. The virtual circuit is assigned the same bandwidth on all the links. This system can be modeled as a flow shop where the subtasks form an identical-length task set with processing times τ.

Sometimes, release times and deadlines are multiples of τ. In this case, we can simply use the classical *earliest-effective-deadline-first* (EEDF) algorithm to optimally schedule all tasks [9]. Again, an algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists. According to the EEDF algorithm, the subtasks T_{ij} on each processor P_j are scheduled nonpreemptively in a *priority-driven* manner. An algorithm is priority-driven if it never leaves any processor idle when there are tasks ready to be executed. According to the EEDF algorithm, priorities are assigned to subtasks on P_j according to their effective deadlines: the earlier the effective deadline, the higher the priority. In other words, the subtask T_{ij} is assigned a higher priority than T_{hj} if $d_{ij} < d_{hj}$. The scheduling decisions are made on the first processor P_1 whenever it is free; the subtask with the highest priority among all ready subtasks is executed until completion. The regular structure of the task set allows us to propagate these scheduling decisions on to the subsequent processors; whenever the subtask T_{ij} completes on P_j, $T_{i(j+1)}$ starts on P_{j+1}.

The scheduling decision on the first processor is slightly more complicated if release times and deadlines are arbitrary rational numbers, that is, not multiples of τ. Garey et al. [13] introduced the concept of *forbidden regions* during which tasks are not allowed to start execution. Their algorithm postpones the release times of selected tasks. This is done to adequately insert the necessary idle times to make an EEDF schedule optimal. We call the release times generated from the given effective release times by the above mentioned algorithm the *modified release times.* In our subsequent discussion, by release times, we mean modified release times. By the earliest-effective-deadline-first algorithm, we mean the earliest-effective-deadline-first algorithm using the modified release times as input parameters rather than the given effective release times.

Theorem 2. For nonpreemptive flow-shop scheduling of tasks that have arbitrary release times and deadlines, and whose subtasks have identical processing times τ on all m processors, the EEDF algorithm never fails to find a feasible schedule whenever such schedules exist. In other words, the EEDF algorithm is optimal.

Proof. We schedule all the first tasks T_{i1} on the first processor P_1 according to the EEDF algorithm. If the resultant schedule $\mathcal{S}_1$ of the subtasks $T_{11}, T_{12}, \cdots, T_{n1}$ on P_1 is feasible, then a feasible schedule of the tasks $T_1, T_2, \cdots, T_n$ on the m processors can be constructed from $\mathcal{S}_1$ as follows: (1) On P_j the subtasks $T_{1j}, T_{2j}, \cdots, T_{nj}$ are executed in the same order as $T_{11}, T_{21}, \cdots, T_{n1}$ on P_1. The resultant schedule is thus a permutation schedule. (2) The execution of T_{ij} on P_j begins as soon as $T_{i(j-1)}$ is completed for $j = 2, 3, \cdots, m$. Since the processing times of all subtasks are identical, this construction is always possible. Moreover, since T_{i1} completes by its effective deadline d_{i1} given by Equation (2.1), T_i completes by its deadline d_i.

Suppose that the EEDF algorithm fails to find a feasible schedule of $T_{11}, T_{21}, \cdots, T_{n1}$ on P_1. Because of the optimality of the EEDF algorithm for scheduling tasks with identical processing times on one processor [9, 13], we can conclude that there is no feasible schedule of $T_{11}, T_{21}, \cdots, T_{n1}$, meeting their effective deadlines $d_{11}, d_{21}, \cdots, d_{n1}$. Therefore there exists no feasible schedule of $T_1, T_2, \cdots, T_n$. □

Corollary 1. For flow-shop scheduling of identical-length tasks sets that have zero release times and arbitrary deadlines on m processors, the EEDF algorithm is optimal.

Proof. This corollary follows from Theorem 2 and the following fact: when the release times of all tasks are identical, the schedules of $T_{11}, T_{21}, \cdots, T_{n1}$ on P_1 produced by the preemptive and nonpreemptive EEDF algorithms are the same. Hence the nonpreemptive EEDF algorithm is optimal among all algorithms. □

4.2 Scheduling Homogeneous Task Sets on Flow Shops

The assumption that the task sets have identical processing times is a very restrictive one. Taking a step toward the complete removal of this assumption, we now consider a more general class of flow shops in which the subtasks T_{ij} on each processor P_j have identical processing times τ_j, but subtasks T_{ij} and T_{ih} on different processors may have different processing times, that is, $\tau_j \neq \tau_h$ in general. This class of flow shops is called flow shops with *homogeneous* task sets. An example of flow shops with homogeneous tasks arises in scheduling a set of identical communication requests over a n-hop virtual circuit when the circuit does not have the same bandwidth on all the links. We can use *Algorithm $\mathcal{A}$*, described in Figure 4.1, to schedule such a set of tasks. According to this algorithm, the scheduling decisions are made on a bottleneck processor: A processor P_b is a *bottleneck processor* if the processing time τ_b of the subtasks on it is the longest among all processors. The decisions are then propagated to other processors.

The following theorem states the optimality of Algorithm $\mathcal{A}$.

Theorem 3. For nonpreemptive flow-shop scheduling of tasks that have arbitrary release times and deadlines and whose subtasks on the processor P_j have processing times τ_j, Algorithm $\mathcal{A}$ never fails to find a feasible schedule whenever such a schedule exists.

Proof. If the resultant schedule $\mathcal{S}_b$ on the bottleneck processor P_b is not feasible, we can conclude that no feasible schedule of $\{T_{ib}\}$ exists because of the optimality of the EEDF algorithm [13]. On the other hand, if $\mathcal{S}_b$ is a feasible schedule of $\{T_{ib}\}$, the propagation method described in Step 3 can always generate a feasible schedule $\mathcal{S}$ of the set $\mathcal{T}$. This claim is true because the start time of the next subtask scheduled after T_{ib} on P_b in $\mathcal{S}_b$ is no earlier than $t_{ib} + \tau_b$, where t_{ib} is the start time of T_{ib}. Hence, it is always possible to schedule $T_{i(b+1)}$ on P_{b+1} immediately after T_{ib} completes, $T_{i(b+2)}$ on P_{b+2} immediately after $T_{i(b+1)}$ completes on P_{b+1}, and so on. Since T_{ib} completes by d_{ib}, T_{ir} completes by d_{ir} for all $r \geq b$. Similarly, the subtask that starts before

Algorithm $\mathcal{A}$:

Input: Task parameters r_{ij}, d_{ij} and τ_j of $\mathcal{T}$.

Output: A feasible schedule $\mathcal{S}$ of $\mathcal{T}$ or the conclusion that feasible schedules of $\mathcal{T}$ do not exist.

Step 1: Determine the processor P_b where $\tau_b \geq \tau_j$ for all $j = 1, 2, \cdots, m$. If there are two or more such processors, choose one arbitrarily. P_b is the *bottleneck processor*.

Step 2: Schedule the subtasks on the bottleneck processor P_b according to the EEDF algorithm. If the resultant schedule $\mathcal{S}_b$ is not a feasible schedule, stop; no feasible schedule exists. Otherwise, if $\mathcal{S}_b$ is a feasible schedule of $\{T_{ib}\}$, let t_{ib} be the start time of T_{ib} in $\mathcal{S}_b$; do Step 3.

Step 3: Propagate the schedule $\mathcal{S}_b$ onto the remaining processors as follows: Schedule $T_{i(b+1)}$ on P_{b+1} immediately after T_{ib} completes, $T_{i(b+2)}$ on P_{b+2} immediately after $T_{i(b+1)}$ completes, and so on until T_{im} is scheduled. For any processor P_r, where $r < b$, we schedule T_{ir} on P_r so that its execution starts at time $t_{ib} - \sum_{s=r}^{b-1} \tau_s$, for $r = 1, 2, \cdots, b-1$.

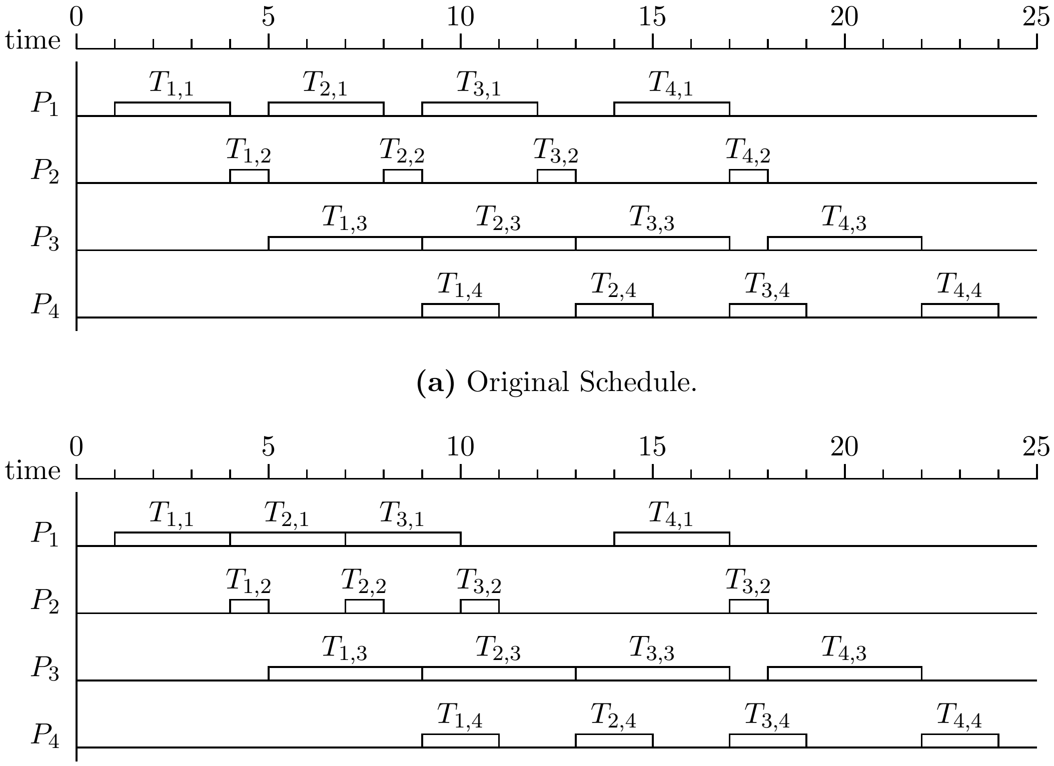
Figure 4.1: Algorithm $\mathcal{A}$.

T_{ib} on P_b in $\mathcal{S}_b$ starts no later than $t_{ib} - \tau_b$. Again, since $\tau_b \geq \tau_j$ for $j \neq b$, the construction in Step 3 is always possible. □

An example illustrating Algorithm $\mathcal{A}$ is shown in Figure 4.2. We have here 4 tasks with task parameters listed in Table 4.1. The bottleneck processor is the one on which tasks have the longest processing times. In this example, it is P_3, or $b = 3$. We note that the schedule $\mathcal{S}$ generated by Algorithm $\mathcal{A}$ is not an EEDF schedule. It is not priority-driven; processors $P_1, P_2, \cdots, P_{b-1}$ sometimes idle when there are subtasks ready to be executed. (This is not the case for the subtasks on $P_{b+1}, P_{b+2}, \cdots, P_m$, since the executions of the subtasks $T_{i(b+1)}, T_{i(b+2)}, \cdots, T_{im}$ begin immediately after they become ready following the termination of the respective preceding subtasks.) The intervals of idle time in $\mathcal{S}$ on the processors preceding P_b can easily be eliminated, however. The schedule shown in Figure 4.2b is constructed from $\mathcal{S}$ in Figure 4.2a by eliminating the intentional idle intervals on P_1 and P_2. The resultant schedule is an EEDF schedule.

Tasks	r_i	d_i	τ_{i1}	τ_{i2}	τ_{i3}	τ_{i4}
T_1	1	10	3	1	4	2
T_2	1	13	3	1	4	2
T_3	5	30	3	1	4	2
T_4	14	26	3	1	4	2

Table 4.1: An Example of a Homogeneous Task Set.



(a) Original Schedule.

(b) Same Schedule with Intentional Idle Times Removed.

Figure 4.2: Schedule Generated by Algorithm $\mathcal{A}$.

4.3 Scheduling Arbitrary Task Sets on Flow Shops

In real-world applications the processing times of tasks on individual processors are often not the same. Some subtasks on a processor may have longer processing times than some other subtasks on the same processors. The heuristic algorithm described in Figure 4.3 assumes that subtasks have arbitrary processing times. It is called *Algorithm* $\mathcal{H}$ and has 5 steps. After Step 1

Algorithm $\mathcal{H}$:

Input: Task parameters r_i, d_i and τ_{ij} of $\mathcal{T}$.

Output: A feasible schedule of $\mathcal{T}$, or the conclusion that it failed to find one.

Step 1: Determine the effective release times r_{ij} and effective deadlines d_{ij} of all subtasks.

Step 2: On each processor P_j, determine the subtask $T_{max,j}$ with the longest processing time $\tau_{max,j}$ among all subtasks T_{ij} on P_j.

Step 3: On each processor P_j, inflate all the subtasks by making their processing times equal to $\tau_{max,j}$. In other words, each inflated subtask T_{ij} consists of a busy segment with processing time τ_{ij} and an idle segment with processing time $\tau_{max,j} - \tau_{ij}$. Now, the inflated subtasks form a homogeneous task set $\mathcal{T}_{inf}$.

Step 4: Schedule the inflated task set $\mathcal{T}_{inf}$ using Algorithm $\mathcal{A}$.

Step 5: Compact the schedule by eliminating as much as possible the lengths of idle periods that were introduced when we inflated the subtasks and when we propagated the schedule. The compaction process can be done using Algorithm $\mathcal{C}$ described later in this section. Stop.

Figure 4.3: Algorithm $\mathcal{H}$ for Scheduling Arbitrary Tasks in Flow Shops.

has computed the effective ready times and effective deadlines of all subtasks, Step 2 and Step 3 transform the given arbitrary task set into a homogeneous task set, by inflating the processing times of all subtasks on each processor P_j to be $\tau_{max,j}$, the longest processing time among all subtasks on P_j. Step 4 uses Algorithm $\mathcal{A}$ to construct a schedule of the resultant homogeneous task set. Then Step 5 tries to improve the schedule produced by Algorithm $\mathcal{A}$ and attempts to construct a feasible schedule $\mathcal{S}$ for the original set of tasks.

Algorithm $\mathcal{H}$ is relatively simple, with complexity $\mathcal{O}(n\ log\ n + nm)$. It provides us with a way to find a feasible schedule of $\mathcal{T}$. By using Algorithm $\mathcal{A}$, Step 4 defines the order in which the tasks are executed on the processors and produces a permutation schedule.

4.3.1 Suboptimality of Algorithm $\mathcal{H}$

Algorithm $\mathcal{H}$ is not optimal for 2 reasons. First, it transforms the given task set into a homogeneous task set. Second, it considers only permutation schedules.

Inflating the processing times of the subtasks to generate the homogeneous task set $\mathcal{T}_{inf}$ in Step 3 increases the workload (consisting of the actual workload and added idle times) that is to be scheduled on the processors. This may in turn increase the number of release-time and deadline constraints that cannot be met. Moreover, Algorithm $\mathcal{A}$, used to schedule the inflated task set $\mathcal{T}_{inf}$, is not optimal for scheduling the original task set $\mathcal{T}$. In particular, Step 2 in Algorithm $\mathcal{A}$ is not optimal for scheduling the original subtasks T_{ib} on the chosen bottleneck processor P_b. Algorithm $\mathcal{A}$ propagates the schedule on the bottleneck processor onto the remaining processors in its Step 3 by reserving $\tau_{max,j}$ time units for the execution of each subtask on processor P_j. This step may lead to an infeasible schedule of $\mathcal{T}$ even when the schedule $\mathcal{S}_b$ of $\{T_{ib}\}$ on the bottleneck processor P_b is feasible. The inflated task set $\mathcal{T}_{inf}$ includes idle times on all the processors. Therefore the schedule generated by the invocation of Algorithm $\mathcal{A}$ in Step 4 can be improved.

One way to reduce the bad effects mentioned above is to add a compaction step that reduces as much as possible the idle times introduced in Step 3 of Algorithm $\mathcal{H}$. To explain this compaction step, Step 5, let t_{ij} be the start time of T_{ij} on P_j. We note again that $\mathcal{S}$ is a permutation schedule. Let the tasks be indexed so that $t_{ij} < t_{hj}$ whenever $i < h$. In other words, T_{1j} starts before T_{2j}, T_{2j} starts before T_{3j}, and so on, for all j. The subtask T_{ij} starts execution on processor P_j at time

$$t_{ij} = t_{ib} + \sum_{k=b}^{j-1} \tau_{max,k} \tag{4.1}$$

for $j > b$ and

$$t_{ij} = t_{ib} - \sum_{k=j+1}^{b-1} \tau_{max,k} - \tau_{ij} \tag{4.2}$$

for $j < b$. However, we can start the execution of the subtask T_{ij} as soon as $T_{(i-1)j}$ terminates and frees the processor P_j. We have to take care not to begin the execution of T_{ij} before the effective release time r_{ij} of T_{ij}. These considerations are taken into account in *Algorithm $\mathcal{C}$*, which is described in Figure 4.4. This algorithm, with complexity $O(nm)$, is used in Step 5. It compacts the schedule $\mathcal{S}$ generated in Step 4 of Algorithm $\mathcal{H}$.

The example given by Table 4.2 and Figure 4.5 illustrates Algorithm $\mathcal{H}$. From Table 4.2 we see that T_3 has the longest processing time on P_1, T_1 the longest processing time on P_2, T_4 on P_3, and T_3 on P_4. Therefore, $\tau_{max,1} = \tau_{31}$, $\tau_{max,2} = \tau_{12}$, and so on. The processor

Algorithm $\mathcal{C}$:

Input: A schedule $\mathcal{S}$ generated in Step 4 of Algorithm $\mathcal{H}$.

Output: A compacted schedule with reduced idle time.

Step 1: Set $\tilde{r}_{ij} = r_{ij}$ for all i and j.

Step 2: Perform the following steps:

```
$t_{11} = max(t_{11}, \tilde{r}_{11})$
for $j = 2$ to $m$ do
    $t_{1j} = t_{1(j-1)} + \tau_{1(j-1)}$
endfor
for $i = 2$ to $n$ do
    for $j = 1$ to $m - 1$ do
        $t_{ij} = max(t_{(i-1)j} + \tau_{(i-1)j}, \tilde{r}_{ij})$
        $\tilde{r}_{i(j+1)} = t_{ij} + \tau_{ij}$
    endfor
    $t_{im} = max(t_{(i-1)m} + \tau_{(i-1)m}, \tilde{r}_{im})$
endfor
```

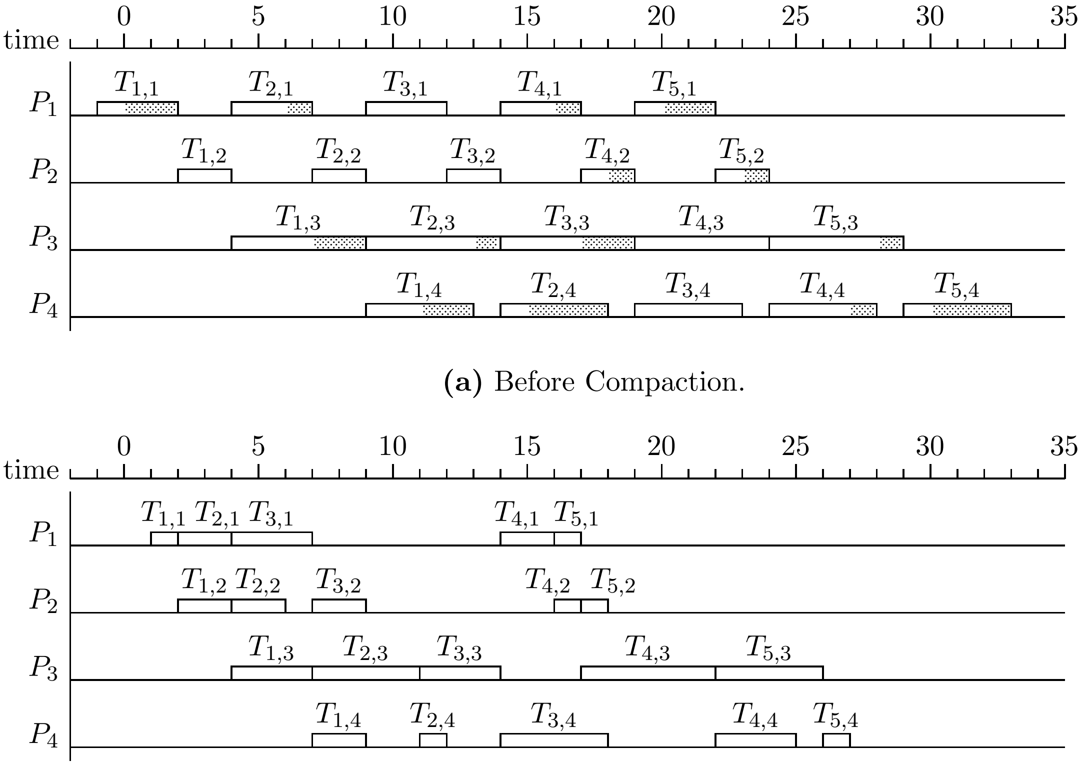
Figure 4.4: Algorithm $\mathcal{C}$.

with the longest $\tau_{max,j}$ is P_3. Therefore, Algorithm $\mathcal{A}$ in Step 4 uses P_3 as the bottleneck processor. Figure 4.5a shows the schedule produced in the first four steps in Algorithm $\mathcal{H}$. After compaction in Step 5, the final schedule is shown in Figure 4.5b. We note that T_1 and T_5 miss their deadlines in the schedule of Figure 4.5a. Moreover, T_1 is forced to start before its release time. Every task meets its release time and deadline in the compacted schedule of Figure 4.5b.

In practice, however, the compaction step (Step 5) of Algorithm $\mathcal{H}$ can be omitted, and be replaced by a priority-driven local scheduler on each processor. The local scheduler on any processor P_i executes the subtasks in the order determined in Step 4. The subtasks on P_1 become ready at their end-to-end release times and the subtasks on subsequent processors

Tasks	r_i	d_i	τ_{i1}	τ_{i2}	τ_{i3}	τ_{i4}
T_1	1	10	1	**2**	3	2
T_2	1	16	2	2	4	1
T_3	1	22	**3**	2	3	**4**
T_4	14	28	2	1	**5**	3
T_5	14	29	1	1	4	1

Table 4.2: Task Set with Arbitrary Processing Times.



(a) Before Compaction.

(b) After Compaction.

Figure 4.5: A Schedule Produced by Algorithm $\mathcal{H}$.

become ready when their predecessor subtasks complete. Such a policy naturally generates a compacted schedule that is identical to the schedule generated by Algorithm $\mathcal{C}$.

Again, the second reason for Algorithm $\mathcal{H}$'s being suboptimal arises from the fact that it considers only permutation schedules. In flow shops with two or more processors, it is possible that there is no feasible permutation schedule when feasible schedules exist. In other words, the

Tasks	r_i	d_i	τ_{i1}	τ_{i2}	τ_{i3}
T_1	0	14	8	2	2
T_2	0	22	1	7	2
T_3	0	22	1	7	2

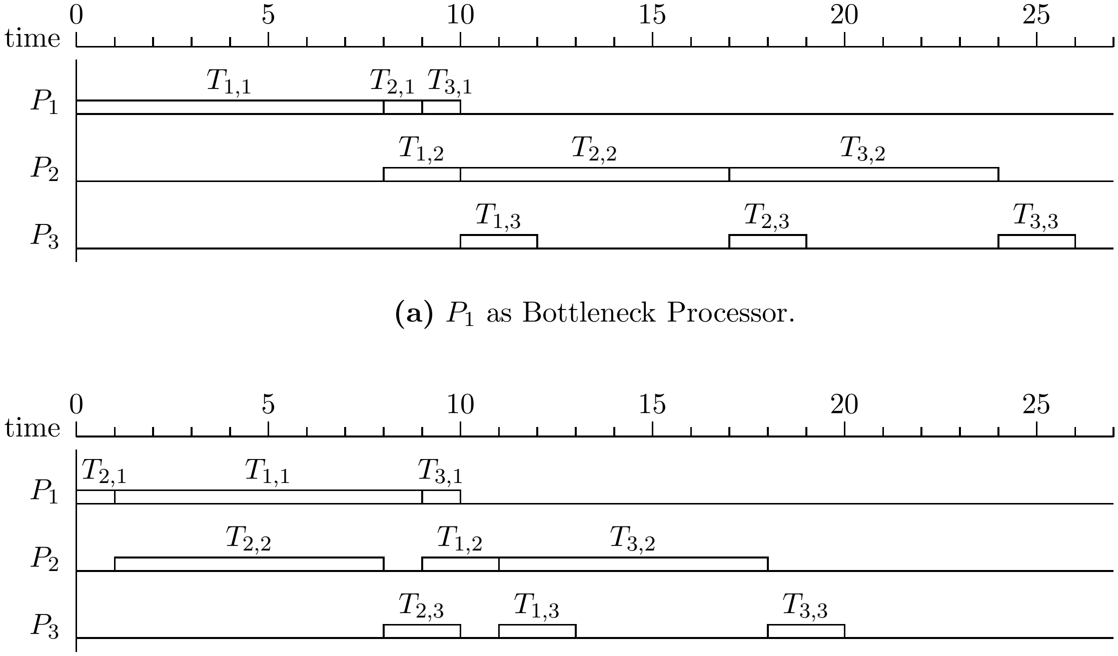
Table 4.3: Task Set where Step 4 of Algorithm $\mathcal{H}$ Fails.

order of execution of the subtasks may vary from processor to processor in all feasible schedules. By generating only permutation schedules, Algorithm $\mathcal{H}$ fails to find a feasible schedule in such cases. Even when feasible permutation schedules exist, Algorithm $\mathcal{H}$ can fail because Step 4 may generate a wrong execution order for subtasks on the bottleneck processor P_b. This also can be caused by the wrong choice of the bottleneck processor.

Table 4.3 and Figure 4.6 show an example where P_1 is the processor with the longest subtask and P_2 is the processor with the largest sum of the processing times of all subtasks on that processor. Using P_1 as the bottleneck processor, Algorithm $\mathcal{A}$ produces a schedule that does not meet all the deadlines, whereas the choice of P_2 as the bottleneck processor results in a feasible schedule. As mentioned earlier, even when the choice of the bottleneck processor is correct, (that is, there exists a feasible schedule on the processor that can successfully be propagated,) Step 2 in Algorithm $\mathcal{A}$ is not optimal for scheduling the original set of subtasks T_{ib} on P_b. Algorithm $\mathcal{H}$ can therefore fail to generate a feasible schedule on P_b when a feasible schedule of the subtasks on P_b exists.

4.3.2 Performance of Algorithm $\mathcal{H}$

We now describe two series of simulation experiments to measure the performance of Algorithm $\mathcal{H}$ and the results of these experiments. The first series of experiments determines the *success rate* of Algorithm $\mathcal{H}$. By success rate we mean the probability of the algorithm being successful in generating a feasible schedule, given that such a schedule exists. In the second series of experiments, we compare Algorithm $\mathcal{H}$ to a group of algorithms that are often used to schedule tasks with timing constraints. In particular, we compare the algorithms using success rate and total tardiness of the generated schedules as performance measures.



(a) P_1 as Bottleneck Processor.

(b) P_2 as Bottleneck Processor.

Figure 4.6: Schedules for Two Choices of Bottleneck Processors.

In both series of experiments, we fed the scheduling algorithms with a number of task sets and analyzed the generated schedules. To describe how the task sets were generated, we use the notation $\mathbf{x} \sim N(\mu; \sigma)$ to indicate that the random variable $\mathbf{x}$ has a normal distribution $N(\mu; \sigma)$ with mean μ and standard deviation σ. We use $N_0(\mu; \sigma)$ to denote the "normal distribution" obtained by truncating $N(\mu; \sigma)$ at zero so that samples distributed according to $N_0(\mu; \sigma)$ have only nonnegative values. Clearly, μ is not the mean of the truncated distribution $N_0(\mu; \sigma)$. Neither is σ the standard deviation of $N_0(\mu; \sigma)$. Nevertheless, the parameters μ and σ give an intuitive description of the truncated distribution $N_0(\mu; \sigma)$. For sake of simplicity, we refer to them loosely as the "mean" and the "standard deviation" of the truncated distribution $N_0(\mu; \sigma)$. We use the term *normalized* normal distribution to indicate a normal distribution with mean $\mu = 1$. Similarly, we use $\mathbf{x} \sim U(a; b)$ to indicate that the random variable $\mathbf{x}$ is uniformingly distributed over the interval from a to b.

The common characteristics of the task sets in each simulation run are determined by the following parameters. We call them *simulation parameters*:

n: The *number of tasks* in the flow shop.

m: The *number of processors* in the flow shop. The parameters n and m control the size of the task sets being generated.

ρ: This parameter controls the ratio between the mean task processing time and mean interval length between consecutive release times. When ρ has a small value, the task sets contain short tasks whose release times are far apart. With a larger value for ρ, the resultant task sets are more difficult to schedule because they have long tasks whose release times are close together.

σ_τ: Standard deviation σ_τ of the normalized distribution of the processing time of the subtasks on a processor. A smaller value for σ_τ leads to task sets that are more homogeneous. Indeed, if σ_τ is zero, the generated task sets are homogeneous. With increasing value for σ_τ, task sets are more and more non-homogeneous.

μ_l: *Mean laxity factor.* This parameter controls the ratio of mean laxity to mean total processing time in the following way: A small value for μ_l results in task sets that have little laxity. Increasing the value of μ_l increases the mean laxity in the task sets being generated.

σ_l: Standard deviation of the normalized normal distribution of laxity factors. We note that, for a homogeneous task set, a zero value for σ_l leads to a constant laxity. Moreover, the interval between the end-to-end release time and the end-to-end deadline of tasks in these sets have a constant length.

All the task sets used in our experiments are generated in the following way:

(1) The *end-to-end release times* r_i are uniformingly distributed in the range $[0, I]$:

$$r_i \sim U(0; I).$$

When the number n of tasks becomes large, the interval between any two consecutive release times become exponentially distributed with mean I/n.

(2) The *mean processing times* μ_{τ_j} of the subtasks on each processor P_j are uniformingly distributed in the range $[0, \rho I]$:

$$\mu_{\tau_j} \sim U(0; \rho I).$$

(3) The *processing time* τ_{ij} of any subtask on P_j is a sample from the truncated normalized normal distribution $N_0(1, \sigma_\tau)$ multiplied by μ_{τ_j}:

$$\frac{\tau_{ij}}{\mu_{\tau_j}} \sim \mu_{\tau_j} N_0(1, \sigma_\tau).$$

(4) The *end-to-end deadline* d_i of each task T_i is determined by adding the total processing time of the task and a *laxity* to the end-to-end release time. The laxity is determined by multiplying the total processing time of the task with a *laxity factor*. The larger the laxity factor for a given task, the larger the amount of laxity of that task. The laxity factor l_i for task T_i is a sample from the normal distribution $N_0(1, \sigma_l)$ multiplied by the mean laxity factor μ_l:

$$\frac{l_i}{\mu_l} \sim \mu_l N_0(1, \sigma_l).$$

In the discussion of the experiment results, we denote the amount of laxity of a task by the *utilization factor* u_i, a measure that is strongly related to the laxity factor, but more intuitive. The utilization factor is the ratio between the total processing time of the task and the sum of its total processing time and laxity. The relation between utilization factor u_i and the laxity factor l_i is straightforward:

$$u_i = \frac{1}{l_i + 1}, \qquad\qquad l_i = \frac{1 - u_i}{u_i}$$

The same relation obviously holds also for the *mean utilization factor* μ_u and the mean laxity factor μ_l.

In our experiments, we varied the size of the task sets (by varying n and m), the degree of homogeneity (by varying σ_τ), and the amount of laxity (by varying μ_u). We have decided to keep ρ constant in order to limit the result space. The variance of the laxity factors was also kept constant (by not varying σ_l). This is because the distribution of the actual laxity is a function of both the distribution of the laxity factor and the distribution of the total processing time. The effect of varying the variance of the laxity factor is therefore marginal.

Parameter	Setting
n	4, 6, . . . , 22
m	4, 6, . . . , 22
ρ	0.25
$\sigma_\mathcal{T}$	0.05, 0.1, 0.2, 0.3, 0.5
μ_u	0.2, 0.4, 0.6, 0.7
σ_l	0.5

Table 4.4: Settings for Simulation Parameters in Experiments.

Table 4.4 gives an overview of the chosen values of the simulation parameters. The half lengths of the 95% confidence intervals of all the results are below 3%, meaning that in all cases the confidence interval for a measured success rate of x% is $(x \pm 3)$%. In the vast majority of the cases the half lengths are below 1.5%.

4.3.2.1 Success Rate of Algorithm $\mathcal{H}$

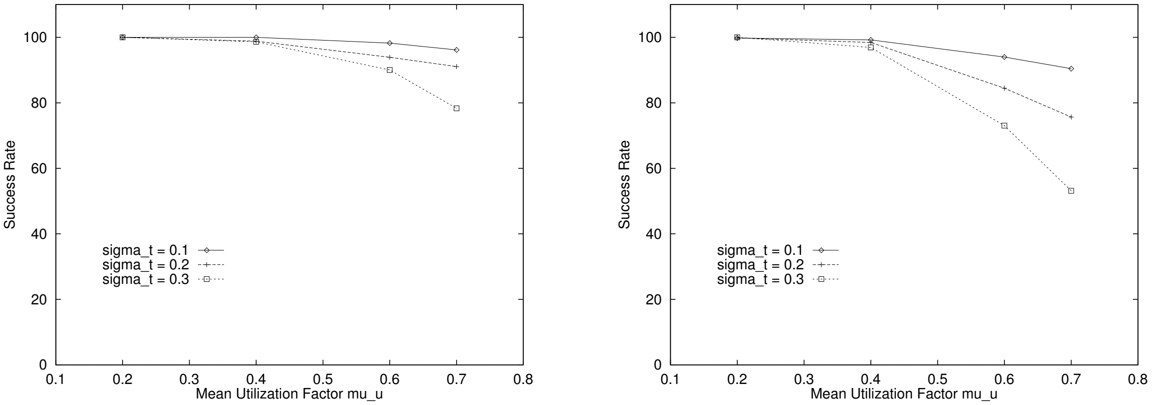
In this experiment, we measured the success rate of Algorithm $\mathcal{H}$, that is, the probability of Algorithm $\mathcal{H}$ finding a feasible schedule when such a schedule exists. In Figure 4.7 the success rate of Algorithm $\mathcal{H}$ is plotted as a function of the mean utilization factor μ_u. We see that the more the task set resembles a homogeneous task set (the smaller $\sigma_\mathcal{T}$), the better Algorithm $\mathcal{H}$ performs. This behavior is to expected, since Algorithm $\mathcal{H}$ is based on Algorithm $\mathcal{A}$, which is known to be optimal for the case of homogeneous task sets. Algorithm $\mathcal{H}$ performs worse with decreasing amount of laxity per task. This also is to be expected, given that it is inherently more difficult to schedule a task set when the laxity is scarce. In some of these experiments, the amounts of laxity are very small (in the order of 40% of the processing time of a task).

In a sequence of experiments we tried larger amounts of laxity (in the order of 4 times the processing time of a task) and with larger numbers of tasks on 4 processors. The remaining other parameters are $\sigma_\mathcal{T} = 0.1$ and $\mu_u = 0.2$. The results, shown in Figure 4.8, indicate that Algorithm $\mathcal{H}$ performs very well.

Unfortunately, the task sets used in this series of experiments are either very small (4 or 6 tasks on 4 or 6 processors) or have large amounts of laxity. This is due to the difficulty in generating feasible task sets with random parameters.

Parameter	Setting
n	4, 6
m	4
ρ	0.25
σ_τ	0.1, 0.2, 0.3
μ_u	0.2, 0.4, 0.6, 0.7
σ_l	0.5

(a) Simulation Parameters



(b) 4 Tasks on 4 Processors

(c) 6 Tasks on 4 Processors

Figure 4.7: Success Rate of Algorithm $\mathcal{H}$ for Small Task Sets.

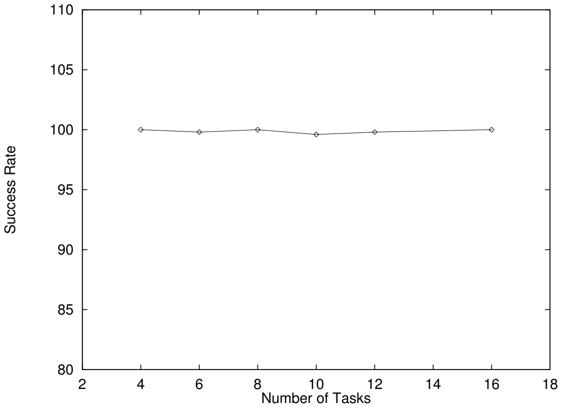


Figure 4.8: Success Rate of Algorithm $\mathcal{H}$ for Larger Task Sets ($\mu_u = 0.2$, $\sigma_\tau = 0.1$).

Two problems arise when generating task sets with random parameters: First, with a small mean laxity factor μ_l or a large number n of tasks in the flow shop, most of the task sets generated are not feasible. Before a feasible task set is found, a large number of others that are not feasible must be discarded. Moreover, every task set being generated has to be tested for feasibility. Determining the feasibility of a task set has been shown in Chapter 2 to be $\mathcal{NP}$-hard, and therefore requires expensive search techniques. Although heuristics are used to detect and discard obviously infeasible task sets (Appendix A describes the heuristics used in this process), the cost of generating large numbers of feasible task sets of larger size becomes prohibitively expensive.

The second problem in generating large feasible task sets is more subtle and serious: The process of sampling the task parameters from a set of distributions and then discarding the infeasible task sets may bias the original distributions. To illustrate this problem, let us consider the task sets with a small mean laxity factor μ_l. As described earlier, the laxity factor l_i of each task is a sample from the distribution $N(1, \sigma_l)$, multiplied by the factor μ_l. If μ_l is small, only task sets with large l_i's can be feasible, effectively biasing the laxity factor distribution towards a larger mean. The distributions of the task parameters of the feasible task sets do not reflect the distributions of the simulation parameters (in this example μ_l) anymore. By limiting this experiments to very small task sets and a minimum mean laxity factor of 3/7, we avoid the problem of biasing the original distributions.

4.3.2.2 Comparison of Algorithm $\mathcal{H}$ to Other Algorithms

Although the results in the previous section show the success rates of Algorithm $\mathcal{H}$ in generating feasible schedules, we do not know how difficult it is to schedule those task sets in the first place. In this section, we compare Algorithm $\mathcal{H}$ against five other algorithms of different complexities. These algorithms are:

First-Come-First-Served (FCFS): According to this algorithm, subtasks on the first processor are executed in a *first-come-first-served* order. In other words, subtasks with earlier release times have higher priorities. This order is preserved on the remaining processors. This algorithm makes no effort to consider the task deadlines.

Least-Laxity-First (LLF): Whenever a processor becomes idle, it selects the subtask that has the lowest laxity at that time among all the ready ones, and executes it to completion. This algorithm is non-preemptive and avoids the well known problem of its preemptive counterpart that sometimes degenerates to a processor-sharing policy.

Effective-Earliest-Deadline-First (EEDF): Whenever a processor becomes ready, it selects the subtask that has the earliest effective deadline among all the ready subtasks, and executes it to completion. We note that for homogeneous task sets, EEDF behaves like LLF.

Preemptive-Effective-Earliest-Deadline-First (pEEDF): This is the preemptive counterpart of Algorithm EEDF. Whenever a processor becomes idle or a subtask becomes ready, the processor selects for execution the subtask with the earliest effective deadline. If necessary, the currently executing subtask is preempted.

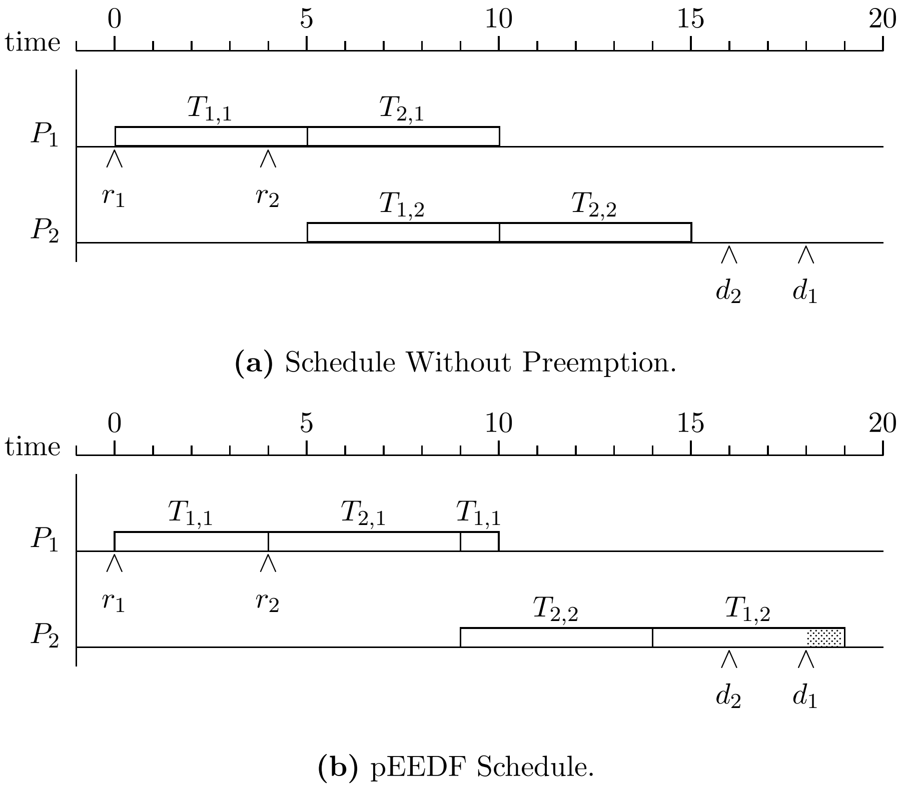
Algorithm Ha: This is an extended version of Algorithm $\mathcal{H}$. It repeatedly executes Algorithm $\mathcal{H}$ until it finds a feasible schedule, up to m times. Each time it selects a different processor as the bottleneck processor. If no feasible schedule is found after all processors have been used as bottleneck processors, the schedule with the lowest total tardiness is returned.

In this series of experiments, we compare the performance of the algorithms listed above and Algorithm $\mathcal{H}$ by measuring the *success rate* and the *relative performance* of each algorithm. The success rate is the rate at which an algorithm generates a feasible schedule. The relative performance is the rate at which the algorithm generates a schedule that is at least as good as the schedules generated by the competing algorithms. Specifically, an algorithm generates a schedule that is at least as good as the others for a given task set if (1) the schedule is feasible, or (2) if no algorithm generates a feasible schedule, and the schedule has the smallest total tardiness among all the schedules. For a given schedule, if a task completes by its deadline, the *tardiness* of the task is zero; otherwise it is equal to the amount of time by which the task misses its deadline. The *total tardiness* of a schedule is the sum of the tardiness of all the tasks. We note that the relative performance is never lower than the success rate and that several algorithms may score as performing the best for a given task set.

The figures plotting the relative performance and the success rates of the algorithms studied here are in Appendix B. In these plots the vertical axis represent either the success rate or the relative performance. In Figures B.1 – B.9, we display the relative performance of the algorithms as a function of the mean utilization factor μ_u for different degrees of homogeneity (i.e. different values for σ_τ). As expected, in all experiments Algorithm Ha performs consistently better than Algorithm $\mathcal{H}$. This is because Algorithm Ha repeatedly uses Algorithm $\mathcal{H}$ and improves on it treating each time a different processor as the bottleneck processor. For task sets with high and medium degrees of homogeneity, that is for $\sigma_\tau = 0.05$ and 0.1, both Algorithm $\mathcal{H}$ and Algorithm Ha perform very good, for large and small task sets. Figures B.1 – B.9 show how for $\sigma_t = 0.05$ and $\sigma_t = 0.1$, Algorithm Ha performs consistently best. In the same cases, Algorithm $\mathcal{H}$ performs at least as good as the remaining algorithms, except for some cases where the EEDF algorithm performs slightly better. For example, this is the case in Figure B.5 where for $\mu_u = 0.4$ the EEDF algorithm performs better than Algorithm $\mathcal{H}$, even for very high degrees of homogeneity ($\sigma_t = 0.05$). If we compare with the success rates of different algorithms in Figure B.13, we note that the success rates drop to nearly zero at $\mu_u = 0.4$, that is, at the same value of μ_u at which we notice a drop in the performance of Algorithm $\mathcal{H}$ and where the EEDF algorithm performs better.

For small task sets and low degrees of homogeneity (e.g. in Figure B.1 with $\sigma_\tau = 0.3$ or 0.5), the preemptive pEEDF algorithm performs better than Algorithm Ha. When the size of the task sets increases, pEEDF algorithm does not perform as well anymore, especially with high utilization factors. The reason that preemption does not pay off with larger task sets is illustrated by the example in Figure 4.9. This example compares a pEEDF schedule against a non-preemptive schedule. By preempting subtask T_{11} after 4 time units, task T_2 finishes earlier by one time unit; but it causes the completion time of task T_1 to be increased by 5 time units. This increase is enough for T_1 to miss its deadline. The simulation results show that this effect becomes more pronounced for larger task sets, containing a larger number of tasks or a larger number of processors.

Figures B.10 – B.18 show the success rates of the algorithms as a function of the mean utilization factor for different degrees of homogeneity. As opposed to the experiments described in Section 4.3.2.1, in this series of experiments the task sets are not checked for schedulability before they are used. Therefore, the value of the success rate does not necessarily reflect the



(a) Schedule Without Preemption.

(b) pEEDF Schedule.

Figure 4.9: Preemption not Always Pays Off.

quality of an algorithm. Specifically, a low success rate may indicate that few task sets were schedulable in the first place. For example, in Figures B.13 – B.16 the success rate for task sets containing 14 tasks on 4 processors is very low except for task sets with very low utilization factors. This does not necessarily indicate that the performance of the algorithms are poor, since it is very unlikely that a randomly chosen task set of that size is schedulable. Therefore, a large portion of the task sets of this size that were used in this experiment are not schedulable by any algorithm.

The results for the success rate are similar as for the relative performance: Algorithm FCFS has lower success rates for all sizes of task sets. For example, as shown in Figure B.14, the success rate of Algorithm FCFS is substantially lower than all the remaining algorithms, even when the task sets with small utilization factors μ_u. Moreover, with increasing μ_u its success rate drops to nearly zero, whereas all other algorithms maintain high success rates. Figures B.14, B.15, and B.18 show that the EEDF algorithm and Algorithm LLF have very similar success rates. Indeed, for task sets with high degrees of homogeneity, the success rates of the two algorithms are nearly identical. The similarity in the performance of the EEDF algorithm and Algorithm LLF reflects the fact that for homogeneous task sets the two algorithms behave the same. In contrast, the values of the relative performance of these algorithms are not similiar. For example, Figures B.5, B.6, and B.9 show a substantial difference between the relative performance of these two algorithms. This difference increases with decreasing degree of homogeneity or increasing mean utilization factor. The reason for this difference in relative performance of the two algorithms that have similar behaviour, is due to the way the relative performance is calculated. If the schedules generated by two algorithms are nearly identical but have a slightly different total tardiness, only one algorithm succeeds when determining the relative performance. Although the schedules generated by the EEDF algorithm are similar to the ones generated by Algorithm LLF, their total tardiness is generally smaller.

Algorithm Ha and Algorithm $\mathcal{H}$ perform best for high and medium degrees of homogeneity, and Algorithm Ha performs best for larger task sets. Figure B.14 supports this conclusion; it shows that Algorithm Ha, Algorithm $\mathcal{H}$, and the pEEDF algorithm have nearly identical success rates for nearly homogeneous task sets. With decreasing degrees of homogeneity, Algorithm $\mathcal{H}$'s success rate decreases in relation to the success rates of Algorithm Ha and pEEDF. While for $\sigma_\tau = 0.05$ all three algorithms have a success rate of nearly 100% at $\mu_u = 0.4$, for $\sigma_\tau = 0.5$ the success rate of Algorithm $\mathcal{H}$ drops to below 80%, whereas the success rate of both Algorithm Ha and pEEDF is around 90%. Figures B.15 and B.17 clearly show that the success rates of both Algorithm $\mathcal{H}$ and Algorithm Ha are higher than those of the pEEDF algorithm. the only exceptions are very non-homogeneous task sets ($\sigma_\tau = 0.5$), where Algorithm Ha and pEEDF have very similar success rates.

We note that when the numbers of processors in a flow shop increases, the success rates for all the algorithms increase too. This is counter-intuitive. We expect that as the number of processors increases and the size of the system becomes larger, task sets should become more difficult to schedule. A careful examination shows that the reason for the increase in success rates with larger numbers of processors lays in the way the task sets are generated: The amount of laxity per task is determined by multiplying the total processing time of the task by the laxity factor. When the number of processors increases, the total processing time of the tasks in the task sets increases, and so does the amount of laxity. The negative effect that additional processors have on the schedulability is more than compensated by the positive effect of the additional laxity.

Figures B.19 – B.24 show the relative performance of the algorithms as a function of the size of the task sets. The results are shown for high and low degrees of homogeneity (with $\mu_{\mathcal{T}}$ equal to 0.05 and 0.5) and for small, medium, and high utilization factors (with μ_l equal to 0.2, 0.4, and 0.7). With the exception of the case of small degrees of homogeneity and medium utilization factors (see Figure B.22), Algorithm Ha clearly outperforms all the other algorithms for all task-set sizes. In general, when the size of the task set varies, the performance of all the algorithms varies in a similar way. The relative performance increases with the number of processors and decreases with the number of tasks. Increasing the number of tasks reduces the relative performance whereas increasing the number of processors increases it. The latter is due to the increase in laxity, as was explained earlier. In particular, the results in Figure B.24 show that for systems with very little laxity ($\mu_u = 0.7$), the performance of some algorithms does not always decrease with decreasing number of processors. In these cases, Algorithm $\mathcal{H}$ and the two deadline-driven algorithms EEDF and pEEDF perform slightly better on small numbers of processors. As can be seen from Figures B.25 – B.30, the success rates for these cases are very small, and so do not contribute to the relative performance. The increase in relative performance of Algorithm $\mathcal{H}$ on task sets with few processors can be explained easily: With decreasing numbers of processors, the probability of choosing the same bottleneck processor as Algorithm Ha increases, and so does the probability of generating the same schedule. Similarly, the probability that the EEDF algorithm generates a schedule that is identical to the one generated by Algorithm Ha increases with decreasing numbers of processors. This argument does not hold for the pEEDF algorithm. Because of the small laxity in the task sets, the schedules generated by the preemptive algorithm are apt to contain many preemptions, thus bearing little resemblance with the schedules generated by Algorithm Ha. However, the negative effects of preemptions are not as pronounced for small numbers of processors.

Figures B.25 – B.30 show the success rate as function of the size of the task set. The results show once more how Algorithm Ha outperforms the other algorithms in all cases, except for task sets in Figure B.28 which have a low degree of homogeneity and little laxity. These figures also show that the success rate increases with increasing numbers of processors. Again, this is due to the higher amount of laxity in task sets with larger numbers of processors.

4.4 Scheduling Highly Non-Homogeneous Task Sets

While in some systems (such as fixed-size packet switched networks) the task sets are homogeneous or nearly homogeneous, in other systems this may not be the case. For instance, the data transmitted over a sequence of communication links may consist of large-sized units (such as entire files) and small units (such as commands). Here we model the communication links as processors. We can model such a system of data transmissions as a task set that contains tasks with long subtasks and tasks with short subtasks. This task set is highly non-homogeneous. However, the task set in this example (call it $\mathcal{T}$) can be partitioned into two task sets $\mathcal{T}_S$ and $\mathcal{T}_L$. $\mathcal{T}_L$ consists of *long tasks* with long subtasks, and $\mathcal{T}_S$ consists of *short tasks* with short subtasks. Each of these two task sets is homogeneous or nearly homogeneous.

In this section we present an algorithm to schedule non-homogeneous task sets that can be decomposed into two identical-length task sets. We call this algorithm Algorithm $\mathcal{PCC}$ and describe it in Figure 4.10. When the release times of all the tasks are identical, Algorithm $\mathcal{PCC}$ is optimal for a task set that consist of two identical-length task sets with processing times τ and $p\tau$, where p is a positive integer.

Algorithm $\mathcal{PCC}$ in Figure 4.10 uses the EEDF algorithm to schedule the subtasks on each of the processors preemptively, starting from the first processor P_1. The release times of the subtasks on P_1 are the end-to-end release times of the tasks. On each of the subsequent processors, the release times of the subtasks are the completion times of the predecessor subtasks on the preceding processor.

When all release times on the first processor are identical, there is no preemption on that processor. On successive processors, preemptions happen only at times that are integer multiples of τ, because the smallest granule of processing time is τ. For this reason, we can think of each subtask in $\mathcal{T}_L$ as a sequence of p subtasks of length τ. We do not need to consider preemptions explicitly in the proof of the following lemma, which will be used in the proof for the optimality of Algorithm $\mathcal{PCC}$.

Lemma 1. Let T_a and T_b be two tasks of identical length in a flow shop. If $r_a \leq r_b$ and $d_a \leq d_b$, any feasible schedule $\mathcal{S}$ can be transformed into another feasible schedule $\tilde{\mathcal{S}}$ where, on every processor, T_a executes before T_b.

```
Algorithm $\mathcal{PCC}$:

Input: Task parameters $d_i$, $\tau$, and $p$ of $\mathcal{T} = \mathcal{T}_S + \mathcal{T}_L$. Both $\mathcal{T}_S$ and $\mathcal{T}_L$ are identical-length task sets,
    with processing time $\tau$ and $p\tau$.
Output: A feasible schedule of $\mathcal{T}$, or the conclusion that feasible schedules of $\mathcal{T}$ do not exist.

Step 1: Determine the effective deadlines $d_{ij}$ of all subtasks. Set the effective release times $r_{i1} = r_i$
    and the effective release time $r_{ij} = 0$ for all $j = 2, \cdots, m$.
Step 2: On each processor $P_j$, starting from $P_1$ and continuing until $P_m$, do the following:
        (1) Use preemptive EEDF on $P_j$ to schedule the subtasks on $P_j$.
        (2) For each $T_{ij} \in \mathcal{T}$, set the effective release time $r_{i(j+1)}$ of $T_{i(j+1)}$ to the time when $T_{ij}$
            completes on $P_j$.
```

Figure 4.10: Algorithm $\mathcal{PCC}$ for Scheduling Task Sets with Two Classes of Identical-Length Tasks.

Proof. We need to consider two cases: (1) all the subtasks in T_a and T_b have processing time $p\tau$, and (2) all the subtasks have processing time τ. We will prove the lemma is true for case (1). That it is true for case (2) follows straightforwardly.

In case (1) the transformation works as follows: T_{aj} and T_{bj} together have $2p$ time units according to the schedule $\mathcal{S}$. In the transformed schedule $\tilde{\mathcal{S}}$, T_{aj} is scheduled during the first p of these time units, and T_{bj} is scheduled during the remaining p units. When there is only one processor, we need not be concerned with subtasks on different processors and the dependencies between the subtasks. $\tilde{\mathcal{S}}$ is therefore a valid schedule. Since $d_a < d_b$, if both T_a and T_b meet their deadlines according to $\mathcal{S}$, they meet their deadline according to $\tilde{\mathcal{S}}$.

Suppose that the lemma holds for the k-processor case. We now prove that it holds for the $(k+1)$-processor case also. In $\mathcal{S}$, either T_{ak} completes at time t_1 and T_{bk} completes at a later time t_2, or vice versa. Because T_a's deadline is no later than T_b and all the subtasks in T_a and T_b have the same processing time $p\tau$, on processor P_k, the effective deadline d_{ak} of T_{ak} is no later than d_{bk}. The first k processors and the subtasks on them can therefore be viewed as a k-processor system. By induction hypothesis, the schedule for the first k processors can be transformed into one with the desired property. The resultant schedule is identical to the schedule $\tilde{\mathcal{S}}$ on the first k processors. Moreover, in $\tilde{\mathcal{S}}$, T_{ak} completes at or before time t_1 and T_{bk} completes exactly at time t_2. Since all dependencies are preserved in $\mathcal{S}$, both $T_{a(k+1)}$ and $T_{b(k+1)}$ are scheduled at or after time t_1. In $\tilde{\mathcal{S}}$, $T_{a(k+1)}$ is scheduled during the first p units of the $2p$ units of time during which $T_{a(k+1)}$ and $T_{b(k+1)}$ are scheduled in $\mathcal{S}$. Clearly, the dependency between T_{ak} and $T_{a(k+1)}$ is preserved in $\tilde{\mathcal{S}}$. After the transformation, $T_{b(k+1)}$ is scheduled in a later interval than it is in $\mathcal{S}$; the dependency between T_{bk} and $T_{b(k+1)}$ is preserved too in $\tilde{\mathcal{S}}$. We conclude that all the dependencies are preserved in $\tilde{\mathcal{S}}$. Moreover in $\tilde{\mathcal{S}}$, T_a completes no later than in $\mathcal{S}$, and T_b completes either when it completes in $\mathcal{S}$ or when T_a completes in $\mathcal{S}$. Both T_a and T_b meet their deadlines in $\tilde{\mathcal{S}}$. □

Theorem 4. When used to schedule any flow-shop task set $\mathcal{T}$ that consists of two identical-length task sets $\mathcal{T}_S$ and $\mathcal{T}_L$ whose tasks have identical release times and whose processing times are τ and $p\tau$, respectively, for some positive integer p, Algorithm $\mathcal{PCC}$ never fails to find an existing feasible schedule.

Proof. We prove that any feasible schedule $\mathcal{S}$ can be transformed into a schedule generated by Algorithm $\mathcal{PCC}$ (a *$\mathcal{PCC}$ schedule*, for short). Let $\mathcal{S}$ be a feasible schedule of the task set $\mathcal{T}$. We transform $\mathcal{S}$ into a $\mathcal{PCC}$ schedule by applying the steps described below. In the following description, by *swapping* two intervals I_a and I_b of length τ between two subtasks T_{aj} and T_{bj}, we mean the following: Before the swapping takes place, I_a and I_b are assigned to T_{aj} and T_{bj}, respectively. (In other words, T_{aj} (or a portion of T_{aj}) is scheduled in I_a and T_{bj} (or a portion of T_{bj}) is scheduled in I_b.) After swapping, I_b is assigned to T_{aj} and I_a is assigned to T_{bj}.

The two-step transformation works as follows: In a first step, we swap the time intervals assigned to all short tasks according to $\mathcal{S}$ so that, after all the swappings are done, every task in $\mathcal{T}_S$ executes before all other tasks in $\mathcal{T}_S$ that have later deadlines. This is done by repeatedly swapping the intervals assigned to a pair of short tasks that are not scheduled in order, until they are in order. Similarly, we repeatedly swap the intervals assigned to pairs of long tasks that are not scheduled in order according to $\mathcal{S}$ until every task in $\mathcal{T}_L$ executes before all other tasks in $\mathcal{T}_L$ that have later deadlines. Lemma 1 shows that this can always be done. We call the resultant ordered schedule $\mathcal{S}_o$.

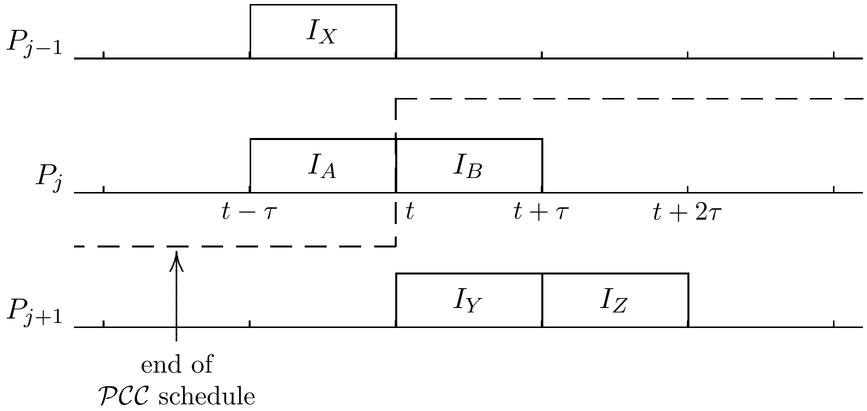


Figure 4.11: Transforming an Arbitrary Schedule into a $\mathcal{PCC}$ Schedule.

In a second step, we scan the ordered schedule $\mathcal{S}_o$ from the left to the right on each processor, starting from processor P_1, then P_2 and so on. Specifically, suppose that while scanning the schedule $\mathcal{S}_o$ on processor P_j, we find that the schedule $\mathcal{S}_o$ is a $\mathcal{PCC}$ schedule until time $t > \tau$; the schedule on processors $P_1, P_2, \cdots, P_{j-1}$ is already a $\mathcal{PCC}$ schedule. At t, we encounter one of the following conditions, making the schedule $\mathcal{S}_o$ not a $\mathcal{PCC}$ schedule. We stop and take the actions described below. To describe these conditions, we will use the following notations: The typical situation encountered while scanning the schedule is described in Figure 4.11. I_A and I_B refer to two adjacent intervals of length τ on processor P_j. I_X refers to the interval which starts and ends at the same time as I_A but is on processor P_{j-1}. I_Y and I_Z are two adjacent intervals of length τ on processor P_{j+1}. I_Y starts at the same time as I_B and I_Z starts after I_Y. Each of the intervals I_A, I_B, I_X, I_Y, and I_Z is either an idle interval, or is assigned to an entire subtask of a short task in $\mathcal{T}_S$ or to a portion of a subtask of a long task in $\mathcal{T}_L$.

(1) The interval I_A is an idle interval and I_B is assigned to T_{aj}: If I_X is not assigned to $T_{a(j-1)}$, swap the intervals I_A and I_B, that is, assign I_A to T_{aj} and leave I_B idle.

(2) I_A is assigned to T_{aj} and I_B is assigned to T_{bj}, where $T_a \in \mathcal{T}_S$ and $T_b \in \mathcal{T}_L$, and $d_{aj} > d_{bj}$: If I_X is assigned to $T_{b(j-1)}$, we continue scanning. Otherwise, we swap I_A and I_B on processor P_j. If I_Y is assigned to $T_{a(j+1)}$, swap I_Y and I_Z so that the dependency between T_{aj} and $T_{a(j+1)}$ is preserved. Similarly, on each of the remaining processors, if necessary

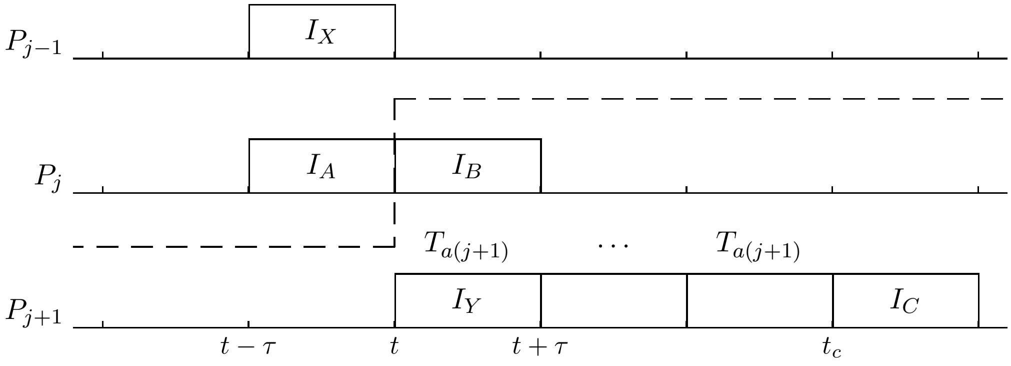


Figure 4.12: $T_{a(j+1)}$ Starts at Time t.

to preserve the dependencies of subtasks in T_a, swap the interval assigned to T_a with the immediately following interval of length τ.

(3) I_A is assigned to T_{aj} and I_B is assigned to T_{bj}, where $T_a \in \mathcal{T}_L$ and $T_b \in \mathcal{T}_S$, and $d_{aj} > d_{bj}$: If I_X is assigned to $T_{b(j-1)}$, continue scanning. Otherwise, swap I_A and I_B. If I_Y is assigned to $T_{a(j+1)}$, then let I_C (starting at time t_c) be the first interval after time t assigned to a subtask other than $T_{a(j+1)}$ as illustrated by Figure 4.12. Swap I_Y and I_C. Similarly, on each of the remaining processors, if necessary to preserve the dependency of subtasks in T_a, swap the first interval assigned to T_a with the first interval of length τ that is not assigned to T_a.

We now argue that the actions described above can always be taken without creating an invalid or infeasible schedule: Case (1) is clearly always possible.

In Case (2), the swapping of I_A and I_B on P_j and I_Y and I_Z on P_{j+1} preserve all dependencies between subtasks on P_j and on P_{j+1}. The same holds true on the remaining processors when the interval assigned to T_a is swapped with the immediately following interval in order to preserve the dependencies of subtasks in T_a. Case (2) is therefore always possible.

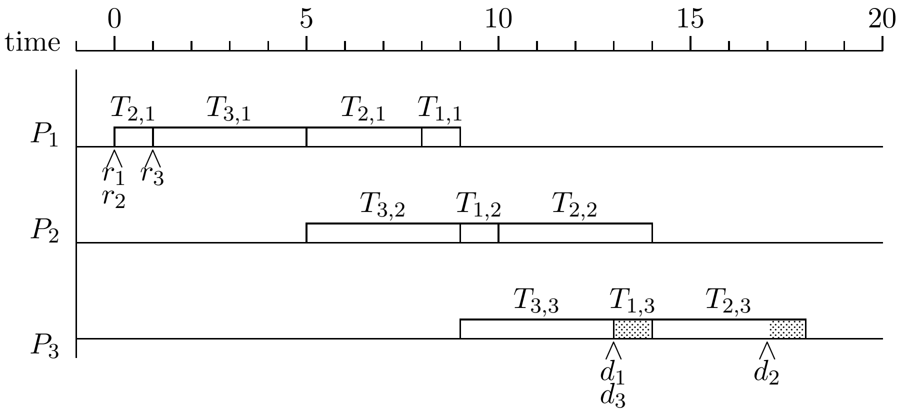
In order to argue that Case (3) is always possible, we need to show first that during the transformation process we never swap two intervals assigned to two tasks both of which belong to either $\mathcal{T}_S$ or $\mathcal{T}_L$; therefore, during the entire transformation process, the orders of tasks within $\mathcal{T}_S$ or $\mathcal{T}_L$ are preserved. To prove this, we now show that the interval I_C is not assigned to a long task by contradiction. Assume that I_C is assigned to T_c, where $T_c \in \mathcal{T}_L$. There is not enough time between $t + \tau$ (the end of the interval I_B) and t_c (the beginning of the interval I_C) to execute T_{cj}. Therefore, at least some portion of T_{cj} executes before some portion of T_{aj}, which is assigned interval I_A. On P_{j+1}, some portion of $T_{a(j+1)}$ precedes a portion of $T_{c(j+1)}$. This contradicts the assumption that after Step 1 the tasks in $\mathcal{T}_L$ are scheduled in order of increasing deadlines. Hence, either I_C is an idle interval, or it is assigned to a short task. If I_C is an idle interval, Case (3) is clearly possible. If, on the other side, I_C is assigned to some short task T_c, we can argue in a similar way as above that the subtask T_{cj} must complete by time $t + \tau$: Assume that T_{cj} completes later than time $t + \tau$. In this case, T_{bj} precedes T_{cj}, and $T_{c(j+1)}$ precedes $T_{b(j+1)}$, contradicting the assumption that all the short tasks are scheduled in order of increasing deadlines. The action taken in Case (3) does therefore not violate any dependency between subtasks on P_j and subtasks on P_{j+1}. That swappings on the remaining processors are possible, if necessary to preserve the dependencies between subtasks in T_a, can be shown by this same argument.

During these repeated transformations we sort the subtasks according to their deadlines and eliminate idle times whenever possible. Hence, this process eventually comes to an end and generates a $\mathcal{PCC}$ schedule.

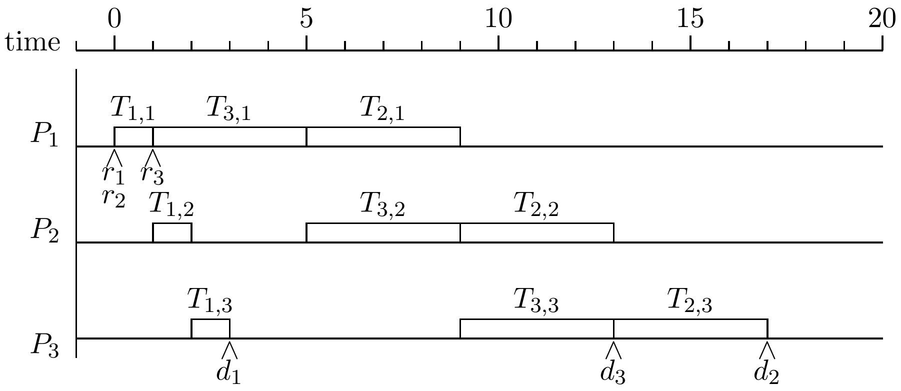
Suppose that Algorithm $\mathcal{PCC}$ fails to find a feasible schedule, that is, Algorithm $\mathcal{PCC}$ generates a schedule that is not feasible, but a feasible schedule exists. We use the transformation process described above to generate a feasible $\mathcal{PCC}$ schedule from the feasible schedule. This leads to a contradiction. Therefore, no feasible schedule exists. □

Algorithm $\mathcal{PCC}$ is not optimal when used to schedule task sets with long and short tasks whose release times are arbitrary. There are two reasons: (1) the unexpected delay a task experiences on later processors when it is preempted, and (2) inconsistency of timing constraints when release times are arbitrary.

The first reason was discussed earlier in Section 4.3.2.2. Figure 4.9 shows that Algorithm pEEDF (and therefore Algorithm $\mathcal{PCC}$) is not optimal for scheduling identical-length task sets with arbitrary release times. Since identical-length task sets are a special case of task sets with long tasks and short tasks, Algorithm $\mathcal{PCC}$ is not optimal for the latter task sets either.



(a) $\mathcal{PCC}$ Schedule with Inconsistent Deadlines.



(b) $\mathcal{PCC}$ Schedule with Adjusted Deadlines.

Figure 4.13: Effect of Inconsistent Deadlines.

The second reason for Algorithm $\mathcal{PCC}$ not being optimal for arbitrary release times is the inconsistency of timing constraints. By a set of timing constraints being inconsistent, we mean that at least one given timing constraint (a release time or a deadline) does not reflect the actual timing constraint that task must meet in order to complete in time. In all feasible schedules, the task starts after the timing constraint, if it is a release time, or finishes before it, if it is a deadline. For example, Figure 4.13a shows a task set whose timing constraints are not consistent. The latest point in time by which task T_{11} must complete on P_1 is $t = 1$, which is well before the effective deadline $d_{11} = 11$. If T_{11} completes after time $t = 1$, the task T_1 can not finish before its deadline d_1, because all processors are used by T_3. To make the set of deadlines consistent, d_{11} must be adjusted from $d_{11} = 11$ to $d_{11} = 1$, and d_{12} from $d_{12} = 13$ to $d_{12} = 2$. As illustrated in Figure 4.13a, because the effective deadline d_{11} does not reflect the time by which of T_{11} must complete on P_1 in order for T_1 to complete in time, a wrong scheduling decisions is made. In Figure 4.13b the effective deadlines for T_1 have been adjusted. The scheduling decisions in Figure 4.13b are based on the adjusted effective deadlines. This causes T_1 to start before T_2 and to hence meet its deadline. The resulting schedule is feasible.

The difficulty is that there is no easy way to determine how effective deadlines should be adjusted. One form of consistency of timing constraints (called *internal consistency*) is formally defined by Garey and Johnson in [10]. This definition is used for the special case of unit-length tasks on two processors with dependencies. Garey and Johnson reduce this two-processor scheduling problem to the problem of adjusting the deadlines to make them internally consistent, and then scheduling the tasks in order of increasing adjusted deadlines. Similarly, the use of *forbidden regions* (defined by Garey et al. in [13] and mentioned before in Section 4.1), during which tasks are not allowed to start execution, is a way to resolve inconsistencies of release times. Unfortunately, ways to find consistent timing constraints are only known for systems with unit-length tasks and one or two processors.

These considerations about why Algorithm $\mathcal{PCC}$ is not optimal lead us directly to a heuristic algorithm, Algorithm $\mathcal{HPCC}$, that we use to schedule task sets with arbitrary task parameters containing short tasks and long tasks. Algorithm $\mathcal{HPCC}$, described in Figure 4.14, is based on Algorithm $\mathcal{PCC}$. Tasks are scheduled according to the preemptive EEDF algorithm on each processor. The release times of tasks on each processor are either the end-to-end release times of the tasks, on the first processor, or the finishing time on the earlier processor, for the remaining processors. The preemptability of tasks is restricted, however. By restricting the preemptability, we mean that only short tasks are allowed to preempt other tasks. Long tasks are not allowed to preempt any task. In this way we deal with the earlier mentioned problem of unexpected delays on later processors when a task is preempted. The delay of the currently executing task caused by preemption is limited to the processing time of a short subtask per each preemption. The penalty incurred by the preempted task is therefore limited.

Algorithm $\mathcal{HPCC}$ does not consider inconsistencies of timing constraints. However, the effect of these inconsistencies on preemptive schedules is much smaller. In one-processor systems with no dependencies, for example, inconsistent deadlines can cause a non-preemptive EDF scheduler

Algorithm $\mathcal{HPCC}$:

Input: Task parameters r_i, d_i and τ_{ij} of $\mathcal{T} = \mathcal{T}_S + \mathcal{T}_L$. $\mathcal{T}_S$ and $\mathcal{T}_L$ are sets of tasks whose subtasks have short and long processing times.

Output: A feasible schedule of $\mathcal{T}$, or the conclusion that feasible schedules of $\mathcal{T}$ do not exist.

Step 1: Determine the effective deadlines d_{ij} of all subtasks. Set r_{i1} to be r_i.

Step 2: On each processor P_j, starting from P_1 and continuing until P_m, do the following:

(1) Use restricted preemptive EEDF on P_j. Restrict the ability to preempt so, that no task can be preempted by a long task.

(2) For each $T_{ij} \in \mathcal{T}$, set the effective release time $r_{i(j+1)}$ of $T_{i(j+1)}$ to the time when T_{ij} completes on P_j.

Figure 4.14: Algorithm $\mathcal{HPCC}$ for Scheduling Task Sets with Short and Long Tasks.

to fail. They have no effect on a preemptive EDF scheduler, however. Therefore, we expect that inconsistencies of timing constraints are much less likely to cause Algorithm $\mathcal{HPCC}$ to fail than would be the case for a non-preemptive algorithm.

The performance of Algorithm $\mathcal{HPCC}$ was evaluated in a sequence of simulation experiments that compared the algorithm to a number of algorithms traditionally used to schedule task sets with timing constraints. In particular, we compared Algorithm $\mathcal{HPCC}$ to the algorithms described in Section 4.3.2.2. We used the method described there: We apply Algorithm $\mathcal{HPCC}$ to randomly generated task sets and measure the success rate and the relative performance of each algorithm.

The task sets that are used in the experiments described below are non-homogeneous and have the following common characteristics:

- The task sets consist of long and short tasks.
- The subtasks of each task have identical processing times. The processing times of subtasks in different tasks may not be identical.
- Short tasks have less laxity than long tasks.

The following simulation parameters are used to generate the task sets. Some of the parameters (n, m, μ_l, σ_l) have the same meaning as in the experiments in Section 4.3.2, whereas the meanings of some other parameters (ρ and σ_τ) are slightly different. Other parameters, such as s and p, have been added:

n: The *number of tasks* in the flow shop.

m: The *number of processors* in the flow shop.

s: The portion of tasks in the task set that are short. The task set contains $\lfloor s\,n \rfloor$ short tasks and $n - \lfloor s\,n \rfloor$ long tasks.

ρ: This parameter controls the *mean processing time* of long subtasks. The mean processing time of long subtasks is set to be ρI, where I denotes the range of the end-to-end release time of all tasks.

p: The ratio of the mean processing times of long subtasks to short subtasks. The larger p, the smaller the processing time of short subtasks.

σ_τ: Standard deviation of the normalized distribution of the processing time of the subtasks in a task. If σ_τ is zero, the generated task sets consist of two identical-length task sets. With large values for σ_τ, task sets generated may no longer consist of two distinguishable classes of (long and short) tasks.

μ_l: Mean laxity factor. This parameter controls the ratio of laxity to total processing time.

σ_l: Standard deviation of the normalized normal distribution of laxity factors.

All the task sets used in the following experiments are generated in the following way, that is similar to the method used in Section 4.3.2:

(1) The *end-to-end release times* r_i are uniformingly distributed in the range $[0, I]$:

$$r_i \sim U(0; I).$$

(2) The *processing time* τ_i of all subtasks of each task is chosen from one of two distributions:

$$\tau_i \sim \rho N_0(1, \sigma_\tau) \quad \text{if } T_i \text{ is a long task}$$

$$\tau_i \sim \frac{\rho}{p} N_0(1, \sigma_\tau) \quad \text{if } T_i \text{ is a short task.}$$

Parameter	Setting
n	4, 12, 20
m	12
s	0.25, 0.75
ρ	0.2
σ_τ	0.05, 0.2, 0.5
μ_u	0.2, 0.4, 0.6, 0.7
σ_l	0.5

Table 4.5: Settings for Simulation Parameters in Experiments.

(3) The *processing times* τ_{ij} of each subtasks is set to τ_i:

$$\tau_{ij} = \tau_i, \text{for all } i \text{ and } j$$

(4) The *end-to-end deadline* d_i of each task T_i is determined by adding the total processing time of the task and a *laxity* to the end-to-end release time. The laxity is determined by multiplying the total processing time of the task with a *laxity factor*. The laxity factor l_i for task T_i is first chosen from the normal distribution $N_0(1, \sigma_l)$ and then multiplied by the *mean laxity factor* μ_l:

$$l_i \sim \mu_l N_0(1, \sigma_l) = N_0(\mu_l, \mu_l \sigma_l).$$

We note that the laxity factors for long and short tasks are sampled from the same distribution. This generates short tasks with smaller amounts of laxity than long tasks.

The settings of the simulations parameters are shown in Table 4.5. The half-length of the 95% confidence intervals of the results are below 1.1% for both the relative performance and the success rate.

The plots of the relative performance and success rates summarizing the results of these experiments are in Appendix C. Figures C.1 to C.3 show the results of this experiment on a flow shop with 12 processors. The results show that Algorithm $\mathcal{HPCC}$ performs very well for task sets with large numbers of long tasks ($s = 0.25$), especially when laxity is scarce ($\mu_u = 0.6, 0.7$). As expected, Algorithm $\mathcal{HPCC}$ does not perform well when the variance of processing times is high (as for $\sigma_{\mathcal{T}} = 0.5$). These task sets no longer consist of long and short tasks, but look much more like task sets with arbitrary task parameters. For such cases the pEEDF algorithm performs better, and, when laxity is scarce ($\mu_u = 0.7$), Algorithm Ha performs best. Algorithm Ha was shown to perform well earlier in the experiments described in Section 4.3.2.2. The algorithms LLF, FCFS, and EEDF perform very bad throughout the experiment. For this reason, we do not include the results on their performance here.

The success rate of Algorithm $\mathcal{HPCC}$, shown in Figure C.4 to Figure C.6, are not as good as its relative performance would suggest. Throughout the experiments, the success rates of Algorithm $\mathcal{HPCC}$ are similar to that of the pEEDF algorithm. For the case of many short tasks in particular, the results are nearly identical. This is because Algorithm $\mathcal{HPCC}$ and the pEEDF algorithm behave identically for task sets that contain no long tasks. In general, the success rates of Algorithm HPCC are slightly lower than those of the pEEDF algorithm, except for task sets with little slack ($\mu_u = 0.6$ or 0.7) and little variance in processing time ($\sigma_{\mathcal{T}} = 0.05$). The good results for the relative performance show that Algorithm $\mathcal{HPCC}$ generates schedules with smaller total tardiness than Algorithm pEEDF.

We note that all algorithms have better results for task sets with large numbers of short tasks. This illustrates how occasional wrong scheduling decisions tend to have smaller effects when the tasks involved are small.

Chapter 5
Scheduling Flow Shops with Recurrence

In this chapter we present two algorithms to schedule identical-length task sets on flow shops with simple recurrence patterns. Specifically, we focus our attention on the case of *simple task sets*. By a simple flow shop with recurrence, we mean one where the visit sequence contains a single simple loop. The visit graph of such a visit sequence is depicted in Figure 2.1. An example of a flow shop whose visit sequence contains a loop is the control system described in Chapter 1 where the two communication links are replaced by a bus.

5.1 Identical Release Times

We extend the EEDF algorithm so that it can be used to optimally schedule simple task sets on flow shops with recurrence and identical release times. We show that a modified version of the EEDF algorithm, called *Algorithm* $\mathcal{R}$, is optimal for scheduling tasks to meet deadlines.

The key strategy used in Algorithm $\mathcal{R}$ is based on the following observation. If a loop in the visit graph has length q, the second visit of every task to a reused processor, corresponding to a node in this loop, should not be scheduled before $(q-1)\tau$ time units after the completion of its first visit to the processor. Let P_{v_l} be the first processor in the loop of length q. Let $\{T_{il}\}$ and $\{T_{i(l+q)}\}$ be the sets of subtasks that are executed on P_{v_l}. T_{il} is the subtask at the first visit of T_i to P_{v_l}, and $T_{i(l+q)}$ is the subtask at the second visit of T_i to the processor. $T_{i(l+q)}$ is dependent on T_{il}.

Algorithm $\mathcal{R}$ is described in Figure 5.1. The scheduling decision is made on a reused processor P_{v_l}, the first processor in the loop in the visit graph. Specifically, Step 1 uses the EEDF algorithm to schedule the set $\{T_{il}\}$. As the subtask T_{il} at the first visit is scheduled, the effective release time of the second visits is postponed whenever necessary, so that it will be scheduled to start no sooner than $(q-1)\tau$ time units after T_{il} completes. The subtask $T_{i(l+q)}$ is scheduled together with other subtasks on the processor according to the EEDF algorithm. In

Algorithm $\mathcal{R}$:

Input: Task parameters r_{ij}, d_{ij}, τ, of $\mathcal{T}$ and the visit graph $\mathcal{G}$. P_{v_l} is the first processor in the single loop of length q in $\mathcal{G}$.

Output: A feasible schedule $\mathcal{S}$ or the conclusion that the tasks in $\mathcal{T}$ cannot be feasibly scheduled.

Step 1: Schedule the subtasks in $\{T_{il}\} \cup \{T_{i(l+q)}\}$ on the processor P_{v_l} using the modified EEDF algorithm described below: the following actions are taken whenever P_{v_l} becomes idle

(1) If no subtask is ready, leave the processor idle.

(2) If one or more subtasks are ready for execution, start to execute the one with the earliest effective deadline. In both cases, when a subtask T_{il} (that is, the first visit of T_i to the processor) is scheduled to start its execution at time t_{il}, set the effective release time of its second visit $T_{i(l+q)}$ to $t_{il} + q\tau$.

Step 2: Let t_{il} and $t_{i(l+q)}$ to be the start times of T_{il} and $T_{i(l+q)}$ in the partial schedule $\mathcal{S}_R$ produced in Step 1. Propagate the schedule to the rest of the processors according to the following rules:

(1) If $j < l$, schedule T_{ij} at time $t_{il} - (l - j)\tau$.

(2) If $l < j \leq l + q$, schedule T_{ij} at time $t_{il} + (j - l)\tau$.

(3) If $l + q < j \leq k$, schedule T_{ij} at time $t_{i(l+q)} + (j - l - q)\tau$.

Figure 5.1: Algorithm $\mathcal{R}$ to Schedule Flow Shops with Single Simple Loops.

the second step, the partial schedule on processor P_{v_l} is propagated to the remaining processors by appending the execution of the remaining subtasks on both sides of the scheduled subtasks.

The following theorem states the optimality of Algorithm $\mathcal{R}$.

Theorem 5. For nonpreemptive scheduling of tasks in a flow shop with recurrence, Algorithm $\mathcal{R}$ is optimal, when the tasks have identical release times, arbitrary deadlines, identical processing times, and a visit sequence that can be characterized by a visit graph containing a single, simple loop.

Proof. By virtue of the effective release times and deadlines, whenever we can find a feasible schedule $\mathcal{S}_{v_l}$ of $\{T_{il}\}$ and $\{T_{i(l+q)}\}$ on processor P_{v_l} in Step 1, we can propagate it to the other processors and thus generate a feasible schedule for the entire flow shop. It is therefore sufficient for us to prove that Step 1, which generates a modified EEDF schedule $\mathcal{S}_R$ on the processor P_{v_l}, is optimal. We do so by showing that any feasible schedule $\mathcal{S}_{v_l}$ on P_{v_l} can be transformed into the schedule $\mathcal{S}_R$.

Suppose that the feasible schedule $\mathcal{S}_{v_l}$ of the subtasks $\{T_{il}\}$ and $\{T_{i(l+q)}\}$ on P_{v_l} is not a modified EEDF schedule. We can transform $\mathcal{S}_{v_l}$ into a modified EEDF schedule by repeatedly applying the steps described below. In the following description, we use the same term *swapping* defined in Section 4.4. Because all tasks have the same release times and identical processing times τ, the schedule $\mathcal{S}_{v_l}$ can be segmented into intervals of length τ, each of which is assigned to a single subtask. Instead of saying that we swap the two intervals that are assigned to two subtasks T_{aj} and T_{bj}, in the following we simply say that we swap the subtasks T_{aj} and T_{bj}.

The transformation process works as follows: we repeatedly scan the schedule $\mathcal{S}_{v_l}$ from the beginning until we reach the end of the schedule or a point t when we encounter one of the following conditions. We take the corresponding action when a condition is encountered.

(1) No subtask is scheduled to start at time $t - \tau$ and the subtask scheduled to start at time t is T_{al}: We schedule T_{al} to start its execution at time $t - \tau$.

(2) No subtask is scheduled to start at time $t - \tau$ and the task scheduled to start at time t is $T_{a(l+q)}$: We schedule $T_{a(l+q)}$ to start at time $t - \tau$ if $t - \tau \geq t_{al} + q\tau$, where t_{al} denotes the start time of T_{al}.

(3) T_{al} and T_{bl} are scheduled to start at times $t - \tau$ and t, respectively, and $d_{al} > d_{bl}$: We swap T_{al} and T_{bl}. If $T_{a(l+q)}$ is scheduled before $T_{b(l+q)}$, we swap these to subtasks.

(4) $T_{a(l+q)}$ and $T_{b(l+q)}$ are scheduled to start at times $t - \tau$ and t, respectively, and $d_{a(l+q)} > d_{b(l+q)}$: If T_{bl} is scheduled to start before $t-(q+1)\tau$, we swap $T_{a(l+q)}$ and $T_{b(l+q)}$. Otherwise we continue scanning.

(5) $T_{a(l+q)}$ and T_{bl} are scheduled to start at times $t - \tau$ and t, respectively, and $d_{a(l+q)} > d_{bl}$: We swap $T_{a(l+q)}$ and T_{bl}.

(6) T_{al} and $T_{b(l+q)}$ are scheduled to start at times $t - \tau$ and t, respectively, and $d_{al} > d_{b(l+q)}$: If T_{bl} is scheduled to start after $t - (q + 1)\tau$, we continue scanning. Otherwise, if T_{bl} is scheduled to start at or before $t - (q + 1)\tau$, we swap T_{al} and $T_{b(l+q)}$. If $T_{a(l+q)}$ is scheduled

to start at $t+(q-1)\tau$, its execution must be delayed by at least τ time units to keep the schedule valid. There are three possibilities:

(a) If no subtask is scheduled to start at time $t+q\tau$, $T_{a(l+q)}$ is delayed to start at time $t+q\tau$.

(b) If a subtask of T_{xl}, the first visit of some task T_x, is scheduled to start at time $t+q\tau$, we swap T_{xl} with $T_{a(l+q)}$.

(c) If a subtask $T_{x(l+q)}$, the second visit of some task T_x, is scheduled to start at time $t+q\tau$, we swap $T_{x(l+q)}$ with $T_{a(l+q)}$.

Each of the actions described above ensures that the schedule remains valid throughout the transformation. Obviously, the actions in cases (1) to (5) are possible, and they do not lead to an infeasible schedule; so are cases (6a) and (6b). Case (6c) is possible because for the original schedule to be valid, T_{xl} must be scheduled to start before $t-\tau$. T_{xl} is not schedule to start at $t-\tau$ or t, and $T_{x(l+q)}$ starts at $t+q\tau$. Therefore, there is enough time between the first and the second visit of T_x to processor P_{v_l} after $T_{x(l+q)}$ is swapped with $T_{a(l+q)}$.

Again, we repeatedly scan the schedule until no more transformation can be applied. In this transformation process we sort the subtasks according to their deadlines and eliminate idle times whenever possible. Therefore, the transformation process eventually comes to an end and generates a schedule $\mathcal{S}_{\mathcal{R}}$, that is, a modified EEDF schedule.

To complete the proof of that Algorithm $\mathcal{R}$ never fails to generate a feasible schedule, whenever such a schedule exists, we suppose that the Algorithm $\mathcal{R}$ fails to find a feasible schedule. In other words, the modified EEDF schedule produced in Step 1 is not feasible. Suppose that there is a feasible schedule $\mathcal{S}_{v_l}$ of $\{T_{il}\}$ and $\{T_{i(l+q)}\}$ on P_{v_l}. We can use the transformation process described above to transform $\mathcal{S}_{v_l}$ into a feasible schedule $\mathcal{S}_{\mathcal{R}}$ that is a modified EEDF schedule. This leads to a contradiction. Therefore, there exists no feasible schedule of $\{T_{il}\} \cup \{T_{i(l+q)}\}$ on P_{v_l} and no feasible schedule of $\{T_i\}$ when the Algorithm $\mathcal{R}$ produces an infeasible schedule. □

The example given in Table 5.1 and Figure 5.2 illustrates Algorithm $\mathcal{R}$. The length of the processing time τ of all subtasks is one. The visit sequence is $V=(1,2,3,4,2,3,5)$, with the subsequence $(2,3,4,2,3)$ forming a single simple loop. Since processor P_2 is the first reused processor in the loop, the scheduling decision is made on P_2.

Tasks	r_i	d_i
T_1	0	8
T_2	0	9
T_3	0	10
T_4	0	12

(a) The Task Set.

$$V = (1, 2, 3, 4, 2, 3, 5)$$

(b) The Visit Sequence.

Table 5.1: Identical-Length Task Set with Identical Release Times.

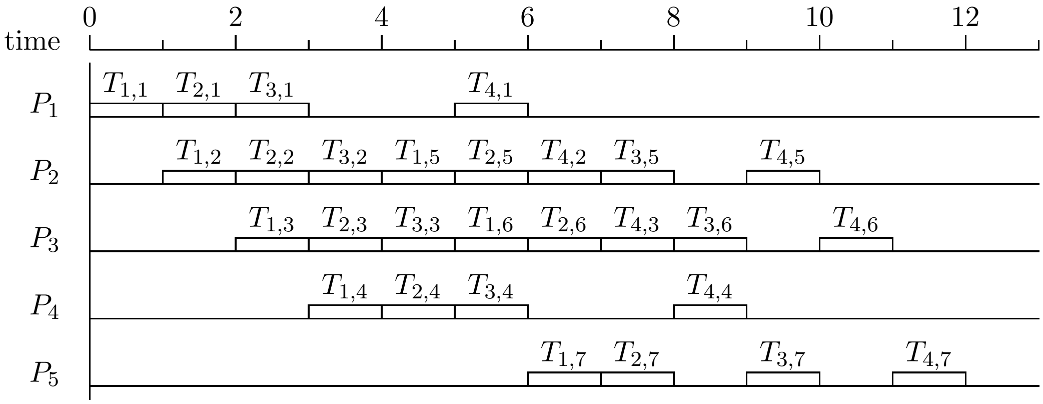


Figure 5.2: Schedule Generated by Algorithm $\mathcal{R}$.

We note that Algorithm $\mathcal{R}$ can also be used for task sets that have individual release times and an overall deadline. For this purpose, before applying Algorithm $\mathcal{R}$, the signs of the release time and the deadline of each task are negated, that is, we treat the release time as the deadline and the deadline as the release time. Also, we reverse the dependencies, that is, $T_{i(m-1)}$ depends on T_{im}, $T_{i(m-2)}$ depends on $T_{i(m-1)}$, and so on. In this way a mirror image of the original task set is generated. The schedule is then generated 'backwards' starting from the original deadline. This is explained by the example depicted in Table 5.2 and Figure 5.3. All tasks in the task set $\mathcal{T}$ in Table 5.2 have identical deadlines, but arbitrary release times. Changing the sign of release times and deadlines and reversing the dependencies generates the mirror image task set $\mathcal{T}'$, in which all tasks have identical release times $r'_i = -d_i = -20$. Figure 5.3a shows the resultant schedule $\mathcal{S}'$ after Algorithm $\mathcal{R}$ has been applied to task set $\mathcal{T}'$. The schedule $\mathcal{S}'$

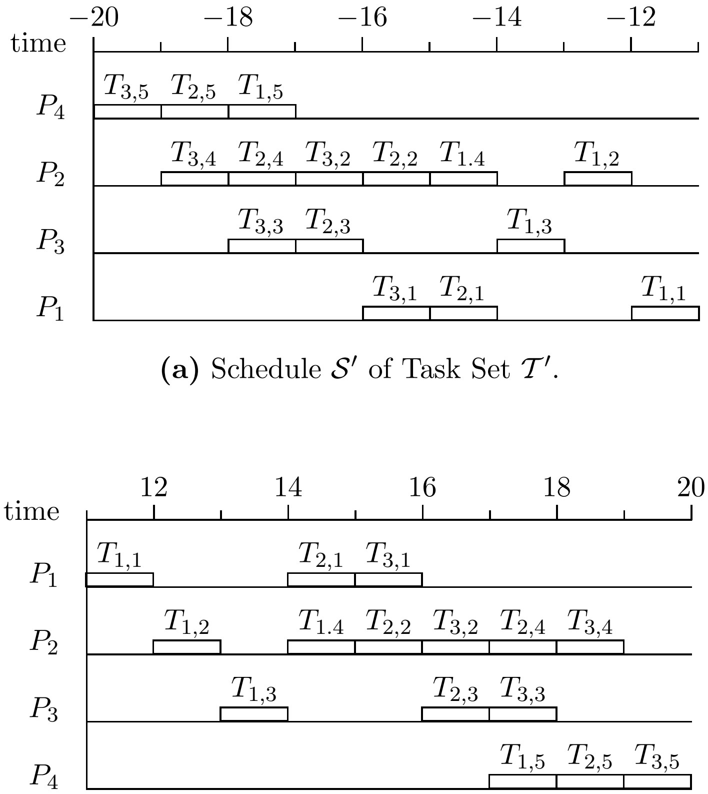
Tasks	r_i	d_i
T_1	11	20
T_2	14	20
T_3	15	20

$$V = (1, 2, 3, 2, 4)$$

(a) The Task Set.

(b) The Visit Sequence.

Table 5.2: Identical-Length Task Set with Arbitrary Release Times and Identical Deadlines.


−20
−18
−16
−14
−12
time
P_4
$T_{3,5}$ $T_{2,5}$ $T_{1,5}$
P_2
$T_{3,4}$ $T_{2,4}$ $T_{3,2}$ $T_{2,2}$ $T_{1.4}$ $T_{1,2}$
P_3
$T_{3,3}$ $T_{2,3}$ $T_{1,3}$
P_1
$T_{3,1}$ $T_{2,1}$ $T_{1,1}$
12
14
16
18
20
time
P_1
$T_{1,1}$ $T_{2,1}$ $T_{3,1}$
P_2
$T_{1,2}$ $T_{1.4}$ $T_{2,2}$ $T_{3,2}$ $T_{2,4}$ $T_{3,4}$
P_3
$T_{1,3}$ $T_{2,3}$ $T_{3,3}$
P_4
$T_{1,5}$ $T_{2,5}$ $T_{3,5}$

(a) Schedule $\mathcal{S}'$ of Task Set $\mathcal{T}'$.

(a) The Final Schedule $\mathcal{S}$.

Figure 5.3: Applying Algorithm $\mathcal{R}$ to Task Set with Arbitrary Release Times and Identical Deadlines.

meets all deadlines d'_i of the task set $\mathcal{T}'$. The schedule $\mathcal{S}$ of the original task set $\mathcal{T}$ is generated by changing the sign of the start times in $\mathcal{S}'$ of the subtasks, effectively generating the mirror image of the schedule $\mathcal{S}'$. Figure 5.3b shows the final schedule $\mathcal{S}$ of the original task set $\mathcal{T}$. All release times and deadlines are met in this schedule.

The problem of scheduling the subtasks on the reused processor P_{v_l} in Step 1 of Algorithm $\mathcal{R}$ is a variation of the problem of scheduling in-trees with separation constraints described by Han in [19]. In Han's problem, a set of unit-processing time tasks, whose dependency graph consists of a set of in-trees, is to be scheduled on one processor in a way that (1) the individual deadlines of the root tasks are met, and (2) whenever T_i is dependent on T_j, T_i must be executed k or more time units after T_j is executed. k is called the *minimum separation* and is a parameter of the problem. Han solves this problem in [19] with a vertex labeling approach in the dependency graph. We note that our problem of scheduling on a reused processor can be formulated as a scheduling problem with separation constraints where the in-trees are chains of two subtasks, with the minimum-distance constraint k equal to the length of the loop.

Both the problem of scheduling on a reused processor and Han's problem of scheduling with separation constraints are related to a wide class of multiprocessor and pipeline scheduling problems that can be formulated as *separation problems* [29]. The separation problem, as well as its dual problem, the *bandwidth-minimization problem*, is defined in the following way: Given an arbitrary graph, find a one-to-one labeling of the vertices, so that the difference between the labelings of any two adjacent vertices is at least k. The problem of scheduling on a reused processor with identical release times can be easily formulated as a separation problem.

5.2 Non-Identical Release Times

Algorithm $\mathcal{R}$ is optimal only if all the tasks in the identical-length task set have identical release times or identical deadlines. This is because the modified EEDF algorithm used in Step 1 of Algorithm $\mathcal{R}$ fails to optimally schedule the subtasks on the reused processor when the subtasks have individual release times and deadlines. In the following, we describe Algorithm $\mathcal{RR}$ for scheduling task sets with individual release times and deadlines, when both release times and deadlines are integer multiples of $q\tau$, where q is the length of the loop. Algorithm $\mathcal{RR}$ is described in Figure 5.4. It is similar to Algorithm $\mathcal{R}$. However, it does not use the modified EEDF algorithm in Step 1, but an algorithm that optimally schedules the subtasks with individual release times and deadlines on the reused processor.

Algorithm $\mathcal{RR}$:

Input: Task parameters r_i, d_i, τ, of $\mathcal{T}$ and the visit graph $\mathcal{G}$. P_{v_l} is the first processor in the single loop of length q in $\mathcal{G}$. The r_{iv_l}'s and d_{iv_l}'s are multiples of $q\tau$.

Output: A feasible schedule $\mathcal{S}$ or the conclusion that the tasks in $\mathcal{T}$ cannot be feasibly scheduled.

Step 1: Schedule the subtasks in $\{T_{il}\} \cup \{T_{i(l+q)}\}$ on the processor P_{v_l} using the following algorithm:

(a) Transform the set of subtasks $\{T_{il}\} \cup \{T_{i(l+q)}\}$ in the following way:

(i) Multiply the processing time of each subtask by q. Each subtask has now the processing time $q\tau$.

(ii) Define each subtask $T_{i(l+q)}$ to be dependent from T_{il}. By this we mean that $T_{i(l+q)}$ can only start executing once T_{il} is terminated.

We call the resulting task set $\tilde{\mathcal{T}}_{v_l}$, and each modified subtask $\tilde{T}_{il}$ or $\tilde{T}_{i(l+q)}$.

(b) Find a feasible q-processor schedule for the task set $\tilde{\mathcal{T}}_{v_l}$. This can be done by using Algorithm $\mathcal{F}$ described later in this section. Call the resultant schedule $\tilde{\mathcal{S}}_{v_l}$. If no feasible schedule can be generated, stop: no feasible schedule exists. Otherwise, go to Step (c).

(c) Scan the schedule $\tilde{\mathcal{S}}_{v_l}$ from left to right. For each time interval of length $q\tau$ in the schedule $\mathcal{S}$, determine which modified tasks $\tilde{T}_{ij}$ execute on the q processors in $\tilde{\mathcal{S}}_{v_l}$. Schedule the corresponding subtasks T_{ij} (of length τ) on P_{v_l} so, that no two visits of T_i to P_{v_l} are scheduled within $(q-1)\tau$ time units of each other. This is done by first executing the first visits to P_{v_l}, and after that the second visits. The second visits $T_{i(l+q)}$ to P_{v_l} are scheduled in the same order as their first visits were schedule earlier. The resultant schedule on P_{v_l} is called $\mathcal{S}_{\mathcal{RR}}$.

Step 2: Propagate the partial schedule $\mathcal{S}_{\mathcal{RR}}$ produced in Step 1 to the remaining processors in the same way as described in Step 2 of Algorithm $\mathcal{R}$.

Figure 5.4: Algorithm $\mathcal{RR}$ to Schedule Flow Shops with Single Simple Loops.

This algorithm used in Step 1 of Algorithm $\mathcal{RR}$ consists of two parts: In a first part, the time line is partitioned into non-overlapping intervals of length $q\tau$, each of which starts at an integer multiple of $q\tau$. All subtasks on P_{v_l} are scheduled to execute during one of these intervals, such that (1) all subtasks meet their timing constraints, (2) no two subtasks belonging to the same task are assigned to the same interval, and (3) for every task, the second subtask does not start earlier than $q-1$ time units after the first subtask completes. This schedule, called $\mathcal{S}_{\mathcal{RR}}$

in Figure 5.4, is used as the basis in Step 2 of Algorithm $\mathcal{RR}$, which produces the schedules for the other processors in the system. To construct the schedule $\mathcal{S}_{\mathcal{RR}}$ in Step 1(a), we first transform the set $\{T_{il}\}\cup\{T_{i(l+q)}\}$ of subtasks on P_{v_l} as follows: For each subtask T_{il} (or $T_{i(l+q)}$), there is a corresponding subtask $\tilde{T}_{il}$ (or $\tilde{T}_{i(l+q)}$) with processing time $q\tau$. Its effective release time and deadline are the same as that of T_{il} (or $T_{i(l+q)}$). $\tilde{T}_{i(l+q)}$ depends on $\tilde{T}_{il}$, as $T_{i(l+q)}$ depends on T_{il}. We then schedule the transformed set $\{\tilde{T}_{il}\}\cup\{\tilde{T}_{i(l+q)}\}$ on q processors. This work is done in Step 1(b). Because the release times of all subtasks are integer multiples of $q\tau$ and the processing times of all subtasks $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$ are equal to $q\tau$, in the resultant schedule, called schedule $\tilde{\mathcal{S}}_{v_l}$ in Figure 5.4, all subtasks are scheduled to start at time instants that are integer multiples of $q\tau$ and are not preempted.

In Step 1(c), the schedule $\tilde{\mathcal{S}}_{v_l}$ is transformed into the schedule $\tilde{\mathcal{S}}_{RR}$ on the processor P_{v_l}. If the subtask $\tilde{T}_{ij}$ is scheduled to start from time t in $\tilde{\mathcal{S}}_{v_l}$, the corresponding subtask T_{ij} is assigned to the interval $[t, t+q\tau]$ in $\tilde{\mathcal{S}}_{RR}$. We schedule the corresponding task T_{ij} in the interval assigned to $\tilde{T}_{ij}$ Clearly, if $\tilde{T}_{ij}$ meets its timing constraints, the corresponding subtask T_{ij} also meets its timing constraints.

Since $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$ are dependent, they are not scheduled at the same time in schedule $\tilde{\mathcal{S}}_{v_l}$. T_{il} and $T_{i(l+q)}$ are therefore assigned to different intervals. After all the subtasks have been assigned, the subtasks in each interval are scheduled in such a way that for each task T_i the second visit $T_{i(l+q)}$ to P_{v_l} does not start earlier than $q-1$ time units after the first visit T_{il} completes. Since there are at most q subtasks assigned to each interval, it is always possible to construct such a schedule.

We now describe the algorithm used in Step 1(b). The problem of scheduling the subtasks $\{\tilde{T}_{il}\}\cup\{\tilde{T}_{i(l+q)}\}$ is the same as the problem of scheduling on q processors unit-processing time tasks that have integer release times and deadlines and whose dependency graph contains chains. This problem can be formulated as a network-flow problem. Algorithm $\mathcal{F}$ used in Step 1(b) is based on this formulation; it is described in Figure 5.6.

Before we generate the network, we partition the time line into a sequence of disjoint intervals $I_k = [t_k, t_{k+1}]$ according to the release times and deadlines of subtasks as follows: The release times and deadlines are sorted in increasing order. Let $t_1, t_2, \cdots, t_u$ for $u \leq 2n$ be the sequence of distinct time instants, where t_k is either a release time or a deadline, obtained by deleting duplicates in the sorted sequence. The end points of the interval I_k are t_i and t_{k+1}.

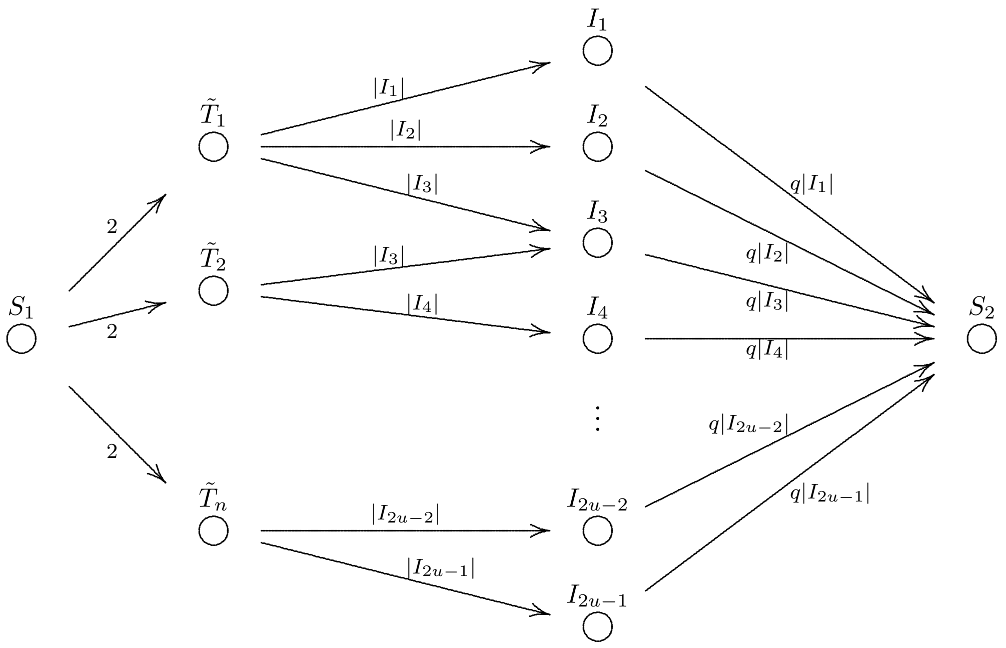


Figure 5.5: Network $G = (V, E)$.

The set V of vertices in the network $G = (V, E)$ consists of the following vertices:

(1) a source S_1 and a sink S_2.

(2) one vertex $\tilde{T}_i$ for each pair of subtasks $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$.

(3) one vertex I_k for each interval. Let I_k represent the time interval $[t_k, t_{k+1}]$. We denote the length $(t_{k+1} - t_k)/q\tau$ of the interval by $|I_k|$.

The set E of arcs is constructed as follows:

(1) The source S_1 is connected to each of the vertices $\tilde{T}_i$ by an arc with capacity 2.

(2) There is an arc connecting $\tilde{T}_i$ to I_k if $r_{il} \geq t_k$ and $d_{il} \leq t_{k+1}$, or if $r_{i(l+q)} \geq t_k$ and $d_{i(l+q)} \leq t_{k+1}$; in other words, when either $\tilde{T}_{il}$ or $\tilde{T}_{i(l+q)}$ can be scheduled during the interval I_k. The capacity of the arc is $|I_k|$.

(3) Each vertex I_k is connected to the sink S_2 by an arc of capacity $q|I_k|$.

The capacity of $|I_k|$ of the arc from $\tilde{T}_i$ to I_k denotes that $\tilde{T}_i$ can be executed for at most $|I_k|$ time units of length $q\tau$ during the interval I_k. The capacity of the arc from the vertex I_k to S_2

Algorithm $\mathcal{F}$:

Input: Task parameters r_i, d_i, of task set $\tilde{\mathcal{T}}$ consisting of pairs of tasks $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$ of length $q\tau$. The r_i' and d_i's are multiples of $q\tau$.

Output: A feasible schedule $\tilde{\mathcal{S}}$ on q processors or the conclusion that the tasks in $\tilde{\mathcal{T}}$ cannot be feasibly scheduled.

Step 1: Generate the sequence of intervals $I_1, I_2, \ldots, I_{2u-1}$ as described in Page 59.

Step 2: Generate the network $G = (V, E)$ as described on Page 59 from the task set $\tilde{\mathcal{T}}$.

Step 3: Find the maximum flow in the network G. If the flow is smaller than $2n$, stop: no feasible schedule exists. Otherwise, go to Step 4.

Step 4: Scan the intervals from I_1 to I_{2u-1}. For each interval scan all the incoming arcs e_i that carry a non-zero flow. If the flow is 1, schedule $\tilde{T}_{il}$ (or $\tilde{T}_{i(l+q)}$ if $\tilde{T}_{il}$ has been scheduled earlier) to start on any available processor as early as possible during the interval I_k. If the flow on e_i is 2, schedule $\tilde{T}_{il}$ on any available processor to start as early as possible during I_k, and schedule $\tilde{T}_{i(l+q)}$ to start on any available processor to start as late as possible during I_k. If at the end of this step some pairs of tasks execute simultaneously, separate them by swapping the $\tilde{T}_{i(l+q)}$ with any task scheduled to run later during I_k.

Figure 5.6: Algorithm $\mathcal{F}$ Used in Step 1(b) of Algorithm $\mathcal{RR}$.

enforces that no more than q processors are used during the interval I_k. Figure 5.5 shows the general form of the network $G = (V, E)$ constructed in the way described above.

To find a feasible schedule, we first try to generate a flow that equals $2n$. If no such flow exists, no feasible schedule exists either. The flow cannot exceed $2n$, because the capacity of the arcs that leave S_1 is $2n$. Once a flow of $2n$ is found, the schedule can be generated from the flows carried by the arcs between the vertices in $\{\tilde{T}_i\}$ and those in $\{I_k\}$ in Step 4 described in Figure 5.6. From each vertex $\tilde{T}_i$ there is either one arc e_i that carries a flow of 2 or two arcs e_{i1} and e_{i2} that carry a flow of 1 per arc. This can be safely assumed since the capacity of each arc from each vertex $\tilde{T}_i$ is equal to 1 or 2 and all integral network-flow problems are known to have integer solutions. Moreover, virtually all network-flow algorithms generate an integer solution [40, 52]. Consequently Step 4 is always possible. To generate the schedule $\tilde{\mathcal{S}}_{v_l}$ we scan the intervals $I_1, I_2, \cdots, I_{2u-1}$. During any interval I_k we look at the incoming arcs to I_k which carry a non-zero flow. Such an arc carries either a flow of 2, in which case both task $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$ execute during I_k, or a flow of 1, in which case only $\tilde{T}_{il}$ or $\tilde{T}_{i(l+q)}$ execute during I_k. By construction, every part of a task (either $\tilde{T}_{il}$ or $\tilde{T}_{i(l+q)}$) that executes during the interval I_k has $r_i \leq t_k$ and $d_i \geq t_{k+1}$, and therefore meets its timing constraints. To generate the schedule $\tilde{\mathcal{S}}_{v_l}$, we have to schedule the executions of the tasks in each interval I_k separately. In this we have to make sure that no two $\tilde{T}_{il}$ and $\tilde{T}_{i(l+q)}$ are scheduled during the same time instant. Step 4 of Algorithm $\mathcal{F}$ does this by scheduling the $\tilde{T}_{il}$'s at the beginning of the interval I_k and the $\tilde{T}_{i(l+1)}$'s at the end of the interval.

As described earlier, the q-processor schedule $\tilde{\mathcal{S}}_{v_l}$ is used in Step 1(c) of Algorithm $\mathcal{RR}$ to generate the schedule $\mathcal{S}_{\mathcal{RR}}$ on the reused processor.

Chapter 6

End-To-End Scheduling of Periodic Flow Shops

Each job J_i in a m-processor periodic flow-shop job set can be logically divided into m subjobs J_{ij}. The period of each subjob J_{ij} is p_i, the period of the job J_i. The subtasks in all periods of J_{ij} are executed on processor P_j and have processing times τ_{ij}. In other words, a set $\mathcal{J}_j$ of subjobs, whose members J_{ij} are characterized by p_i and τ_{ij}, is to be scheduled on each processor P_j. Each subjob J_{ij} is a sequence of subtasks that are invoked periodically. When it is necessary to distinguish the individual subtasks, the subtask in the k^{th} period (the k^{th} invocation) of subjob J_{ij} is called $T_{ij}(k)$. For a given i, the subjobs on different processors are dependent, since the subtask $T_{ij}(k)$ cannot begin until $T_{i(j-1)}(k)$ is completed. Unfortunately, there are no known polynomial-time optimal algorithms that can be used to schedule dependent periodic jobs to meet deadlines, and there is no known schedulability criteria to determine whether the jobs are schedulable. Hence, it is not fruitful to view the subjobs of each job J_i on different processors as dependent subjobs. In the approaches described in the following sections we consider the subjobs to be scheduled on all processors independent and schedule the subjobs on each processor independently from the subjobs on the other processors. We will describe how we effectively take into account the dependencies between subjobs of each job, so that $T_{ij}(k)$ never begins execution until $T_{i(j-1)}(k)$ is completed according to our schedules.

6.1 The Phase Modification Approach

Let b_i denote the time at which the first task $T_{ij}(1)$ becomes ready. b_i is called the *phase* of J_i. In particular, b_{i1} is the phase of the subjob J_{i1} of J_i on the first processor. Hence, the k^{th} period of the subjob J_{i1} begins at $b_{i1} + (k-1)p_i$. Suppose that we can schedule the subjobs on P_1 in such a way that we can be sure that every subtask $T_{i1}(k)$ is completed by the time $r_{ik} = b_i + (k-1)p_i + c_{i1}$. We call c_{ij} the *worst-case completion time* of the subjob J_{ij}. The worst-case completion time C_i of the job J_i is the sum of the worst-case completion times c_{ij} of the subjobs in J_i, that is, $C_i = \sum_{j=1}^{m} c_{ij}$. Now, we let the phase of every subjob J_{i2} of J_i

on processor P_2 be $b_{i2} = b_{i1} + c_{i1}$. By postponing the phase b_{i2} of every subjob J_{i2}, we delay the ready time of every subtask $T_{i2}(k)$ in every subjob J_{i2} on processor P_2 until its predecessor subtask $T_{i1}(k)$ is surely completed on processor P_1. In this way we can ignore the precedence constraints between subjobs on the two processors. Any schedule produced by scheduling the subjobs on P_1 and P_2 independently in this manner is a schedule that satisfies the precedence constraints between the subjobs J_{i1} and J_{i2}. Similarly, if every task in J_{i2} is guaranteed to complete by the time instant c_{i2} units after its ready time, we delay the phase b_{i3} of J_{i3} by this amount, and so on.

Suppose that the subjobs on each of the m processors are scheduled independently from the subjobs on the other processors, and all subtasks in J_{ij} complete by c_{ij} units of time after their respectively ready times, for all i and j. Moreover, suppose that $C_i = \sum_{j=1}^{m} c_{ij} \leq D_i$, where D_i is the deadline of J_i. We can delay the phase of each subjob J_{ij} on P_j by $c_{i(j-1)}$ units. The resultant schedule is a feasible schedule where all precedence constraints and all deadlines are met. We call this method of transforming dependent subjobs into independent subjobs the *phase modification* method. In the following sections we describe methods that allow to use existing schedulability criteria [26, 27, 31] to determine whether there is a set of $\{c_{ij}\}$ where $\sum_{j=1}^{m} c_{ij} \leq D_i$. The job system $\mathcal{J}$ can be feasibly scheduled if such a set of c_{ij}'s exists.

6.2 Phase Modification and Rate-Monotonic Scheduling

In this section we describe a simple way of applying phase modification to end-to-end scheduling of periodic flow shops based on the schedulability bounds provided by the rate-monotonic scheduling theory. Suppose that the set $\mathcal{J}_1$ of subjobs is scheduled on the first processor P_1 according to the well-known rate-monotone algorithm [31]. This algorithm is priority-driven; it assigns priorities statically to jobs (and, hence, to individual tasks in them) on the basis of their periods; the shorter the period of a job, the higher its priority. Without loss of generality we assume that the n jobs in the job system $\mathcal{J}$ have n different priorities. A job J_i having priority ϕ_i means that $\phi_i - 1$ jobs in the job system $\mathcal{J}$ have a higher priority than J_i. We index the jobs in order of decreasing priority, that is, $\phi_i = i$, with J_1 having the highest priority, and J_n the lowest.

Jobs	τ_{i1}	τ_{i2}	p_i	c_{i1}	c_{i2}	C_i
J_1	2	1	8	3.3	3.6	6.9
J_2	1	2	10	4.125	4.5	8.625
J_3	1	2	16	6.6	7.2	13.8

Table 6.1: Set of Periodic Jobs on a Two-Processor Flow Shop.

Equation (6.1) [27] gives the so called *schedulability bound* $u_{max}(n, \rho)$ on the total utilization $u_j(n) = \sum_{i=1}^{n} \tau_{ij}/p_i$ of the subjobs on each processor P_j; a set of subjobs whose total utilization is equal to or less than $u_{max}(n, \delta)$ is surely schedulable by the rate-monotone algorithm to complete within δp_i units after their ready time.

$$u_{max}(n, \delta) = \begin{cases} n((2\delta)^{1/n} - 1) + (1 - \delta), & \frac{1}{2} \leq \delta \leq 1 \\ \delta & 0 \leq \delta \leq \frac{1}{2} \end{cases} \tag{6.1}$$

Given the total utilization u_j on processor P_j we can find from Equation (6.1) $\delta_j = u_{max}^{-1}(n, u_j)$, where u_{max}^{-1} denotes the inverse of the function u_{max}. If we delay the phase of each subjob J_{ij} on P_j by $c_{ij} = \delta_j p_i$ units, and if $\sum_{j=1}^{m} \delta_i p_i \leq D_i$, the resultant rate-monotonic schedule if feasible and all precedence constraints and deadlines are met.

Table 6.1 shows an example of a set of three periodic jobs to be scheduled on a flow shop with two processors. We want to know if we can always complete every job by the end of its period. In this example we schedule the jobs on the processors using the rate-monotone algorithm. The total utilization factors of subjobs on processors P_1 and P_2 for the job set in Table 6.1 are $u_1 = 0.4125$ and $u_2 = 0.45$, respectively. By applying the formula in Equation (6.1) to the two processors, we get $\delta_1 = 0.4125$ and $\delta_2 = 0.45$. Therefore we know that $T_{11}(k)$ always terminates by time $c_{11} = \delta_1 p_1 = 3.3$ units, $T_{21}(k)$ by time $c_{21} = \delta_1 p_2 = 4.125$ units, and $T_{31}(k)$ by time $c_{31} = \delta_1 p_3 = 6.6$ units after their respective release times. We therefore delay the phases of the jobs J_{12}, J_{22}, and J_{32} on P_2 by 3.3, 4.125, and 6.6 units, respectively. On P_2, $T_{12}(k)$, $T_{22}(k)$, and $T_{32}(k)$ always complete by time $c_{12} = \delta_2 p_1 = 3.6$ units, $c_{22} = \delta_2 p_2 = 4.5$ units, and $c_{32} = \delta_2 p_3 = 7.2$ units, respectively, after their release times. Every invocation of J_1 is completed at or before time 6.9 units after its release time and therefore before the end of its period. Hence, it meets its deadline. The same holds for J_2 and J_3.

Jobs	τ_{i1}	τ_{i2}	p_i	c_{i1}	c_{i2}	C_i
J_1	5	5	10	0.553	0.553	1.106
J_2	0.5	0.5	10	0.553	0.553	1.106

Table 6.2: Unschedulable Set of Periodic Jobs on a Two-Processor Flow Shop.

Table 6.2 shows an example that is due to Lehoczky et al. [27]. The job set in this example can not always be scheduled so that both jobs meet their deadlines at the end of their respective periods. When the two jobs have the same phase, J_1 is interrupted to let J_2 execute and misses its deadline. The total utilization factors u_1 on P_1 and u_2 on P_2 are both 0.55 . The maximum utilization on each processor for the two jobs to be schedulable on a two-processor flow shop dropped to 0.5 from 0.83 in the single-processor case, and Lehozcky shows how the maximum utilization drops to $1/r$ for r processors. In this example we show that we can achieve a higher utilization bound if we allow the deadlines of the jobs to be delayed beyond the end of the period. By solving Equation (6.1) for δ_1, we deduce that the subjobs on P_1 can always complete within 0.553 times their periods. We therefore delay the phase of the subjobs J_{i2} by $0.553p_i$. A similar analysis for the subjobs on P_2 shows that all the jobs can complete within $1.106p_i$ time units. By postponing the deadlines of the jobs slightly more than 10% beyond the period, we can guarantee the job set to be schedulable.

We can refine the schedulability bounds for the rate-monotonic scheduling algorithm, and use the tighter schedulability condition to reduce the values of δ_i's in order to reduce the amount of delay necessary on successive processors and preserve the dependencies. One straightforward way to reduce these delays becomes evident from the following simple observation: If all the subjobs on a given processor are independent, subjobs with a lower-priority do not interfere with subjobs with a high priority. In calculating c_{ij} for a specific subjob J_{ij}, only J_{ij} and the subjobs with higher priorities, i.e. $J_{1j}, J_{2j}, \cdots, J_{(i-1)j}$ need to be considered. c_{ij} therefore becomes $c_{ij} = \delta_{ij}p_i$, where

$$\delta_{ij} = u_{max}^{-1}(i, u_j(i)),$$

and the function $u_j(i)$ denotes the total utilization of subjobs $J_{1j}, J_{2j}, \ldots, J_{ij}$ on P_j; in other words $u_j(i) = \sum_{k=1}^{i} \tau_{kj}/p_k$. The function $u_{max}^{-1}(i, u_j(i))$ is strictly decreasing with increasing priority i, giving us smaller values of c_{ij} for for high-priority jobs. Table 6.3 shows a comparison

					Basic Method				Refined Method			
Jobs	τ_{i1}	τ_{i2}	τ_{i3}	p_i	c_{i1}	c_{i2}	c_{i3}	C_i	c_{i1}	c_{i2}	c_{i3}	C_i
J_1	2	1	1	8	3.8	3.6	2.8	10.2	2	1	1	4
J_2	1	2	1	10	4.75	4.5	3.5	12.75	3.5	3.25	2.25	9
J_3	2	2	2	16	7.6	7.2	5.6	20.4	7.6	7.2	5.6	20.4

Table 6.3: Refined Schedulability Bounds for Rate-Monotonic Scheduling

Jobs	τ_{i1}	τ_{i2}	τ_{i3}	p_i	ϕ_{i1}	c_{i1}	ϕ_{i2}	c_{i2}	ϕ_{i3}	c_{i3}	C_i
J_1	1	1	1	10	1	1	1	1	1	1	3
J_2	1	1	1	12	2	2.2	2	2.2	2	2.2	6.6
J_3	1	1	5	14	3	3.567	3	3.567	3	7.592	14.726

Table 6.4: Unschedulable Job System with Rate-Monotonic Priority Assignment.

of the completion times for a job system on three processors. The jobs are scheduled according to the rate-monotonic algorithm with phase modification. When the basic method is used, no task appears to be schedulable. With the refined method, however, the higher-priority jobs J_1 and J_2 become schedulable, while the lowest-priority job J_3 remains not schedulable.

6.3 Phase Modification and General Fixed-Priority Scheduling

According to the rate-monotonic priority algorithm, all subjobs within a job have the same priority. This strategy may fail to schedule some job sets that are schedulable. Table 6.4 shows an example. This job system can not be scheduled by the rate-monotonic algorithm, even when the phase delays are set according to the refined method. We note that job J_3 consists of two short subjobs, J_{31} and J_{32}, followed by the very long subjob J_{33}. Intuitively, we would like to assign a higher priority to J_{31} and J_{32} in order to allow more time for J_{33} to execute before its deadline. In this section we describe an efficient method based on the deadline-monotonic approach to determine approximate upper bounds on the c_{ij}'s for arbitrary fixed-priority assignments. This method allows us to compute the worst-case completion times of the subjobs in different processors when they are assigned different priorities.

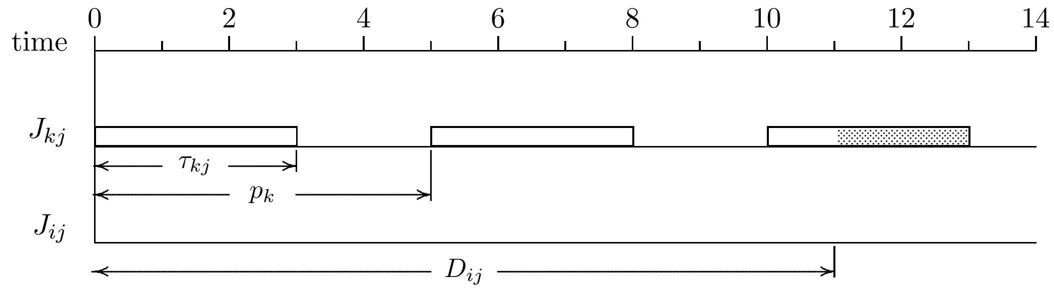


Figure 6.1: Higher-Priority Subjobs Interfering with J_{ij}.

The *deadline-monotonic* priority algorithm assigns priorities to jobs according to their deadlines D_i: The shorter the deadline, the higher the priority. This assignment is optimal for job systems in which the relative deadlines of jobs are shorter or equal to their periods, that is $D_i \leq p_i$ [2, 30]. (In the special case when the D_i's are proportional to the periods, this assignment is identical to the rate-monotonic assignment.) In [2] we are given a sufficient but not necessary schedulability condition for a job set on one processor with arbitrary deadlines $D_i \leq p_i$ to be schedulable according to the deadline-monotonic priority algorithm. Equation (6.2) is based on this condition. This equation states that, for subjobs on P_j to be schedulable, the sum of the processing time of each task and the processing times of the higher-priority tasks can not be larger than D_{ij}.

$$\forall i: \quad \tau_{ij} + I_{ij} \leq D_{ij} \tag{6.2}$$

where

$$I_{ij} = \sum_{k=1}^{i-1} \left(\left\lfloor \frac{D_{ij}}{p_k} \right\rfloor \tau_{kj} + min\left(\tau_{kj}, D_{ij} - \left\lfloor \frac{D_{ij}}{p_k} \right\rfloor p_k\right)\right). \tag{6.3}$$

Figure 6.1 illustrates the meaning of I_{ij}: The subjob J_{kj} has a higher priority than J_{ij}, and therefore interferes with J_{ij}'s execution. $D_{ij} = 11$. $\lfloor D_{ij}/p_k \rfloor = 2$. Up to time $t = 2p_k = 10$, J_{ij} gives up $2\tau_{kj}$ units of execution time to J_{kj}. From $t = 0$ to $D_{ij} = 11$, J_{ij} gives up $D_{ij} - 2p_k$ or τ_{kj} units of execution time, whichever is bigger. Equation (6.2) can be rewritten as follows:

$$\forall i: \quad \tau_{ij} + \sum_{k=1}^{i-1} \left(\left\lfloor \frac{D_{ij}}{p_k} \right\rfloor \tau_{kj} + min\left(\tau_{kj}, D_{ij} - \left\lfloor \frac{D_{ij}}{p_k} \right\rfloor p_k\right)\right) \leq D_{ij} \tag{6.4}$$

We approximate $min\left(\tau_{kj}, D_{ij} - \left\lfloor\frac{D_{ij}}{p_k}\right\rfloor p_k\right)$ by τ_{kj}, and $\left\lfloor\frac{D_{ij}}{p_k}\right\rfloor\tau_{kj} + \tau_{kj}$ by $\left(\frac{D_{ij}}{p_k}+1\right)\tau_{kj}$, and obtain the following condition:

$$\forall i: \quad \tau_{ij} + \sum_{k=1}^{i-1}\left(\frac{D_{ij}}{p_k}+1\right)\tau_{kj} \leq D_{ij} \tag{6.5}$$

From Equation (6.5) we can derive the following approximate bounds for the D_{ij}'s:

$$\forall i: \quad \frac{\sum_{k=1}^{i}\tau_{kj}}{1-\sum_{k=1}^{i-1}\frac{\tau_{kj}}{p_k}} \leq D_{ij} \tag{6.6}$$

In a feasible schedule, the deadline D_{ij} of a subjob J_{ij} is an upper bound on the completion time c_{ij} of the subjob. The bound on the deadline derived in Equation (6.6) can be used as an upper bound on the time by which the subjob J_{ij} is completed. We therefore derived the following approximation for the worst-case completion time c_{ij} of the subjob J_{ij}:

$$\forall i: \quad \frac{\sum_{k=1}^{i}\tau_{kj}}{1-\sum_{k=1}^{i-1}u_{kj}} = c_{ij}, \tag{6.7}$$

where u_{kj} is $\frac{\tau_{kj}}{p_k}$. Given an arbitrary priority assignment, Equation (6.7) gives the worst-case completion times c_{ij} of the subjobs on a single processor. This bound becomes intuitively clear when we view the expressions in it as those of supply and demand of time: The fraction of processor time available to the subjobs J_{ij} of job J_i on processor P_j is $1-\sum_{k=1}^{i-1}u_{kj}$. In a time period of length c_{ij} the supply of time is $c_{ij}\left(1-\sum_{k=1}^{i-1}u_{kj}\right)$. The demand of processing time in the same period of length c_{ij} is $\sum_{k=1}^{i}\tau_{kj}$. c_{ij} must be large enough so that the supply of time covers the demand for time. The process of comparing the supply and the demand of time to determine the schedulability of a system is known as *time demand analysis* [32, 43].

The bounds for the c_{ij}'s can now be used in combination with the phase modification method to determine the schedulability of periodic flow shops with arbitrary priority assignments. Given a priority assignment that assigns the priority ϕ_{ij} to the subjob J_{ij}, we can use the formula in Equation (6.7) to determine the worst-case completion time c_{ij} for each processor P_j separately, giving rise to the following conditions:

$$\forall i,j: \quad \frac{\sum_{\phi_{kj}\leq\phi_{ij}}\tau_{kj}}{1-\sum_{\phi_{kj}<\phi_{ij}}u_{kj}} = c_{ij}, \tag{6.8}$$

By assigning arbitrary priorities we gain some flexibility in scheduling. This is illustrated by the example in Table 6.5b, where three jobs are scheduled on three processors according to the priority assignments listed here. We call a priority assignment a *feasible priority assignment* for

Jobs	τ_{i1}	τ_{i2}	τ_{i3}	p_i	ϕ_{i1}	c_{i1}	ϕ_{i2}	c_{i2}	ϕ_{i3}	c_{i3}	C_i
J_1	1	1	1	10	1	1	1	1	1	1	3
J_2	1	1	1	12	2	2.2222	2	2.2222	2	2.2222	6.6667
J_3	1	1	5	14	3	3.6735	3	3.6735	3	8.5714	15.9184

(a) Rate-Monotonic Priority Assignment.

Jobs	τ_{i1}	τ_{i2}	τ_{i3}	p_i	ϕ_{i1}	c_{i1}	ϕ_{i2}	c_{i2}	ϕ_{i3}	c_{i3}	C_i
J_1	1	1	1	10	1	1	1	1	1	1	3
J_2	1	1	1	12	3	3.6207	3	3.6207	2	2.2222	9.4636
J_3	1	1	5	14	2	2.2222	2	2.2222	3	8.5714	13.016

(b) Example of a Feasible Priority Assignment.

Table 6.5: Priority Assignment that Guarantees Schedulability.

a given flow shop, if jobs are schedulable according to the assignment. For the same flow shop, the rate-monotonic priority assignment is not feasible, as was shown earlier in Table 6.4. In Table 6.5b the subjobs on P_3 are scheduled according to the rate-monotonic priority assignment. This is not the case on P_1 and P_2, where the job J_3 is assigned a higher priority than J_2. This assignment leads to shorter worst-case completion times c_{31} and c_{32} and leaves enough time for J_{33} to complete before the end of the period. Table 6.5a shows the same job system with the rate-monotonic priority assignment, but with the values for the c_{ij}'s computed by the formula in Equation (6.8). We note that the bounds on the completion time determined by Equation (6.8) are higher than the ones derived from Equation (6.1). This stems from the fact that c_{ij} in Equation (6.7) is an approximation for the bound on the completion time in Equation (6.2). Moreover, Equation (6.2) represents sufficient but not necessary conditions for the job system to be schedulable.

We now address the problem of finding a feasible priority assignment for a given periodic flow shop. By finding a feasible priority assignment we mean determining the value for the priority ϕ_{ij} of each subjob J_{ij}, so that the sum of the worst-case completion times c_{ij} for each job J_i are smaller than its deadline. Figure 6.2 defines the *feasible-priority-assignment* (FPA) problem more formally.

Problem FPA:

Given: A job system $\mathcal{J}$ with n jobs on a m-processor flow shop. Each job J_i has a deadline D_i.

Problem: Assign each subjob J_{ij} a priority ϕ_{ij} such that the following conditions hold:

$$\forall i, j: \quad \frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau_{kj}}{1 - \sum_{\phi_{kj} < \phi_{ij}} u_{kj}} = c_{ij}$$

and

$$\forall i: \quad C_i = \sum_{j=1}^{m} c_{ij} \leq D_i.$$

Figure 6.2: The FPA Problem.

Unfortunately, the FPA problem is $\mathcal{NP}$-complete. In order to prove the $\mathcal{NP}$-completeness of the FPA problem, let us look at the simple case where the flow shop has only two jobs. In this case, the worst-case completion time c_{1j} has one of two values; so has c_{2j}. Specifically, the worst-case completion time c_{ij} has the value $w_{ij\phi}$ if the subjob J_{ij} executes at priority ϕ.

Theorem 6. The FPA problem with two jobs is $\mathcal{NP}$-complete.

Proof. The proof is by reduction from Set-Partition [11]: Assume that we have a set $\mathcal{A}$ with elements $\{a_1, a_2, \cdots, a_m\}$. Each a_j has a weight α_j. Can we partition $\mathcal{A}$ into two sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ such that the elements in $\mathcal{S}$ and $\overline{\mathcal{S}}$ have the same sum of weights?

We reduce this problem to the following FPA problem with two jobs: Find a feasible priority assignment for a job shop with two jobs J_1 and J_2 on m processors with the following values $w_{ij\phi}$ and deadlines D_i: The values $w_{ij\phi}$ for the completion times are defined as follows:

$$w_{ij1} = \alpha_j, \quad w_{ij2} = 0.$$

The deadlines D_1 and D_2 are both given by

$$D_1 = D_2 = \frac{1}{2} \sum_{j=1}^{m} \alpha_j.$$

The solution of this priority assignment problem solves the set partitioning problem as follows: Assigning priority 1 to J_{1j} (and priority 2 to J_{2j}) means assigning a_j to the set $\mathcal{S}$, whereas assigning it priority 2 (and priority 1 to J_{1j}) means putting a_j into set $\overline{\mathcal{S}}$. Therefore, every priority assignment is isomorphic to a partitioning of the set $\mathcal{A}$ into two sets. The deadlines D_1 and D_2 force the elements in the two sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ to have the same sum of weights $\frac{1}{2}\sum_{j=1}^{m}\alpha_j$. □

Corollary 2. The FPA problem is $\mathcal{NP}$-complete.

Proof. The corollary follows by restriction. □

Fortunately, the following simple and efficient heuristic algorithms perform well in assigning priorities to the subjobs in order to meet the deadlines.

Global-Deadline-Monotonic (GDM): This algorithm assigns priorities in order of decreasing end-to-end deadlines. In other words, J_{ij} has a higher priority than J_{kj} if $D_i < D_k$. (We note that all subjobs within a job have the same priority.)

Laxity-Monotonic (LM): According to this algorithm, the priorities on processor P_j are assigned on the basis of the amounts of laxity that remain for the subjobs. Specifically, the *laxity* l_{ij} of the periodic job J_i after finishing on processor P_j is equal to

$$l_{ij} = D_i - \sum_{k=1}^{j} c_{ik} - \sum_{l=j+1}^{m} \tau_{il}.$$

The subjob J_{ij} has a higher priority than the subjob J_{kj} if $l_{ij} < l_{ik}$.

Relative-Laxity-Monotonic (RLM): This algorithm assigns the priorities of subjobs on processor P_j according to their relative laxities. The *relative laxity* of the periodic job J_i after finishing on processor P_j is defined as

$$\lambda_{ij} = \frac{l_{ij}}{p_i}.$$

The subjob J_{ij} has a higher priority than the subjob J_{kj} if $\lambda_{ij} < \lambda_{kj}$. We note that, in contrast to Algorithm LM, jobs with larger periods suffer a penalty under Algorithm RLM.

6.4 Phase Modification and General Distributed Systems

Many systems of practical interest cannot adequately be described as a flow shop, but can be modeled as a collection of flow shops that share one or more processors. An example is

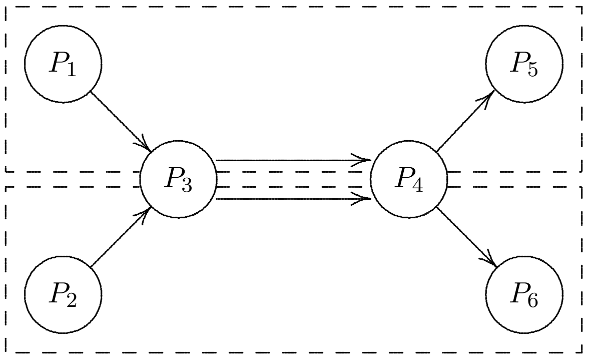


Figure 6.3: Collection of Flow-Shops.

shown in Figure 6.3. The system in this figure can be modeled by two different flow shops that share two processors, P_3 and P_4. The jobs in the first flow shop execute on P_1, then on P_3, then P_4, and P_5. The jobs in the second flow shop execute on P_2, P_3, P_4, and P_6. In other words, subjobs of jobs belonging to both flow shops execute on P_3 and P_4. The system in this example is clearly not a flow shop in the traditional sense. Nevertheless, we can use the phase-modification method to schedule such a system: The formulas in Equation (6.1) and Equation (6.8) can be directly applied to determine the worst-case execution time of a subjob on each of the processors. Since all the jobs are independent, only subjobs that are executing on the same processor influence the worst-case completion time of any particular subjob. Let A_j denote the subset of jobs that execute on the processor P_j. The formula for the worst-case completion time of jobs for arbitrary priority assignments ϕ_{ij} therefore becomes:

$$\forall i, j: \quad \frac{\sum_{\phi_{kj} \leq \phi_{ij}}^{J_k \in A_j} \tau_{kj}}{1 - \sum_{\phi_{kj} < \phi_{ij}}^{J_k \in A_j} u_{kj}} = c_{ij} \tag{6.9}$$

As a practical alternative, we can use the formula in Equation (6.8) by letting the processing times τ_{kj} of the subjob in J_k be zero if J_k does not execute on P_j.

A slight complication arises when dependencies between subjobs are not captured by the flow-shop model: An example is the system shown in Figure 6.4. In this simple system the subjobs J_{i2} and J_{i3} belong to the same job, but can execute concurrently. Such a system can not be modeled by a flow shop or a collection of flow shops. The worst-case completion times of individual subjobs can be determined in the same way as in systems that consist of collections of flow shops, for example with the formula in Equation (6.9). However, since some subjobs can

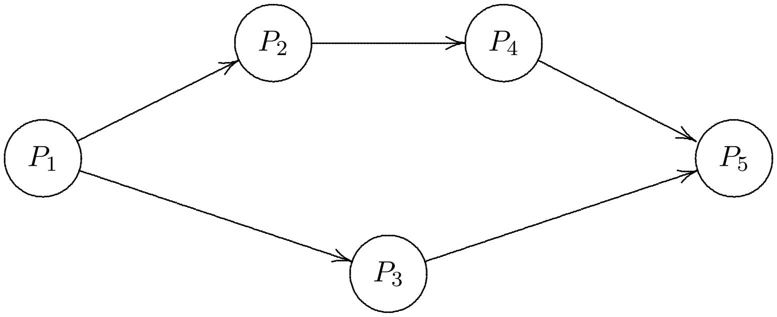


Figure 6.4: A System That Can Not Be Modeled as a Collection of Flow-Shops.

execute concurrently, the total worst-case completion time C_i of job J_i is not necessarily the sum of the worst-case completion times of its subjobs. The example in Figure 6.4 illustrates that the subjob $J_{i5}(k)$ starts execution when both subjobss $J_{i3}(k)$ and $J_{i4}(k)$ are completed. The worst-case completion time C_i therefore is

$$C_i = c_{i1} + max(c_{i2} + c_{i4}, c_{i3}) + c_{i5}.$$

In general, the worst-case total completion time of a job can be computed by first determining the longest path in the dependency graph of the system and then summing the worst-case completion times of subjobs along it. In determining the longest path, we sum the weights of vertices along each path in the dependency graph, where the weight of the vertex representing J_{ij} is the worst-case computation time c_{ij} of J_{ij}. The longest path is the one with the longest total weight. In the example in Figure 6.4 the longest path is (P_1, P_2, P_4, P_5) for a job J_i if $c_{i2} + c_{i4} > c_{i3}$; otherwise the longest path is (P_1, P_3, P_5).

In determining the schedulability of job systems that go beyond the periodic flow-shop model we appreciate the simplicity of the phase-modification method. With this method, the problem of scheduling systems of independent periodic jobs with arbitrary dependency constraints between subjobs in a distributed system is reduced to the simple problem of determining the critical path in a PERT-like graph [34, 44].

6.5 Phase Modification and Loosely Coupled Systems

The phase-modification method described in Section 6.1 assumes that all processors in the system have tightly synchronized clocks. This assumption is often not satisfied in distributed systems that are loosely coupled. Therefore, a question that must be resolved when applying this method to loosely-coupled distributed systems is how to account for the clock drift due to lack of synchronization.

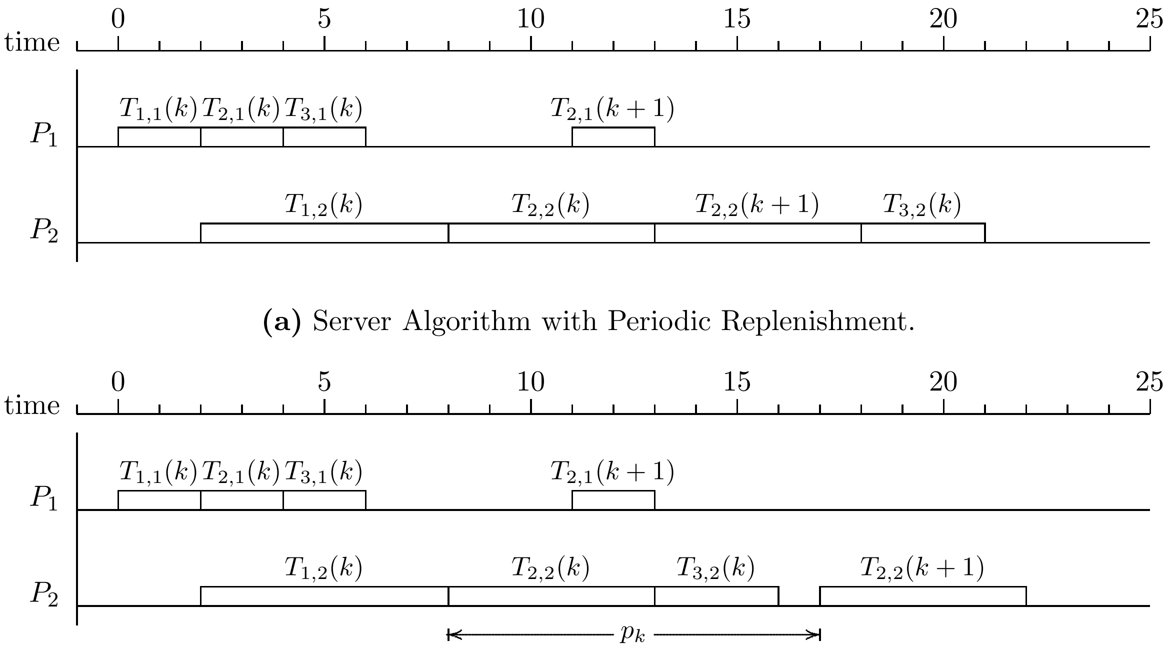
If clock synchronization cannot be guaranteed, phase modification can not assure that the dependencies between successive subtasks are preserved. Specifically, suppose that the clocks on the successive processors P_j and P_{j+1} in a flow shop are guaranteed to be synchronized within ϵ_j time units. Moreover, in an earlier step it was determined that the phase of $J_{i(j+1)}$ is offset by c_{ij} time units to preserve the dependencies between the subtasks $T_{ij}(k)$ and $T_{i(j+1)}(k)$ in the k^{th} execution of J_i. If the subjob J_{ij} has phase b_i, the resulting phase of $J_{i(j+1)}$ is $b_i + c_{ij}$. However, the clock on P_{j+1} can be ahead of the clock on P_j. Consequently, some subtask $T_{ij}(k)$ may terminate after $T_{i(j+1)}(k)$ starts executing on P_{j+1}, thus violating the dependency relation.

The above mentioned problem can be easily dealt with if the job system contains enough laxity. Given sufficient laxity, we can compensate for the clock drift in the following simple way: If the clocks of two successive processors P_j and P_{j+1} are guaranteed to be synchronized within ϵ_j time units, we can assure that the dependencies between subtasks on P_j and P_{j+1} are preserved by including the upper bound ϵ_j on the clock drift in the worst-case completion time of the subjob on P_j. The worst-case completion time of subtasks in J_{ij} is therefore given by the following formula:

$$c_{ij} = \frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau_{kj}}{1 - \sum_{\phi_{kj} < \phi_{ij}} \frac{\tau_{kj}}{p_k}} + \epsilon_j \tag{6.10}$$

We call this method to account for clock drift the *synchronous method.* We will later describe an asynchronous method and compare it to the synchronous method.

Another question that must be answered in order to apply the phase-modification approach in loosely-coupled systems is how to trigger the execution of the successor subtask $T_{i(j+1)}(k)$ on the next processor, once the current subtask $T_{ij}(k)$ is completed. In a tightly coupled system, we can use a time-driven mechanism: the subtask $T_{i(j+1)}(k)$ is released on P_{j+1} exactly c_{ij} time units after $T_{ij}(k)$ is released on P_j. In a loosely-coupled system such an approach is often not possible because of the lack of synchronicity. In such systems, tasks would have to be invoked asynchronously. For example, whenever a subtasks $T_{ij}(k)$ completes on processor P_j, a signal is sent to processor P_{j+1}, causing $T_{i(j+1)}$ to become ready for execution.



(a) Server Algorithm with Periodic Replenishment.

(b) Sporadic Server Avoids Deferred Execution Effects.

Figure 6.5: The Deferred Execution Problem in Flow Shops.

The example in Figure 6.5a illustrates the *deferred execution effect* [28], a problem that appears on all but the first processor in a flow shop where jobs are invoked asynchronously. $T_{ij}(k)$ and $T_{ij}(k+1)$ in this example are the k^{th} and $(k+1)^{th}$ invocations of subjob J_{ij}. The priorities are assigned in order of increasing job index, that is, $\phi_{1j} < \phi_{2j} < \phi_{3j}$. Figure 6.5a illustrates how the deferred-execution effect penalizes lower-priority jobs because of back-to-back executions of high-priority jobs. The execution of $T_{22}(k)$ on the second processor is deferred by the higher-priority task $T_{12}(k)$. When $T_{22}(k)$ completes, the lowest-priority task $T_{32}(k)$ is not allowed to start, because the next invocation of subjob J_{22}, $T_{22}(k+1)$ is ready. Since the execution budget of J_{22} is replenished periodically (in this case at time $t = 4$ and $t = 13$), J_{22} can inflict back-to-back hits to lower-priority subjobs; in this case, J_{32}'s execution is delayed by twice the length of subjob J_{22}. In some cases this may cause lower-priority jobs to miss their deadlines.

We describe here a method to schedule subtasks in each job asynchronously so that the problem due to deferred execution will not arise and the schedulability result produced by the phase modification technique remains correct. For this purpose we make use of results in scheduling of sporadic jobs on traditional single-processor systems. On all processors except P_1, every subjob J_{ij} that is asynchronously invoked after $J_{i(j-1)}$ terminates behaves like a sporadic job with mean execution time p_i. *Sporadic jobs* differ from their periodic counterparts in that their inter-release time is not constant, but has a statistical distribution. According to the *Sporadic Server* algorithm [50] to schedule sporadic jobs, we introduce a special periodic job, called a *server*, that is in charge of executing a sporadic job: During the time assigned to the server, the sporadic job served by it executes. A server job is characterized by its execution budget and a replenishment period p for the budget. The execution budget is the maximum amount of time the server can execute its sporadic job before the next replenishment. The replenishment time is set to the time when the invocation of a job starts, plus the replenishment period. If one invocation of a job is scheduled to start at time t, the budget is replenished at time $t + p$ where p is the replenishment period. In this approach, each subjob J_{ij} on such a processor is handled by one sporadic-server job with execution budget τ_{ij} and replenishment period p_i. With this simple mechanism the Sporadic Server algorithm guarantees that the execution time of the server job never exceeds the budget during a replenishment period.

The example in Figure 6.5 illustrates how sporadic servers can be used on processors where the executions of jobs are triggered asynchronously by the completion of the jobs on other processors. Figure 6.5b shows the example in Figure 6.5a with J_{22} scheduled according to the Sporadic Server algorithm: When $T_{22}(k)$ starts executing at time $t = 8$, the replenishment time is set to $t = 8 + p_2 = 17$. Although $T_{22}(k+1)$ becomes ready, there is no execution budget left. Because $T_{22}(k+1)$ now must wait until after the replenishment time, $T_{32}(k)$ is allowed to execute.

When the Sporadic Server method is used to schedule jobs asynchronously, we can safely apply the schedulability analysis that is based on phase modification to determine the worst-case completion times. In the worst case the schedule generated by the sporadic servers is identical to the schedule generated by the synchronous version of the phase-modification method. We have therefore shown that we can use the phase-modification method to determine the schedulability of loosely-coupled systems where jobs are periodic on the first processor and are invoked asynchronously on the remaining processors.

Earlier, we examined how the synchronous method accounts for clock drifts in synchronous systems: Such a system does not allow for asynchronous execution of subjobs. We have to delay the execution of subjobs on the subsequent processors long enough to include clock drift; otherwise, dependencies might be violated. When we allow for asynchronous execution of subjobs, lack of clock synchronization can not cause dependencies to be violated. However, clock drift must be accounted for in the schedulability analysis. Consider a subjob on J_{ij} with period p_i in a system with an upper bound ϵ_{j-1} on the clock drift between P_{j-1} and P_j, that is, the clocks of the processors P_{j-1} and P_j are guarateed to be synchronized within ϵ_{j-1}. Subjob J_{ij} can be modeled as a sporadic subjob with worst-case average inter-arrival time $p_i - \epsilon_{j-1}$. The subjob J_{ij} can be replaced by a server job with an execution budget of τ_{ij} and replenishment period $p_i - \epsilon_{j-1}$. Similarly, the other subjobs in the system can be replaced by their server jobs. Since this method relies on asynchronous execution of jobs, we call it the *asynchronous method* to account for clock drift. By using the asynchronous method, we have the following worst-case completion times:

$$\frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau_{kj}}{1 - \sum_{\phi_{kj} < \phi_{ij}} \frac{\tau_{kj}}{p_i - \epsilon_{j-1}}} = c_{ij}, \tag{6.11}$$

In some cases the asynchronous method can be used to generate better (that is, smaller) values for the c_{ij}'s than the synchronous method.

As an example, consider the special case of identical processing times $\tau_{ij} = \tau$, identical periods $p_i = p$, and a bound on the clock-drift rate γ so that $\epsilon_{j-1} = \gamma p$. With a given value for γ, the clocks differ by γp at the end of the period. For sake of simplicity, we assume that at the end of each period by the processors's local clock, the clock is synchronized. We determine the condition under which the value c_{ij} for a specific subjob J_{ij} achievable by the asynchronous method is better than the value c_{ij} achievable with the synchronous method:

$$\frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau}{1 - \sum_{\phi_{kj} < \phi_{ij}} \frac{\tau}{p(1-\gamma)}} < \frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau}{1 - \sum_{\phi_{kj} < \phi_{ij}} \frac{\tau}{p}} + \gamma p. \tag{6.12}$$

The four tables in Figure 6.6 show the results of a comparison between the two methods for each job in a job system with 20 jobs. Each table is for a different value for the clock-drift rate γ. The rows in a table are for different processing times of the jobs in relation to the period. Entries in each row in a table list the individual jobs in the job system. The high-priority jobs are on the left, and the low priority jobs on the right side. The processing time τ in Equation (6.12) is

u	Priority ϕ_{ij}																			
	1																			20
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.4	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+
0.6	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+
0.8	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+
1.0	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+	+

(a) $\gamma = 0.01$

u	Priority ϕ_{ij}																			
	1																			20
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.4	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.6	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+
0.8	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+
1.0	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+	+

(b) $\gamma = 0.05$

u	Priority ϕ_{ij}																			
	1																			20
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.4	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+
0.6	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+
0.8	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+
1.0	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+	—

(c) $\gamma = 0.10$

u	Priority ϕ_{ij}																			
	1																			20
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
0.4	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+
0.6	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+
0.8	—	—	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+
	—	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+	—
1.0	—	—	—	—	—	—	—	—	—	+	+	+	+	+	+	+	+	+	—	—

(d) $\gamma = 0.15$

— : Asynchronous method performs better.
+ : Synchronous method performs better.

Figure 6.6: Comparison of Synchronous and Asynchronous Method.

given by the parameter u, where $\tau = u\frac{p}{n}$, and n is the number of tasks. Again, in this example the number of tasks is $n = 20$. The parameter u happens to be the utilization of the processor, $u = n\frac{\tau}{p}$. A "+" in a given row and column means that the worst-case completion time is smaller when the synchronous method is used for a job with the given priority ϕ running on a processor with the given utilization u. A "-" means that, for the same job, the asynchronous method gives a smaller worst-case completion time.

The results show that the high-priority jobs have smaller worst-case completion times with the asynchronous method, whereas the synchronous method is better for low-priority jobs. With increasing processor utilization the portion of high-priority jobs for which the asynchronous method is better decreases. The results also show that variations in the clock drift have very little effect in this example. The portions of the jobs for which one or the other method is better changes very little with even large variations of γ. In general the question of which method is better must be investigated by experimental means.

Chapter 7

Periodic End-to-End Scheduling with Shared Resources

In the previous chapter we have been concerned with systems $\mathcal{J}$ consisting of independent jobs. When subjobs access common resources, and need arises for synchronization between subjobs, two problems must be addressed [47]: deadlocks and *uncontrolled priority inversion.* Priority inversion occurs when a higher-priority subjob waits while low-priority subjobs are executing. It is uncontrolled, since this condition may last for indefinite lengths of time. Several resource access protocols have been devised to avoid deadlock and to prevent uncontrolled priority inversion [3, 37, 47]. By using these protocols, the execution times of the subjobs can be made predictable. In the following we call one interval between the request and the release of a resource a *critical section.*

7.1 Local Resources

In this section we focus on the *priority ceiling protocol* (PCP) [47] to control access to local resources and describe how it can be used in distributed systems with end-to-end timing constraints. PCP is an extension of the *priority inheritance protocol*, according to which a low-priority subjob that blocks a high-priority subjob inherits its priority. In this way, the amount of time during which the high-priority subjob is blocked by the lower-priority subjob is limited. Simple priority inheritance is not enough to avoid deadlocks. PCP avoids deadlocks by using additional information about the resource requirements of subjobs: For each resource R, PCP keeps track of the highest priority of all tasks that will require R. This value is called the *priority ceiling* Π_R of the resource R. When a subjob J_{ij} requests a resource (enters a critical section), the request is only granted according to PCP if the following two conditions hold:

(1) R is not currently allocated to another subjob.

(2) The priority of J_{ij} is higher than all the priority ceilings of resources that are currently allocated to subjobs other than J_{ij}.

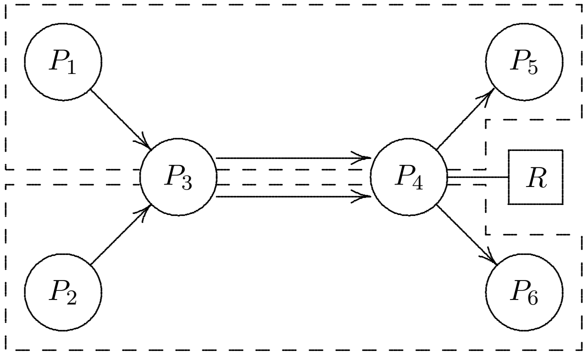


Figure 7.1: A System with One Local Resource.

When used in combination with any fixed-priority scheduling algorithm, PCP (1) prevents deadlocks, and (2) ensures that no subjob is blocked more than once by a lower-priority subjob during each critical section.

In its basic form, this protocol controls the access to local resources on a single processor. Consequently, in a first step we limit ourselves to applying PCP in distributed systems with resources that are local to processors. (In the next section we will address the issue of global resources.) Figure 7.1 shows an example of such a system: Two flow shops share processors P_3 and P_4. Moreover, there is a resource R on processor P_4. Except for the resource, this example is identical to the one depicted in Figure 6.3. The fact that resource R is on a shared processor is not important here. However, we must note that only subjobs running on P_4 can access R.

Let B_{ij} be the *blocking factor* of subjob J_{ij}. During any period, subjob J_{ij} is blocked for at most B_{ij} time units due to resource contention. We can therefore extend Equation (6.2) to determine the worst-case delay c_{ij} for subjob J_{ij}:

$$c_{ij} = \tau_{ij} + I_{ij} + B_{ij}, \tag{7.1}$$

where I_{ij} was defined earlier in Equation (6.3) to be the amount of time J_{ij} gives up execution to higher-priority tasks. After applying the same transformations as on Page 68, we derive the following approximation for the worst-case completion time c_{ij} of the subjob J_{ij}:

$$c_{ij} = \frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau_{kj} + B_{ij}}{1 - \sum_{\phi_{kj} < \phi_{ij}} u_{kj}} \tag{7.2}$$

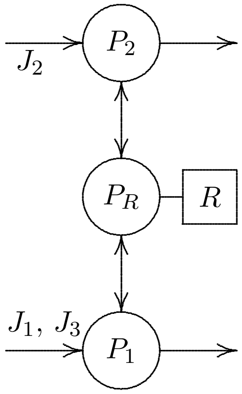
The bounds for the c_{ij}'s given by this expression can now be used in combination with phase modification to determine the schedulability of distributed systems in the same way as in the case of no resources.

7.2 Global Resources

In a multiprocessor system, resources can be local or global. By a *global* resource, we mean a resource that can be requested by subjobs that run on a processor that may be different from the processor that manages the resource. We call a critical section between the request and release of a global resource a *global critical section*, while a local resource gives rise to a *local critical section.*

Rajkumar et al. [42] have extended PCP to control the access to local and global resources in a multiprocessor environment. The *multiprocessor priority ceiling protocol* (MPCP), which we will describe at the end of this section, makes the following assumptions about the system: The processors are divided into two groups, *application processors* and *synchronization processors.* The application processors execute only application jobs. The synchronization processors manage the global resources. (They may also execute application jobs.) When a subjob on the application processor P_j requests a global resource R (enters a global critical section), it frees P_j and "migrates" to the synchronization processor P_R that manages the resource. The subjob executes on P_R as long as it holds the resource R. Once the subjob releases the resource (leaves the global critical section), it migrates back to P_j, where it continues its execution.

In our flow shop model we partition each subjob J_{ij} that accesses a global resource R into a sequence of subjobs $J_{ij}^{(1)}$, $J_{ij}^{(R)}$, and $J_{ij}^{(2)}$ with length $\tau_{ij}^{(1)}$, cs_{ij}, and $\tau_{ij}^{(2)}$, respectively. The subjobs $J_{ij}^{(1)}$ and $J_{ij}^{(2)}$ represent the execution of J_{ij} on the application processor before and after the request for resource R. $J_{ij}^{(R)}$ represents the critical section of J_{ij}, that is, the portion of J_{ij} that holds the resource R and executes on the synchronization processor. When $J_{ij}^{(R)}$ starts executing, it requests the resource R and releases it when it terminates. In other words, the sequence of the execution of $J_{ij}^{(1)}$, $J_{ij}^{(R)}$, and $J_{ij}^{(2)}$ is modeled by three subjobs. A system in which subjobs on different processors access the same global resource can be modeled as a collection of flow shops that all share the synchronization processor. We can therefore model the access to global resources as a flow shop problem with end-to-end timing constraints.



Jobs	Proc	$\tau_i^{(1)}$	cs_i	$\tau_i^{(2)}$	p_i
J_1	P_1	1	1	1	9
J_2	P_2	4	2	1	14
J_3	P_1	5	5	5	30

Figure 7.2: A System with One Global Resource.

By modeling a system with local and global resources in this way, every resource in the system is requested only by subjobs that are executing on the processor that is local to the resource. We eliminate the need for handling global resource acess differently from local resources. Hence, there is no need for a multiprocessor resource access protocol; the access to global resources can be managed by a combination of a single-processor protocol, such as PCP, and phase modification.

In order to compute the worst-case delays for the phase modification, we use Equation (7.2). A special consideration must be made when determining the worst-case delay of subjobs that access global resources. In order to compute $c_{ij}^{(1)}$ (or $c_{ij}^{(2)}$) of part $J_{ij}^{(1)}$ (or $J_{ij}^{(2)}$) of the subjob J_{ij}, we can assume that the other part $J_{ij}^{(2)}$ (or $J_{ij}^{(1)}$) does not interfere with its execution as long as J_{ij} completes before the end of its period. Therefore, we do not need to consider the other part $J_{ij}^{(2)}$ (or $J_{ij}^{(1)}$) when determining the worst-case delay $c_{ij}^{(1)}$ (or $c_{ij}^{(2)}$). The worst-case delay $c_{ij}^{(1)}$ is given by:

$$c_{ij}^{(1)} = \frac{\sum_{\phi_{kj} \leq \phi_{ij}} \tau_{kj} - \tau_{ij}^{(2)} + B_{ij}^{(1)}}{1 - \sum_{\phi_{kj} < \phi_{ij}} u_{kj}}, \tag{7.3}$$

where $B_{ij}^{(1)}$ denotes the blocking factor for the first part $J_{ij}^{(1)}$. $c_{ij}^{(2)}$ is computed similarly.

To illustrate this approach, we apply it to the simple system shown in Figure 7.2. This example illustrates how worst-case delays are determined for subjobs in systems with global resources. Figure 7.2 shows a simple system that consists of two application processors P_1 and P_2 and a synchronization processor P_R that manages the single global resource R. The three jobs J_1, J_3 (both running on P_1), and J_2 (running on P_2) access resource R. The jobs in this

Jobs	Proc	$\tau_i^{(1)}$	cs_i	$\tau_i^{(2)}$	p_i	$B_i^{(R)}$	$c_i^{(1)}$	$c_i^{(R)}$	$c_i^{(2)}$	C_i
J_1	P_1	1	1	1	9	5	1	6	1	8
J_2	P_2	4	2	1	14	5	4	9	1	14
J_3	P_1	5	5	5	30	0	9	10.72	9	28.72

Table 7.1: Schedulability Analysis Using Phase-Modification.

example are assigned priorities in order of increasing index, that is, $\phi_1 < \phi_2 < \phi_3$. The system depicted in Figure 7.2 could be a subcomponent of a larger system, in which case the jobs J_1, J_2, and J_3 could be subjobs of larger jobs. As long as P_1 and P_2 only execute these three jobs, and no other subjob accesses R, the schedulability analysis described below is also valid for the larger system.

In this example, each job J_i requests resource R after $\tau_i^{(1)}$ units of execution. The length of J_i's critical section is cs_i. After the resource is released, job J_i executes for another $\tau_i^{(2)}$ units of time before completing. The three jobs J_1, J_2, and J_3 are partitioned into three subjobs $J_i^{(1)}$, $J_i^{(R)}$, and $J_i^{(2)}$ for each $i = 1, 2, 3$. We note that the blocking factors of all the subjobs executing on the application processors P_1 and P_2 are zero. We determine the $c_i^{(1)}$'s (since in this example all jobs consist of one subjob only, we drop the second index j) by applying the formula Equation (7.3) to both $J_1^{(1)}$ and $J_3^{(1)}$ on P_1 and $J_2^{(1)}$ on P_2. Since J_1 and J_2 execute at the highest priority and access no resources on their application processors, they experience no delay by other jobs, thus having $c_1^{(1)} = \tau_1^{(1)}$, $c_2^{(1)} = \tau_2^{(1)}$, $c_1^{(2)} = \tau_1^{(2)}$, and $c_2^{(2)} = \tau_2^{(2)}$. J_3 executes at lower priority than J_1 on P_1 and its worst-case delays are $c_3^{(1)} = c_3^{(2)} = 9$. The worst-case delays $c_i^{(R)}$ on the synchronization processor P_R are computed using Equation (7.2). The blocking factor for $J_1^{(R)}$ and $J_2^{(R)}$ are equal to $cs_3 = 5$. Since $J_3^{(3)}$ runs at lowest priority, its blocking factor is zero. $J_1^{(R)}$ executes at highest priority and has a worst-case delay of $cs_1 + B_1^{(R)} = 6$. $J_2^{(R)}$ executes at the next lower priority, thus having a worst-case delay of

$$c_2^{(R)} = \frac{cs_1 + cs_2 + B_3}{1 - \frac{cs_1}{p_1}} = \frac{1+2+5}{8/9} = 9.$$

Similarly, $c_3^{(R)}$ is computed to be 10.72. The results in Table 7.1 show that all jobs complete by their deadline.

Table 7.2 lists the completion times when applying MPCP to the same example of Figure 7.2. MPCP uses PCP on every processor, independently of whether it is an application processor or a synchronization processor. In order to avoid preemption of subjobs that are granted a global resource by subjobs that are not, the priority ceiling Π_R of each global resource R must be higher than the highest priority π_{max} of any tasks in the system. When π_R is the highest priority of all tasks that require the global resource R, the priority ceiling of R is set to:

$$\Pi_R = \pi_R + \pi_{max} + 1.$$

When a subjob is executing on a synchronization processor after being granted the resource R, it does so with priority $\Pi_R = \pi_R + \pi_{max} + 1$.

The schedulability analysis of MPCP is the same as for the single-processor version of PCP. Equation (7.2) therefore is valid to determine the worst-case delays of subjobs that access global resources. The worst-case blocking factor B_{ij} is calculated differently, however. The value of B_{ij} is the sum of the following four quantities, which are described in detail in [42]. Let n_{ij}^G be the number of times subjob J_{ij} requests a global resource.

(1) *Local Blocking (LB):* The upper bound on the local blocking is $n_{ij}^G + 1$ times the duration cs_{ij}^L of the longest local critical section that can block J_{ij}.

$$LB_{ij} = (n_{ij}^G + 1) \times cs_{ij}^L$$

(2) *Global Blocking (GB):* The upper bound on the global blocking is n_{ij}^G times the duration cs_{ij}^G of the longest global critical section that can block a global critical section of J_{ij}.

$$GB_{ij} = n_{ij}^G \times cs_{ij}^G$$

(3) *Remote Blocking (RB):* Let A_r denote the set of subjobs that have a higher priority than J_{ij}, execute on another processor than P_j, and access a global semaphore. Each subjob $J_{kl} \in A_r$ can contribute to a maximum blocking time of $CS_{kl} \times \lceil p_i/p_k \rceil$. CS_{kl} is the duration that subjob J_{kl} spends within global critical sections when executing alone on the synchronization processor.

$$RB_{ij} = \sum_{J_{kl} \in A_r} CS_{kl} \times \lceil p_i/p_k \rceil$$

(4) *Deferred Blocking (DB):* Let A_l denote the set of subjobs that have a higher priority than J_{ij}, execute on the same processor P_j, and require a global resource. Each subjob

Jobs	Proc	$\tau_i^{(1)}$	cs_i	$\tau_i^{(2)}$	p_i	LB_i	GB_i	RB_i	DB_i	B_i	C_i
J_1	P_1	1	1	1	9	0	5	0	0	5	8
J_2	P_2	4	2	1	14	0	5	2	0	7	14
J_3	P_1	5	5	5	30	0	0	6	1	7	30.86

Table 7.2: Results of Schedulability Analysis Using MPCP.

$J_{kj} \in A_l$ can contribute to a maximum blocking time of $min(\tau'_{kj}, cs^G_{kj})$ where τ'_{kj} is the computation time after the first suspension of J_{kj} and cs^G_{kj} is the length of the longest global critical section of J_{kj}.

$$DB_{ij} = \sum_{J_{kj} \in A_l} min(\tau'_{kj}, cs^G_{kj})$$

The worst-case blocking factor B_{ij} is given by

$$B_{ij} = LB_{ij} + GB_{ij} + RB_{ij} + DB_{ij}.$$

The results of the schedulability analysis are shown in Table 7.2. Since all resources are global, all local blocking factors LB_i are zero. The maximum amount of time that the jobs J_1 and J_2 can be blocked on R by a lower-priority job is cs_3, which is 5. Since J_3 has the lowest priority among all the jobs in the system, its critical section can not be blocked by lower priority jobs. GB_3 is therefore zero. J_3 can experience remote blocking by J_2, which is executing on the other processor. RB_3 is $cs_2 \times \lceil p_2/p_3 \rceil$, which is equal to 6. J_2 on the other side can be remotely blocked by J_1 for 2 time units. Since J_1 has the highest priority among all the jobs in the system, it does not experience remote blocking, which gives $RB_1 = 0$. Among all the jobs, only J_3 can experience deferred blocking, because it is the only job that is executing on the same processor as a higher-priority job (in this case J_1). DB_3 is therefore equal to $min(\tau_1^{(2)}, cs_1) = 1$.

The worst-case completion times for the jobs in this example are computed using Equation (7.2). For example, C_3 is determined as follows:

$$C_3 = \frac{\tau_1^{(1)} + \tau_1^{(2)} + \tau_3^{(1)} + \tau_3^{(1)} + cs_3 + B_3}{1 - \frac{\tau_1^{(1)} + \tau_1^{(2)}}{p_1}} = \frac{24}{7/9} = 30.86\ .$$

The worst-case completion time for the other two jobs is computed similarly. According to this analysis, J_1 finishes 2 time units before the end of its period, and J_2 finishes exactly by the end of its period. J_3, however, fails to terminate by the end of its period. According to this analysis, the job system is therefore not schedulable.

Chapter 8
Summary and Conclusion

This thesis is concerned with the problem of scheduling tasks in distributed time-critical systems. In our model, a distributed system is a collection of tasks that execute on one or more processors. When a task executes on more than one processor, it is composed of a sequence of subtasks, each of which executes on one processor. The timing constraints of such composite tasks are end-to-end, meaning that we are not concerned with the time of execution of intermediate subtasks, as long as each overall task meets its timing constraints.

The simplest model of such a distributed system is the flow shop, in which tasks are composed of subtasks that all execute in turn on the same sequence of processors. The timing constraints of a task consist of a release time and a deadline. A schedule meets the timing constraints if the first subtask starts no earlier than the release time, and the last subtask completes no later than the deadline of the task. Based on the traditional flow shop model, we derived models for more general cases of distributed systems. The flow-shop-with-recurrence model allows us to represent systems in which processors are visited more than once by individual tasks. The periodic-flow-shop model allows us to represent systems of tasks that execute periodically on flow shops. Simple criteria to determine the schedulability of periodic workloads on single processors make it possible to easily determine the schedulability of periodic flow shops *a priori*, that is, without having to generate schedules and check them for feasibility. The schedulability analysis techniques presented in this thesis can be used to analyze the schedulability of large classes of distributed systems that cannot be modeled directly as periodic flow shops. These techniques can also be used for systems with resources, both local and global. Finally, resource access protocols can be simplified when systems are modeled and scheduled as distributed systems with end-to-end timing constraints.

8.1 Summary

In order to study the problem of scheduling tasks with end-to-end timing constraints in distributed systems, we started by investigating the scheduling of non-periodic tasks on traditional flow shops. We expanded the basic flow shop model by allowing recurrence and periodic invocations of tasks. The result is a set of scheduling algorithms and schedulability analysis techniques for a wide variety of distributed systems.

8.1.1 Traditional Flow Shop Scheduling

We have developed a series of efficient algorithms to schedule tasks in traditional flow shops to meet end-to-end timing constraints. Some of these algorithms (Algorithm $\mathcal{A}$ and Algorithm $\mathcal{PCC}$) have been proven optimal for special cases of flow shops. Other algorithms are heuristic (Algorithm $\mathcal{H}$, its expanded version Algorithm Ha, and Algorithm $\mathcal{HPCC}$) and have been shown to perform very well.

In Figure 8.1 we show the universe of all possible flow shops on a two dimensional space that is parameterized; on one axis according to the variance of the processing times on any processor and on the other axis according to the variance of the processing times between processors. When the variance in processing times per processor is small (the lower border of the diagram), task sets are homogeneous. On the other hand, if the variance in processing times between processors is small (the left border of the diagram), task sets consist of two or more sets of nearly identical-length tasks sets. The intersection of these two classes of task sets in the lower-left corner of the diagram forms the class of identical-length task sets. All flow shops along the lower border of the diagram can be optimally scheduled with Algorithm $\mathcal{A}$. Some flow shops along the left border of the diagram are optimally scheduled by Algorithm $\mathcal{PCC}$ (as long as the tasks have identical release times and the task set consists of only two identical-length task sets). We have shown that Algorithm $\mathcal{H}$ performs well for task sets with a small variance in processing times per processor. For tasks with a slightly higher variance, Algorithm Ha performs well. A task set that has a big variance in processing times between processors can sometimes be partitioned into two subsets: a subset of long tasks and a subset of short tasks. In this case, Algorithm $\mathcal{HPCC}$ works very well.

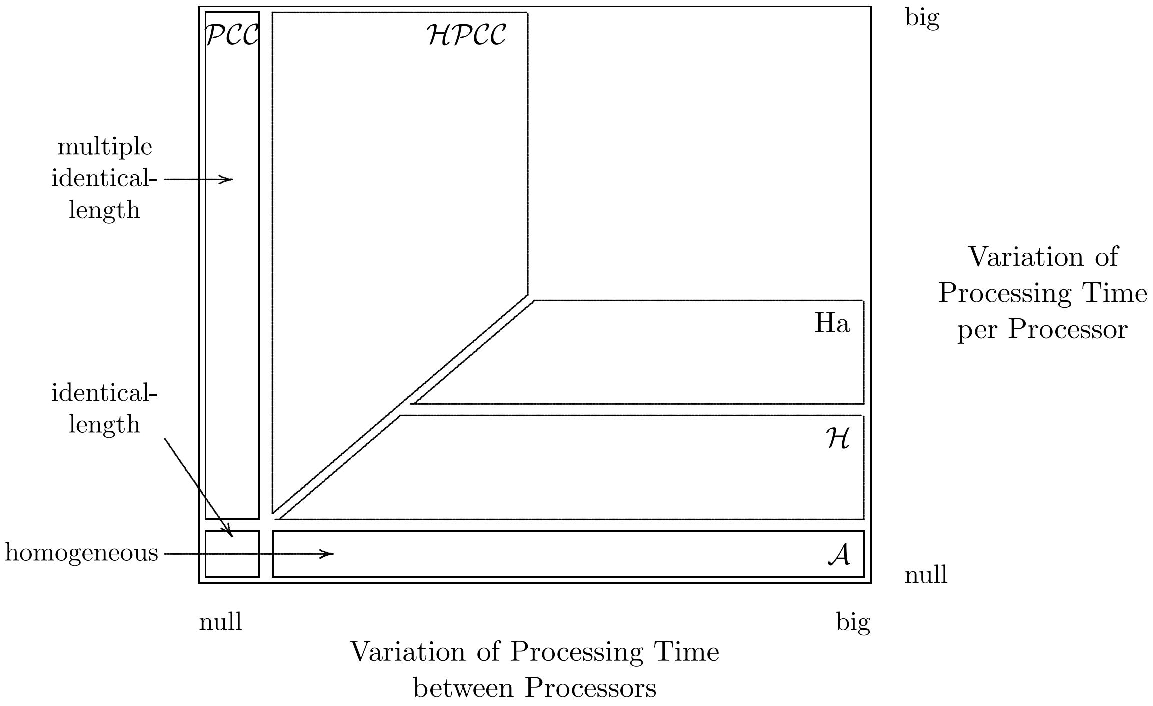


Figure 8.1: Different Algorithms for Different Flow Shops.

When both the variance in processing times per processor and between processors are large, our evaluations have shown that for flow shops with little laxity Algorithm Ha performs best. A simple preemptive deadline-driven scheme (as implemented in Algorithm pEEDF) is better for systems with more laxity.

8.1.2 Flow Shops with Recurrence

When tasks execute on the processors in the same order, but visit one or more processors more than once, the system can be modeled as a flow shop with recurrence. We have developed two efficient algorithms to schedule flow shops where all the subtasks have unit length and where one processor, or a sequence of processors, is visited twice. Algorithm $\mathcal{R}$ was proven to be optimal for the case where tasks have identical release times and arbitrary deadlines. When the release times are a multiple of the processing time between the two visits to the same processor, a second algorithm, Algorithm $\mathcal{RR}$, is optimal.

8.1.3 Periodic Flow Shops

When the workload consists of periodic jobs, a closed-form solution for the worst-case completion time of a subjob on a processor gives a lower bound on the worst-case length of time which the subjob on the next processor has to wait. By shifting the phase of the next subjob by this worst-case delay time, we guarantee that the dependencies between the invocations of the two subjobs are preserved. We call this method of delaying the execution of subjobs on successive processors *phase modification*. Phase modification can be used in combination with any scheduling algorithm for which we can compute bounds on the worst-case completion times of jobs. In this thesis, we introduced this technique in combination with the rate-monotonic algorithm, which has a simple formula for the worst-case completion time of subjobs. This worst-case completion time of a subjob on a processor depends on the utilization of the processor. Much tighter bounds on the completion times can be found when applying time demand analysis, using the formulas for deadline-monotonic scheduling. The bounds resulting from these formulas can also be used for arbitrary fixed-priority scheduling algorithms, even for algorithms that assign different priorities to different subjobs of the same job. Devising a fixed-priority algorithm to schedule tasks in a distributed systems becomes equivalent to assigning priorities to subjobs in order to meet constraints on sums of worst-case completion times.

Although we introduced the phase modification technique on periodic flow shops, it can be used on much more general systems, such as collections of flow shops sharing processors, or even systems with arbitrary dependency graphs. In order for phase modification to work in distributed systems, clock drift must be accounted for. This is easily done by increasing the bounds on worst-case completion times in order to separate the invocations of the subjobs on two successive processors enough to compensate for clock drift. When subjobs are executed asynchronously, we have shown that we can account for clock drift with smaller increases in worst-case computation time bounds, and therefore with smaller sacrifices of laxity. We showed that this holds especially for high-priority jobs or jobs on processors with a low utilization.

We have shown that resource access can easily be included in the phase modification technique. Resource-access protocols, such as the priority-ceiling protocol or the multiprocessor priority-ceiling protocol, give upper bounds on the time jobs are blocked waiting for resources. These bounds are directly used in the computation of the worst-case completion time bounds.

End-to-end scheduling in general, and phase modification in particular, are an alternative way to deal with global resource access. For this we partition jobs that access global resources into three parts: the portion before entering the critical section, the global critical section itself, and the portion after exiting the critical section. A global-resource-access protocol is no longer necessary, because the access to the processor that manages the global resource is scheduled according to an end-to-end scheduling algorithm. A local-resource-access protocol is still necessary on the processor that manages the resource to control the local access to the resource. We have shown that by eliminating the global resource access protocols we can achieve better bounds on the worst-case completion times. In particular, we have demonstrated by an example how we can guarantee the schedulability of a system with phase modification in combination with the priority ceiling protocol, while the multiprocessor priority-ceiling protocol fails. In addition to eliminating the need for global-resource-access protocols, end-to-end scheduling of accesses to global resources allows us to assign different priorities to the three different parts of every job accessing the resource, effectively giving more flexibility when scheduling the jobs.

8.2 End-to-End Schedulability Analysis

Part of the results in end-to-end scheduling of periodic workloads described in this thesis have been implemented as a part of PERTS, a prototyping environment for real-time systems, which is currently under development in the Real-Time Systems Laboratory at the University of Illinois [32, 33]. PERTS will contain schedulers and resource access protocols, in combination with tools and a simulation environment for the analysis and validation of real-time systems. PERTS is designed as a vehicle to evaluate new design approaches and to experiment with alternative scheduling and resource management strategies.

A central component in PERTS is the schedulability analyzer that allows the analysis of real-time systems built on the periodic-job model. One component of the schedulability analyzer is the *End-to-End Analysis*, where the user is provided with a picture of the jobs in the system, together with the information about the processor to which each job is assigned and the resources it accesses. The job system is represented as a directed graph where each connected component in the graph is interpreted as a job with the jobs in the component being its subjobs.

The user can choose to provide the release time and deadline of each subjob. Alternatively, only the end-to-end release times and deadlines are given; the analyzer then uses phase modification to assign the individual release times and deadlines of intermediate subjobs. It is possible to freely combine end-to-end timing constraints with individual timing constraints for some or all of the intermediate subjobs, either to satisfy system requirements, or to guide some (e.g. deadline-monotonic) scheduling algorithms.

8.3 Outlook

Many issues addressed in this thesis remain open or can be extended. For example, the problem of preemptively scheduling task sets that consist of multiple identical-length task sets on traditional flow shops has only been solved for identical release times and task sets that consist of two identical-length task sets. As another example, the complexity of scheduling identical-length task sets on flow shops with recurrence is not known for the case of arbitrary release times and deadlines.

Several general questions need to be further investigated for periodic systems. We showed that the problem of assigning priorities to subjobs in order to meet end-to-end timing constraints with fixed-priority scheduling is $\mathcal{NP}$-complete for very simple systems with two jobs. We have briefly described heuristic algorithms that can be used to determine good priority assignments. These algorithms must be further evaluated and compared.

Several further issues must be addressed if the techniques described here are to be successfully applied. First, processors in real-life systems typically have limited buffer sizes. The flow-shop model, both in the preemptive and non-preemptive case, assumes that jobs can be buffered at any processor to wait for other jobs to complete execution. Scheduling algorithms must be devised that allow for bounds on the number of preempted or waiting jobs on each processor.

Another issue that needs to be addressed is the problem of additions or deletions of jobs in the job system or changes in the parameters of certain jobs while the system is executing. In the single-processor case, such a change while the system is alive is called *mode change*. Protocols have been devised to ensure smooth execution of mode changes [48]. In distributed systems mode changes can have much more severe effects than on a single processor. Simple changes to job parameters, like increasing a single job's processing time, may require reassignment of priorities of many subjobs on one or more processors, or even migration of some subjobs to different processors. Protocols need to be devised that ensure smooth changes between modes, or that guarantee bounds on the amount of disruption caused by mode changes.

Throughout this thesis we have made the assumption that global and local status information about the system is available. In loosely-coupled systems this information may not be available or may not be kept current. The problem of end-to-end scheduling in such systems, where the schedulers may make incoherent decisions, remains to be investigated.

Appendix A
Heuristic-Supported Search for Flow-Shop Schedules

The problem of scheduling flow shops to meet end-to-end release times and deadlines is an example of the well known *Constraint Satisfaction Problem* (CSP) [11]. Enumerative approaches to solving CSPs rely on heuristic-supported exhaustive search methods. We describe a *depth-first backtrack search algorithm* to find feasible schedules for flow shops. In this approach, the schedule is generated by repeatedly selecting a subtask (selecting a *variable*, in CSP terms) and assigning it a start time (assigning it a *value*, in CSP terms). If the partial schedule that has been generated during the execution of the search cannot be completed without violating any constraint – be it a dependency constraint or a timing constraint – the search process has reached a *deadend* and has to *backtrack* by reversing one or more earlier assignments. The worst-case complexity of this search approach is exponential. However, several heuristics can typically be applied to reduce its average cost [45, 54]. We will formulate these heuristics as rules that guide the search process.

- **Consistency-enforcing rules**: To eliminate those alternatives from the search space that can not be part of any global solution (pruning of the search tree.)
- **Look-ahead rules**: To decide which subtask to select next and which start time to assign to it. We differentiate between
  - *Subtask-ordering rules*: Try to select a subtask first that is 'difficult' to schedule. This reduces the amount of backtracking.
  - *Start-time-ordering rules*: The start time assigned to the selected subtask should be expected to maximize the probability to successfully complete the partial schedule.
- **Look-back rules**: To speedily recover from deadends by possibly backtracking more than one assignment. Possibly learn from the past development of the search process.

Generally, there is a trade-off between the amount of effort invested in minimizing the number of generated search states and the effective savings in search time. In the following, we will describe a depth-first backtrack search algorithm to find solutions for the flow-shop problem. We will discuss heuristics that support the search process. For each rule we must prove its *correctness.* A rule is correct if it does not keep the search process from eventually generating a feasible global schedule. Look-ahead rules are by nature correct, since they only control the order of how the search space is traversed. Consistency-enforcing rules and look-back rules control which portions of the search space is traversed, and hence must be checked for correctness. We note that the correctness of a rule does no imply that the rule is usefull in speeding up the search process.

A.1 Searching for Feasible Schedules

The basic algorithm to search for feasible is described by *Procedure S* in Figure A.1. Procedure S recursively schedules the subtasks on a processor-by-processor basis. First, it tries to schedule all subtasks on the first processor. If it succeeds (in Step 4), it moves to the next processor and starts scheduling subtasks there, until it successfully schedules all the subtasks on the last processor (in Step 3) or declares failure. Let Q be the set of subtasks on the current processor P_j that have already been scheduled, and U the set of subtasks on P_j that have not been scheduled yet. In the following, t_{ij} denotes the start time and e_{ij} the completion time of T_{ij} in the partial schedule.

Procedure S gives a framework where additional rules can be inserted. Consistency-enforcing rules can be used in Step 2 to determine if the newly generated partial schedule can possibly be expanded into a feasible schedule. Subtask-ordering rules can be used in Step 4 to determine which subtask must be chosen next to be schedule on P_j. Look-back rules can be used during Step 2 to determine if the search should be continued at this point or if a backtrack is necessary. In the following, we describe examples of such rules that can be used to speed up the search process.

A.1.1 Consistency-Enforcing Rules

The goal of consistency-enforcing rules is to detect oncoming deadends early in the search process. The consistency-enforcing rule detects cases where the choice of a specific alternative can not possibly result in a feasible schedule and prevents the search algorithm from further

Procedure $S(j, i, Q, U)$:

Input: Set Q already scheduled subtasks on processor P_j. Index of one selected subtask T_{ij} that has not been scheduled yet. Set U of remaining subtasks on processor P_j. The following holds: $Q + T_{ij} + U = \mathcal{T}_j$, where $\mathcal{T}_j$ is the set of all subtasks on P_j.

Output: A feasible schedule, or the conclusion that no such schedule exists.

Step 1: Schedule the subtask T_{ij} to start at time $t_{ij} = max\{r_{ij}, max_{T_{lj} \in Q}\{e_{lj}\}\}$.

Step 2: If the partial schedule is feasible [CONSISTENT], go to Step 3. Otherwise, a deadend is detected; return [BACKTRACK].

Step 3: If $U - T_{ij} = \{\}$ and $j = m$, stop; a feasible schedule has been generated.

Step 4: If $U - T_{ij} = \{\}$, repeatedly select a subtask $T_{pj} \in U$ [SUBTASK-ORDERING RULE] and call $S(j, p, Q \cup T_{ij}, U - T_{pj})$.
Otherwise, repeatedly select a subtask $T_{pj} \in \mathcal{T}_{j+1}$ [SUBTASK-ORDERING RULE] and call $S(j, p, \{\}, \mathcal{T}_{j+1} - T_{pj})$.

Figure A.1: Procedure S, Describing the Basic Search Algorithm.

exploring that alternative and thereby wasting search time. Clearly, there is a trade-off between the time spent in enforcing consistency and the actual savings achieved in search time. The simplest possible consistency-enforcing rule is represented by Rule **C1**, which states that the newly scheduled subtask T_{ij} is not allowed to exceed its effective deadline.

Rule **C1**: The following must hold:

$$t_{ij} + \tau_{ij} \leq d_{ij}.$$

Theorem 7. Rule **C1** is correct.

Proof. Trivial. □

Rule **C1** can be considered to be the ultimate consistency check, since it is equivalent to a feasibility test for the partial schedule. We will describe in a later section how the violation of this rule can trigger a special backtracking behavior of the search process.

The following rules use different ways to determine lower bounds on the tardiness of schedules for the tasks sets that consist of subtasks in $U, \mathcal{T}_{j+1}, \ldots, \mathcal{T}_m$, the set of all subtasks that remain to be scheduled. Whenever such a lower bound on the tardiness is larger than zero, no feasible completion of the partial schedule exists. Rule **C1** and Rule **C2** use different techniques to determine two lower bounds on the tardiness of schedules for the remaining subtasks.

The following Rule **C2** uses the fact that the EDF algorithm is optimal to generate feasible schedules for tasks with identical ready times on a single processor.

Rule **C2**:	Schedule the remaining subtasks in U according to the EEDF algorithm, assuming that all release times are equal to $r = max\{e_{ij}, \min_{T_{lj} \in U}\{r_{lj}\}\}$. No subtask is allowed to exceed its effective deadline.

Theorem 8. Rule **C2** is correct.

Proof. In every schedule that is based on the current partial schedule, the subtasks in U are bound to start at time r or later. Rule **C2** defines the release time of all the subtasks in U to be r. Consequently, it does not postpone the release times beyond the earliest point in time by which the subtasks in U can start. If a task set is schedulable with release times equal or larger than r, clearly it is also schedulable if all release times are exactly r. It follows that, when the EEDF algorithm fails to feasibly schedule the subtasks in U with release times r, no feasible schedule exists for the subtasks in U. Consequently, the partial schedule generated at this point in the search process can not possibly be completed in a feasible way. □

The following Rule **C3** follows a different approach: Instead of testing whether a feasible schedule exists, it determines the optimal maximum completion time c_{max} of all the possible schedules of the subtasks in U and compares it with the latest effective deadline d_{max} among the subtasks in U, i.e. $d_{max} = \max_{T_{lj} \in U}\{d_{lj}\}$. If $d_{max} < c_{max}$, then, for every schedule of subtasks in U, there is at least one subtask that can not meet its effective deadline. Therefore, the partial schedule generated at this point in the search process can not be feasibly completed.

Comparing the optimal maximum completion time c_{max} with the maximum deadline d_{max} is a looser rule to check for consistency than checking for feasibility directly. There exist task sets that have $d_{max} > c_{max}$, for which no feasible schedules exist. However, rules that directly check for feasibility (like Rule **C2**) rely on optimal algorithms to generate feasible schedules. Unfortunately, efficient such algorithms are mostly restricted to single-processor problems. On the other hand, good lower bounds for the optimal maximum completion time can be found for multiprocessor- and multistage-scheduling problems like the flow-shop scheduling problem. Rule **C3** takes advantage of the following well-known fact: To minimize the maximum completion time in a flow shop with identical release times and no more than three processors, we only need to consider permutation schedules. French [8] and Szwarc [51] describe ways to determine lower bounds on the optimal maximum completion time for 3-processor permutation flow shops. We are going to use two of these methods in the following way. If in the search process there are still more than two processors left to be scheduled, we can determine a lower bound c_{max} on P_{j+2} for all the tasks that have not been scheduled yet on P_j. If $c_{max} > \max_{T_{lj}\in U}\{d_{l(j+2)}\}$, we can not possibly expand the partial schedule into a feasible one. By considering bounds on completion time and effective deadlines in the subsequent processor P_{j+2}, the consistency check becomes substantially more powerful than if it limited itself to P_j. To develop the lower bounds, we consider three best-case scenarios: the processing on either one of the three processors P_j, P_{j+1}, or P_{j+2} is continuous. In other words, on one of the three processors, the processor is never idling. From these scenarios we derive three components lb^A_j, lb^A_{j+1}, and lb^A_{j+2} of the lower bound lb^A for c_{max}:

$$\begin{aligned} lb^A_j &= r + \sum_{T_{lj}\in U} \tau_{lj} + \min_{T_{kj}\in U}\{\tau_{k(j+1)} + \tau_{k(j+2)}\} && \text{(A.1)} \\ lb^A_{j+1} &= r + \min_{T_{lj}\in U}\{\tau_{lj}\} + \sum_{T_{kj}\in U} \tau_{k(j+1)} + \min_{T_{oj}\in U} \tau_{o(j+2)} && \text{(A.2)} \\ lb^A_{j+2} &= r + \min_{T_{lj}\in U}\{\tau_{lj} + \tau_{l(j+1)}\} + \sum_{T_{kj}\in U} \tau_{k(j+2)} && \text{(A.3)} \end{aligned}$$

where we defined r earlier as $r = max\{e_{ij}, \min_{T_{lj}\in U}\{r_{lj}\}\}$ and

$$lb^A = \max\{lb_1, lb_2, lb_3\}. \tag{A.4}$$

Similar considerations result in the following lower bound:

$$lb^B = r + \max_{T_{lj}\in U}\{\tau_{lj} + \tau_{l(j+1)} + \tau_{l(j+2)} + \sum_{T_{kj}\in U, T_{kj}\neq T_{ij}} \min\{\tau_{lj}, \tau_{k(j+2)}\}\} \tag{A.5}$$

We formulate Rule **C3** in the following way:

Rule **C3**: If $j \leq m - 3$, determine lb^A, lb^B as described in Equation (A.4) and Equation (A.5). Determine $lb = max\{lb^A, lb^B\}$ and $d_{max} = \max_{T_{lj} \in U}\{d_{l(j+2)}\}$. The following must hold:

$$lb \leq d_{max}.$$

Theorem 9. Rule **C3** is correct.

Proof. All subtasks in U are assumed to be ready to execute at the same time r, that is, as soon as the partial schedule allows the first subtask to start its execution. If a feasible schedule existed for the task set, certainly a feasible schedule would exist for the same schedule with all the release times for the unscheduled subtasks set to r. If Rule **C3** is not satisfied, we can not complete the partial schedule without at least one subtask on P_3 missing its effective deadline. □

Despite the efforts to guide the search process, it can not be prevented from running into deadends and having to backtrack. When the search process repeatedly runs into the same deadend, it is said to be *thrashing*. Often thrashing could be prevented if the search process carried along information from past failures. We describe an effective technique to reduce thrashing by storing information about past failures. This technique is especially useful to avoid thrashing during the process of generating a feasible schedule on a single processor. We describe it for the special case where the search process has generated a feasible partial schedule for all processors $P_1, \ldots, P_{m-1}$ and tries to complete it on the last processor P_m. If at any given point it fails to feasibly schedule a set U of remaining subtasks on P_m, starting from time r, we can safely assume that the same set U_m can not be feasibly scheduled when starting at time r or later. We note that the same holds for any superset of U. If at any point during the search process a set W (with $W \supseteq U$) remains to be scheduled at time r or later, we can safely backtrack, without having to continue the search process into the same deadend again.

If the consistency-enforcing rules allow us to determine a lower bound on the tardiness for the subtasks in U if scheduled starting at time r, we can be even stricter: Let us assume that we have determined that the set U of subtasks can not be scheduled starting at time r with a total tardiness less than T. We can safely assume that the same set U, or any superset of it, can not be feasibly scheduled starting at time $r - T/|U|$ or later.

The same technique can be used for the generation of partial schedules on all the processors $P_1, \ldots, P_m$. Special care must be taken, however, when the search process moves from processor to processor. If , for example, the search process is generating a feasible schedule on P_j, a failure to feasibly complete the schedule does not necessarily imply that the subtasks in U can not be feasibly scheduled starting at time r. The search process may have failed to schedule any tasks in the subsequent sets $\mathcal{T}_{j+1}, \ldots, \mathcal{T}_m$. If we store the value

$$r_{crit}(U) = r - T/|U| \tag{A.6}$$

whenever we cannot schedule the set of subtasks U starting at time r with a tardiness smaller than T, we can use the following rule, the correctness of which was shown above.

Rule **C4**:	The following must hold: $t_{i(k)j} \leq r_{crit}(V)$ for every $V \subseteq U$, where r_{crit} has been defined in Equation (A.6).

A.1.2 Look-Ahead Rules

Two ways to reduce the average cost of backtrack search are to either carefully select the order in which the subtasks are selected (through subtask-ordering techniques) or the start time that is assigned to subtasks (through start-time-ordering techniques). Intuitively, a good subtask-ordering rule chooses 'difficult' subtasks first. In this way, the rule avoids spending much time in building partial solutions that cannot be completed later. Moreover, it is easier for the consistency-enforcing rules to detect upcoming deadends. Good start-time-ordering rules on the other hand select the least-constraining start times for the newly selected subtasks.

The basic Algorithm S defines the start time t_{ij} to be assigned to the newly scheduled subtask, and therefore leaves no room for a start-time-ordering rule. In this section, we describe two subtask-ordering rules that proved to support the search for feasible schedules.

Rule **LA1** takes advantage of the fact that the probability that no feasible schedule exists on a processor increases with increasing the ratio of processing time to laxity on that processor. By definition of the effective release time and deadlines, the amount of slack is identical on all the processors. A good subtask-ordering rule therefore tries to schedule first subtasks on processors with long subtasks. Unfortunately, Algorithm S always starts by scheduling subtasks on processor P_1, then on P_2, and so on. In a flow shop with long subtasks on processors that are towards the end of the flow shop, the search algorithm would start scheduling difficult subtasks very deep in the search tree, a case that would cause thrashing. We describe a way to transform such flow shops in a flow shop that has processors with long task sets at the beginning. The search algorithm would then search for a feasible schedule for the task set $\mathcal{T}$ by trying to schedule the transformed task set $\sigma(\mathcal{T})$. The transformation σ is defined as follows:

$$\begin{aligned} \sigma(\tau_{ij}) &= \tau_{i(m+1-j)} \\ \sigma(d_{ij}) &= -r_{i(m+1-j)} \\ \sigma(r_{ij}) &= -d_{i(m+1-j)} \end{aligned}$$

We can think of $\sigma(\mathcal{T})$ as being the symmetrical image of $\mathcal{T}$, where release times and deadlines have been switched. Whenever the set $\mathcal{T}$ is feasibly schedulable, the same is true for the task set $\sigma(T)$. Moreover, if $\mathcal{T}$ has long subtasks on late processors, $\sigma(T)$ has long subtasks on early processors. This motivates the following subtask-ordering Rule **LA1**. (Again, there is no need to check look-ahead rules for their correctness.)

Rule **LA1**:	Before the search process starts, check if the task set $\mathcal{T}$ contains longer subtasks in the processors $P_{\lceil m/2\rceil+1}, \ldots, P_m$. If this is the case, apply the search algorithm to the transformed task set $\sigma(\mathcal{T})$.

The following subtask-ordering rule defines the order in which to select the next subtask T_{pj} among the subtasks in U in Step 4 of Algorithm S. The goal is to generate a partial schedule on P_j that can easily be completed into a feasible schedule (for the processor P_j). Feasible schedules for single-processor problems tend to be very similar in general to EDF schedules. Rule **LA2** therefore tries to look for an EDF schedule first by defining an EDF subtask-ordering in the following way:

Rule **LA2**: During Step 4 of Algorithm S, select T_{pj} to be that subtask in U with the earliest effective deadline among all subtasks that have not been selected yet.

A.1.3 Look-Back Rules

Whenever a deadend is encountered and backtracking is invoked, a decision must be made as which assignment to reverse. Algorithm S uses a *chronological* backtracking scheme, in which the assignment is reversed that was made last, and which led to the deadend. Sometimes thrashing can be avoided if a look-back rule can detect that not only the last assignment must be reversed, but earlier assignments also. The following backtrack Rule **B1** determines when to reverse both the last assignment to T_{ij} and the assignment before. (Let T_{xj} be the subtask that was selected and assigned a start time immediately before T_{ij}.)

Rule **B1**: If Rule **C1** fails for T_{ij}, that is, if the newly scheduled subtask T_{ij} does not meet its effective deadline d_{ij}, then reverse the assignments to T_{ij} and T_{xj}.

Theorem 10. Rule **B1** is correct.

Proof. To show that Rule **B1** is correct, we must first show that, whenever Rule **B1** fires, there is no idle time between T_{xj} and T_{ij}. If there was idle time, T_{ij} would have to start as soon as $T_{i(j-1)}$ is terminated, that is, $t_{ij} = e_{i(j-1)}$. This cannot occur, since $d_{i(j-1)} = d_{ij} - \tau_{ij}$, and therefore $d_{i(j-1)} < e_{i(j-1)}$. Rule **C1** would have failed at the time when $T_{i(j-1)}$ was scheduled. According to Algorithm S, T_{ij} must therefore start as soon as T_{xj} is terminated, i.e. $t_{ij} = e_{xj}$. If T_{ij} fails to meet its deadline when scheduled at time e_{xj}, we can safely reverse the assignment to T_{xj} also. The partial schedule including T_{xj} cannot be successfully completed, because T_{ij} would have to be scheduled at time e_{xj} or later, and would therefore miss its effective deadline. □

Procedure $S^T(j, i, tard_{acc}, Q, U)$:

Input: Set Q already scheduled subtasks on processor P_j. Index of one selected subtask T_{ij} that has not been scheduled yet. Set U of remaining subtasks on processor P_j. The following holds: $Q + T_{ij} + U = \mathcal{T}_j$, where $\mathcal{T}_j$ is the set of all subtasks on P_j.
The schedule generated up to this point has the total tardiness $tard_{acc}$.

Output: The schedule with the minimum total tardiness.

Step 1: Schedule the subtask T_{ij} to start at time $t_{ij} = max\{r_{ij}, max_{T_{lj} \in Q}\{e_{lj}\}\}$. Set $e_{ij} = t_{ij} + \tau_{ij}$ and $tard_{new} = max\{e_{ij} - d_{ij}, 0\}$. If $j < m$, set $r_{i(j+1)} = e_{ij}$.

Step 2: If $tard_{acc} + tard_{new} \leq tard_{bound}$ [CONSISTENT], go to Step 3. Otherwise, a deadend is detected; return [BACKTRACK].

Step 3: If $U - T_{ij} = \{\}$ and $j = m$, a feasible schedule has been generated with a lower tardiness than the current bound. Define $tard_{bound} = tard_{acc} + tard_{new}$. Backtrack.

Step 4: If $U - T_{ij} = \{\}$, repeatedly select a subtask $T_{pj} \in U$ [SUBTASK-ORDERING RULE] and call $S(j, p, tard_{acc} + tard_{new}, Q \cup T_{ij}, U - T_{pj})$.
Otherwise, repeatedly select a subtask $T_{pj} \in \mathcal{T}_{j+1}$ [SUBTASK-ORDERING RULE] and call $S(j, p, tard_{acc} + tard_{new}, \{\}, \mathcal{T}_{j+1} - T_{pj})$.

Figure A.2: Procedure S^T to Minimize Total Tardiness.

A.2 Minimizing Total Tardiness

Algorithm S can be easily transformed into the branch-and-bound Algorithm S^T to minimize the sum of tardiness. *Procedure* S^T, described in Figure A.2, differs only slightly from Procedure S. First, it keeps track of the lowest total tardiness found so far (Step 3) and compares against it to determine whether to backtrack or not. Second, it does not terminate when it succeeds in generating a complete schedule with a lower tardiness, but forces a backtrack (Step 3) to continue the search. Algorithm S^T compares the total tardiness $tard_{acc}$ accumulated in the partial schedule with the optimal total tardiness found during the search process. If the accumulated tardiness exceeds the current bound $tard_{bound}$, the search algorithm can safely backtrack, since the completion of the partial schedule can only increase the total tardiness. After the search process terminates, the optimal value for the tardiness is $tard_{bound}$.

The performance of every branch-and-bound algorithm depends greatly on the initial estimate for the bound. The lower the bound, the more search paths can be avoided, and the better the performance of the algorithm is. For flow-shop scheduling, we use two approaches to determine an initial value for the upper bound $tard_{bound}$ for the total tardiness:

- Use fast heuristic algorithms. We determine an initial bound on the tardiness by using Algorithm $\mathcal{H}$. This bound can usually be improved by having Algorithm $\mathcal{H}$ run multiple times, each time defining a new processor to be the bottleneck.
- Determine the minimum total tardiness for permutation schedules. This is can again be done by applying branch-and-bound algorithms. Optimal permutation schedules can be found reasonably quickly, and have a total tardiness that can be used as a good bound for non-permutation schedules. Since Algorithm $\mathcal{H}$ generates only permutation schedules, it can be used to determine a good initial bound for the search for permutation schedules.

Once an initial bound on the tardiness has been determined, the search can start. The search process itself can be supported by a set of rules that are very similar to the rules used to find feasible schedules. The rules defined in Section A.1 can by large be used in Algorithm S^T. However, the consistency-enforcing rules described in Section A.1.1 are not correct for the problem of minimizing total tardiness. In the following, we will describe four consistency-enforcing rules to minimize total tardiness.

A.2.1 Consistency-Enforcing Rules to Minimize Total Tardiness

Consistency-enforcing rules were used in Section A.1.1 to determine whether the current partial schedule could be completed without any task missing its deadline. In this section, consistency-enforcing rules are applied to ensure that the current partial schedule can be expanded to a complete schedule whose total tardiness does not exceed the current bound $tard_{bound}$.

Rule $\mathbf{C1}^T$ is closely derived from Rule **C1** in Section A.1.1:

Rule $\mathbf{C1}^T$: The following must hold:

$$tard_{acc} + t_{ij} + \tau_{ij} - d_{ij} \leq tard_{bound}.$$

Theorem 11. Rule $\mathbf{C1}^T$ is correct.

Proof. Trivial. □

Rule **C2** described in Section A.1.1 is based on the optimality of the EDF algorithm to determine a feasible schedule for subtasks on one processor with identical release times. Unfortunately, the EDF algorithm does not generate schedules with minimal total tardiness, and can therefore not be used in general to determine a lower bound on the total tardiness for the subtasks in U. Nevertheless, EDF schedules for task sets with identical release times have two useful characteristics: First, as long as no task starts after its deadline in an EDF schedule, the total tardiness is minimized [6]. Second, in any EDF schedule, the maximum tardiness is minimized [25]. These two characteristics are used in the following Rule $\mathbf{C2}^T$:

Rule $\mathbf{C2}^T$: Schedule the subtasks in U according to the EEDF algorithm, assuming that all release times are equal to $r = max\{e_{ij}, \min_{T_{lj} \in U}\{r_{lj}\}\}$. Determine the total tardiness $tard_{tot}(U)$ and the maximum tardiness $tard_{max}(U)$.

Rule $\mathbf{C2.1}^T$:

If for all $T_{lj} \in U$, $t_{lj} \leq d_{lj}$, the following must hold:

$$tard_{acc} + tard_{tot}(U) \leq tard_{bound}.$$

Rule $\mathbf{C2.2}^T$:

If for some $T_{lj} \in U$, $t_{lj} > d_{lj}$, the following must hold:

$$tard_{acc} + tard_{max}(U) \leq tard_{bound}.$$

Theorem 12. Rule $\mathbf{C2}^T$ is correct.

Proof. As described in Section A.1.1, the schedulability of the subtasks in U is not affected if the release time of all subtasks is r. By the same argument, neither the minimum total tardiness nor the minimum maximum tardiness are increased when all subtasks in U are released at time

r. By the optimality of the EDF algorithm, $tard_{tot}(U)$ and $tard_{max}(U)$ are indeed lower bounds on the total or maximum tardiness of the task set U. □

Rule **C3** in Section A.1.1 uses the maximum completion time of subtasks to determine a bound on the maximum tardiness. We use this in Rule **C3**^{T}:

Rule **C3**^{T}: If $j \leq m - 3$, determine lb^A, lb^B as described above. Determine $lb = max\{lb^A, lb^B\}$ and $d_{max} = \max_{T_{lj} \in U}\{d_{l(j+2)}\}$. The following must hold:

$$tard_{acc} + lb - d_{max} \leq tard_{bound}.$$

Theorem 13. Rule **C3**^{T} is correct.

Proof. The correctness of Rule **C3**^{T} follows directly from the correctness of Rule **C3**. □

Rule **C4** in Section A.1.1 can easily be adopted to use information about passed failures to check for consistency:

Rule **C4**^{T}: The following must hold:

$$t_{ij} - r_{crit}(V) \leq (tard_{bound} - tard_{acc})/|U|$$

for every $V \subseteq U$ where r_{crit} has been defined.

Theorem 14. Rule **C4**^{T} is correct.

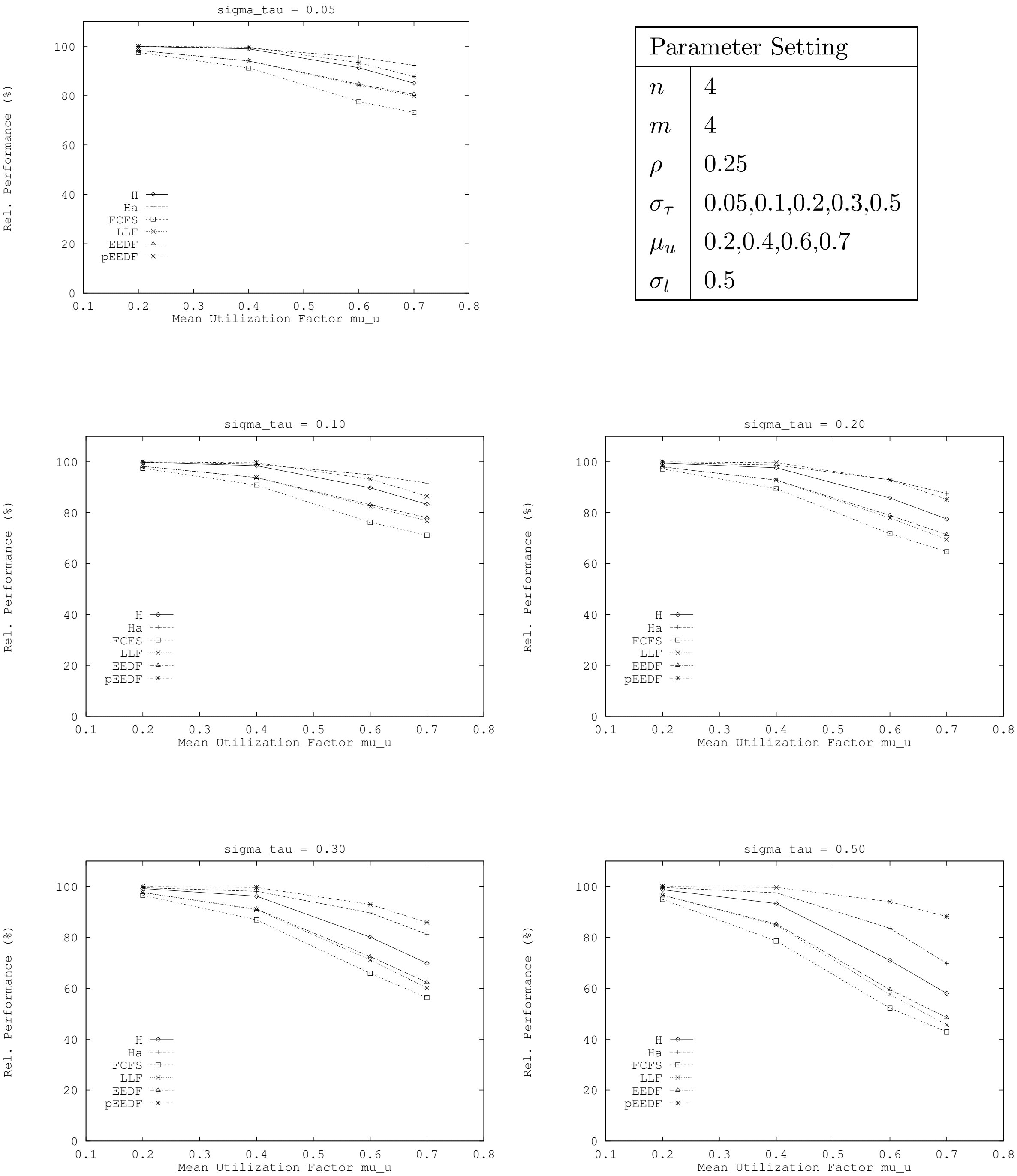
Proof. The correctness of Rule **C4**^{T} follows directly from the correctness of Rule **C4**. □

A.2.2 Conclusion

We used all of the rules described here at some time or the other in branch-and-bound algorithms to search for feasible flow shop schedules or flow shop schedules with minimum total tardiness. These algorithms were all based on either Procedure S or Procedure S^T. All rules contributed considerably to cut down the number of search states that had to be generated during the search process. With the exception of Rule **C4** and Rule **C4**^{T}, They also contributed to reduce the search time. Rule **C4** and Rule **C4**^{T} require the storing of large amounts of information about past failures. This causes the additional computation overhead to outweigh the gains in search space reduction, thus slowing down the search process.

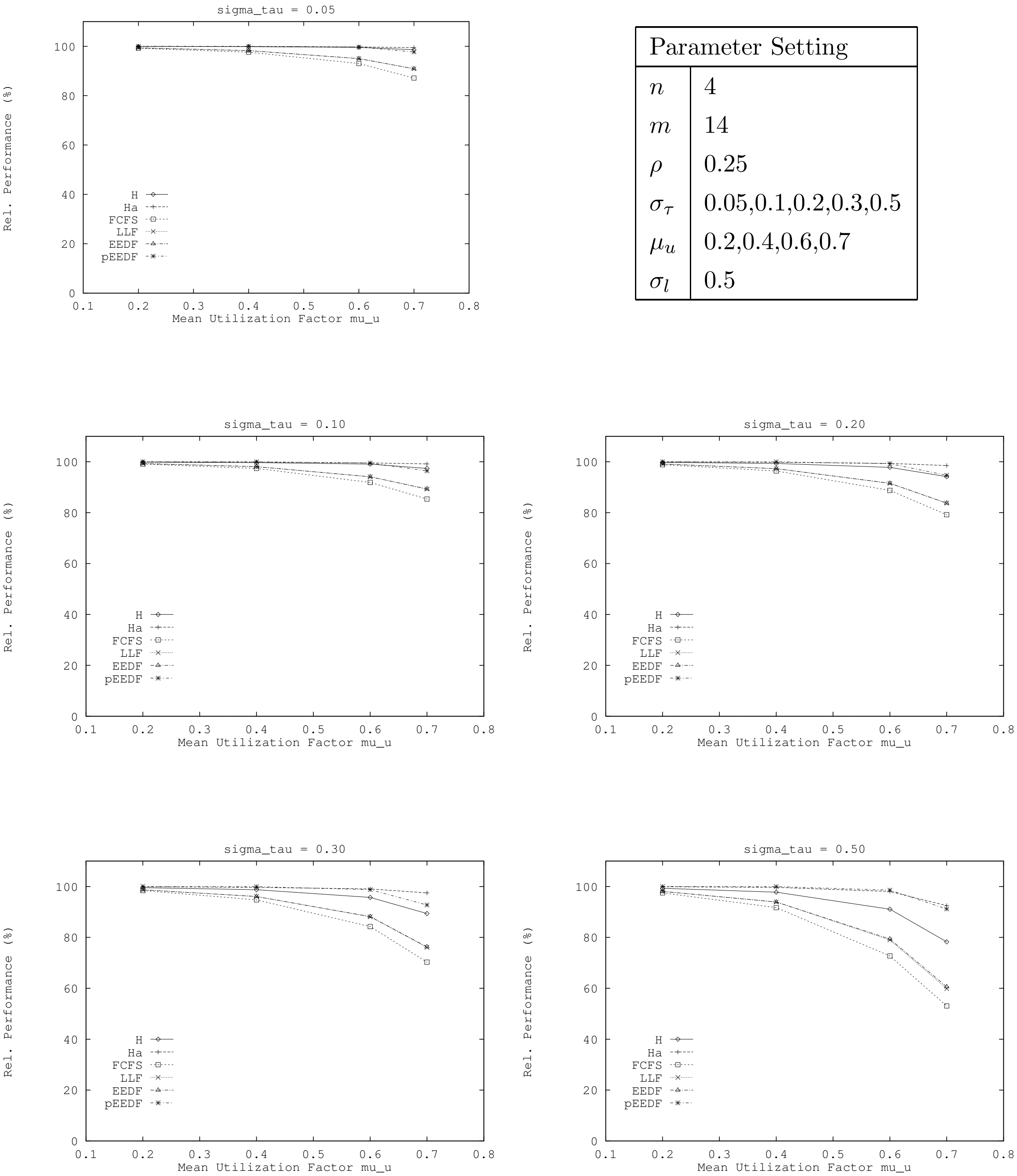
Appendix B

Simulation Results for Algorithm $\mathcal{H}$



Parameter Setting	
n	4
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.1: Relative Performance: 4 Tasks, 4 Processors.



Parameter Setting	
n	4
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.2: Relative Performance: 4 Tasks, 14 Processors.

Parameter Setting	
n	4
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

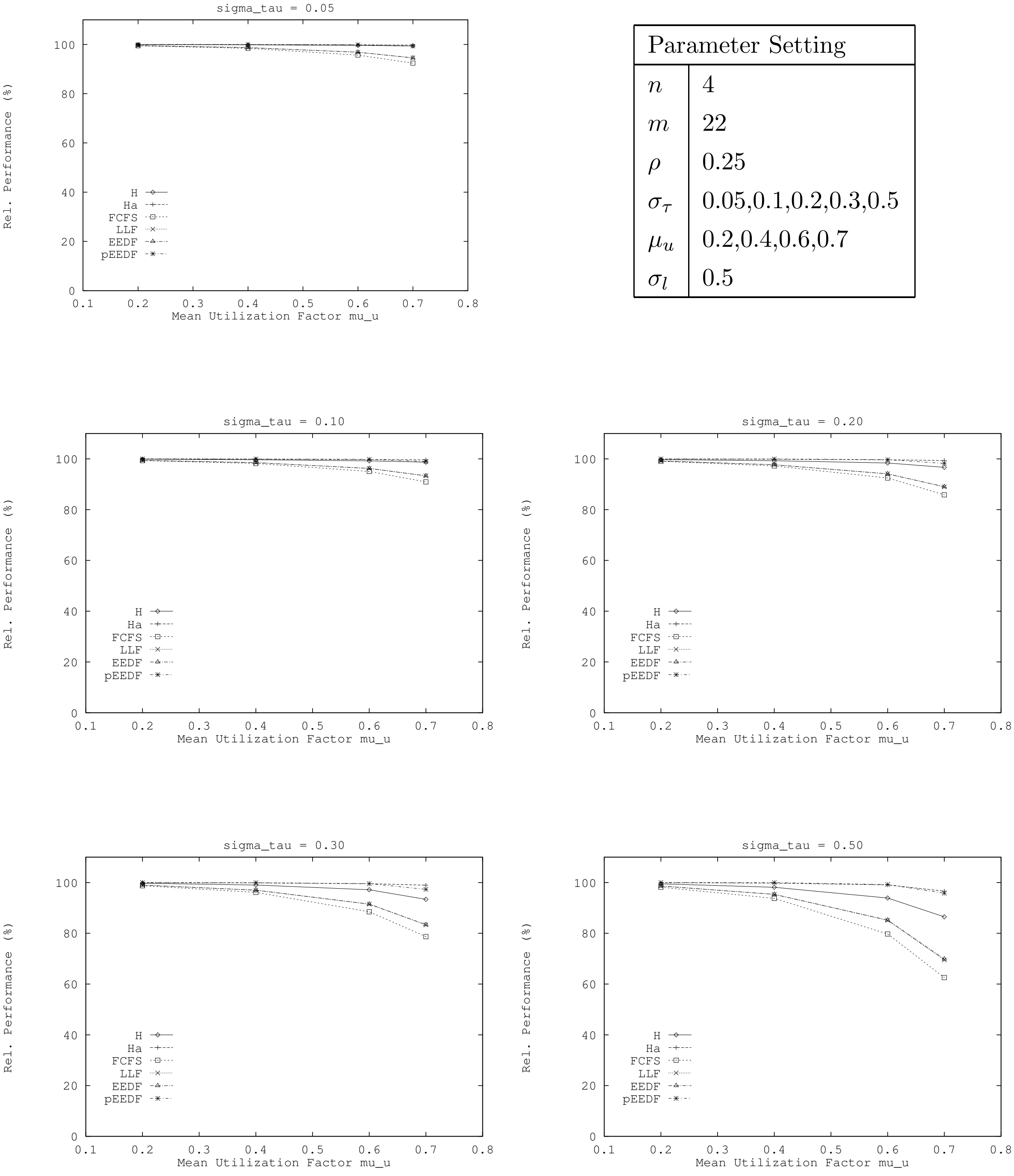
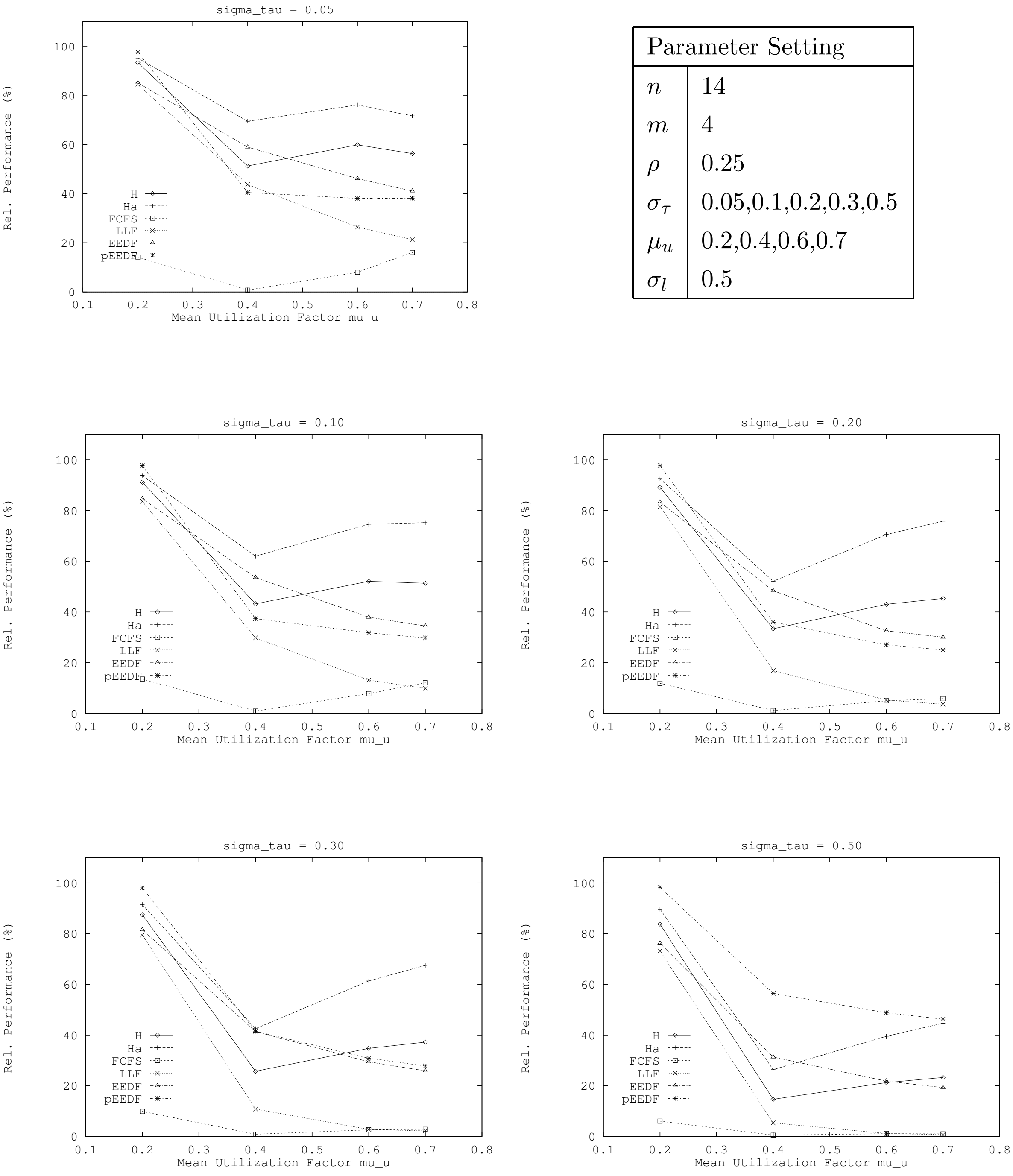
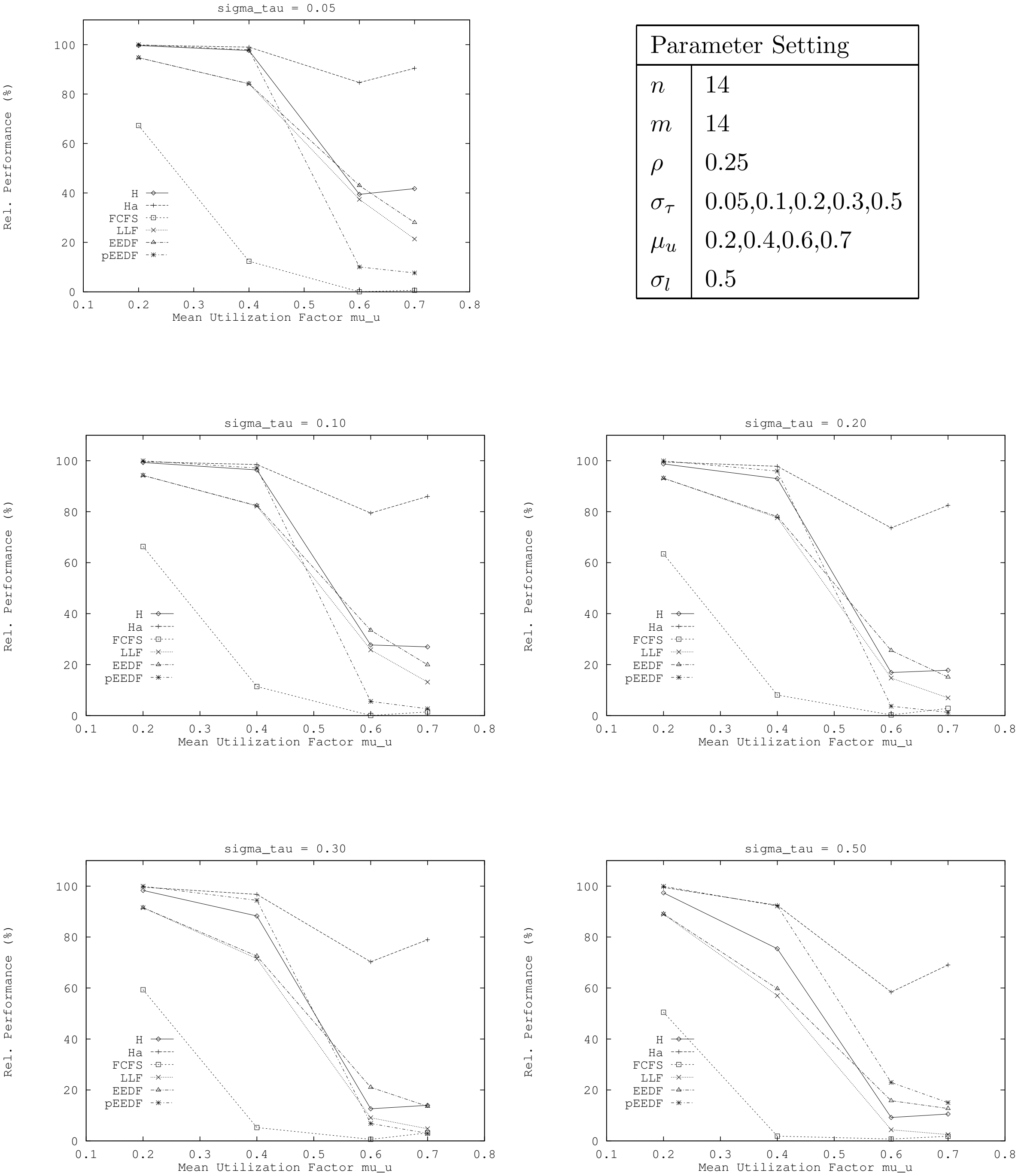


Figure B.3: Relative Performance: 4 Tasks, 22 Processors.



Parameter Setting	
n	14
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.4: Relative Performance: 14 Tasks, 4 Processors.



Parameter Setting	
n	14
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.5: Relative Performance: 14 Tasks, 14 Processors.

Parameter Setting	
n	14
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

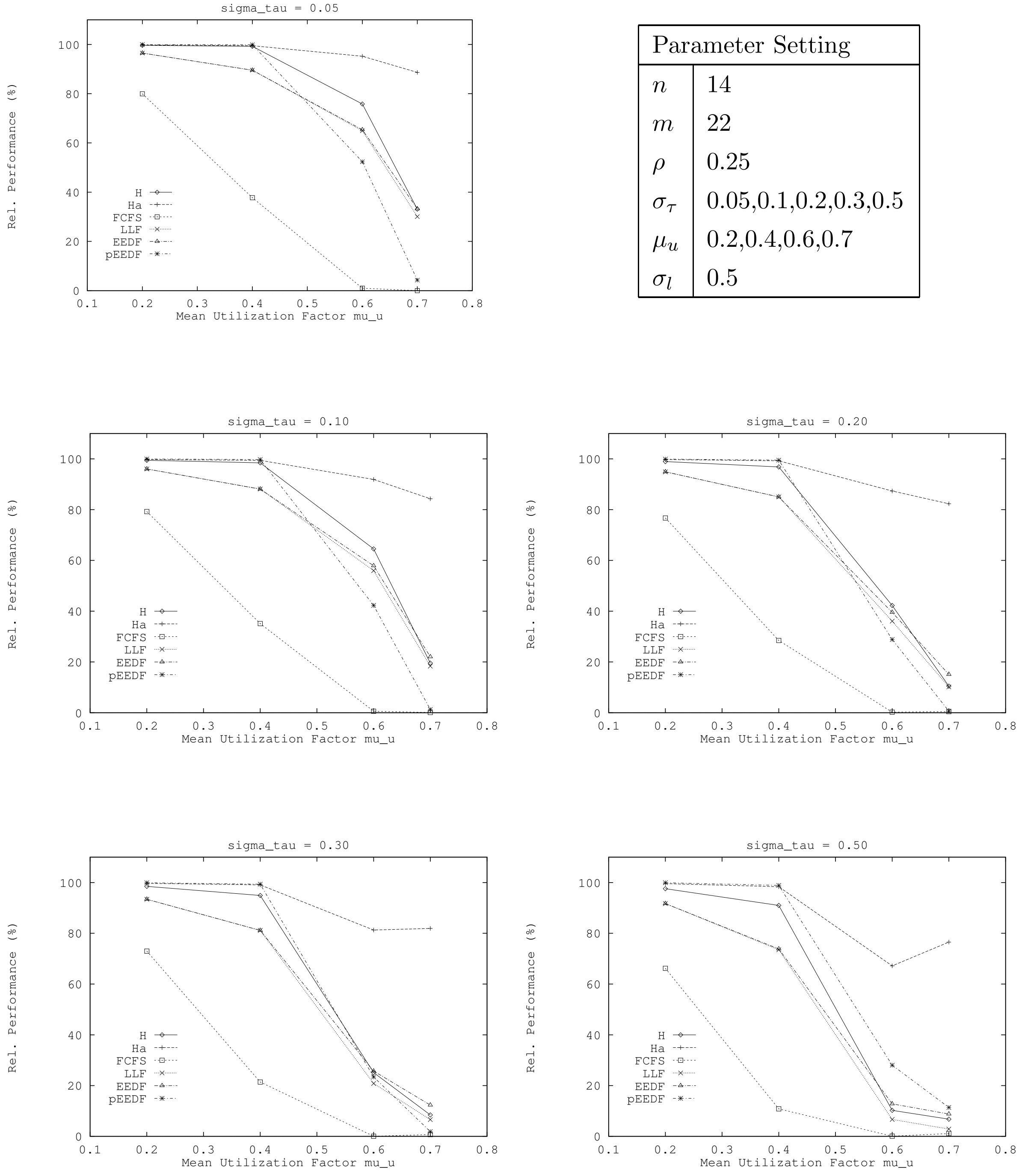
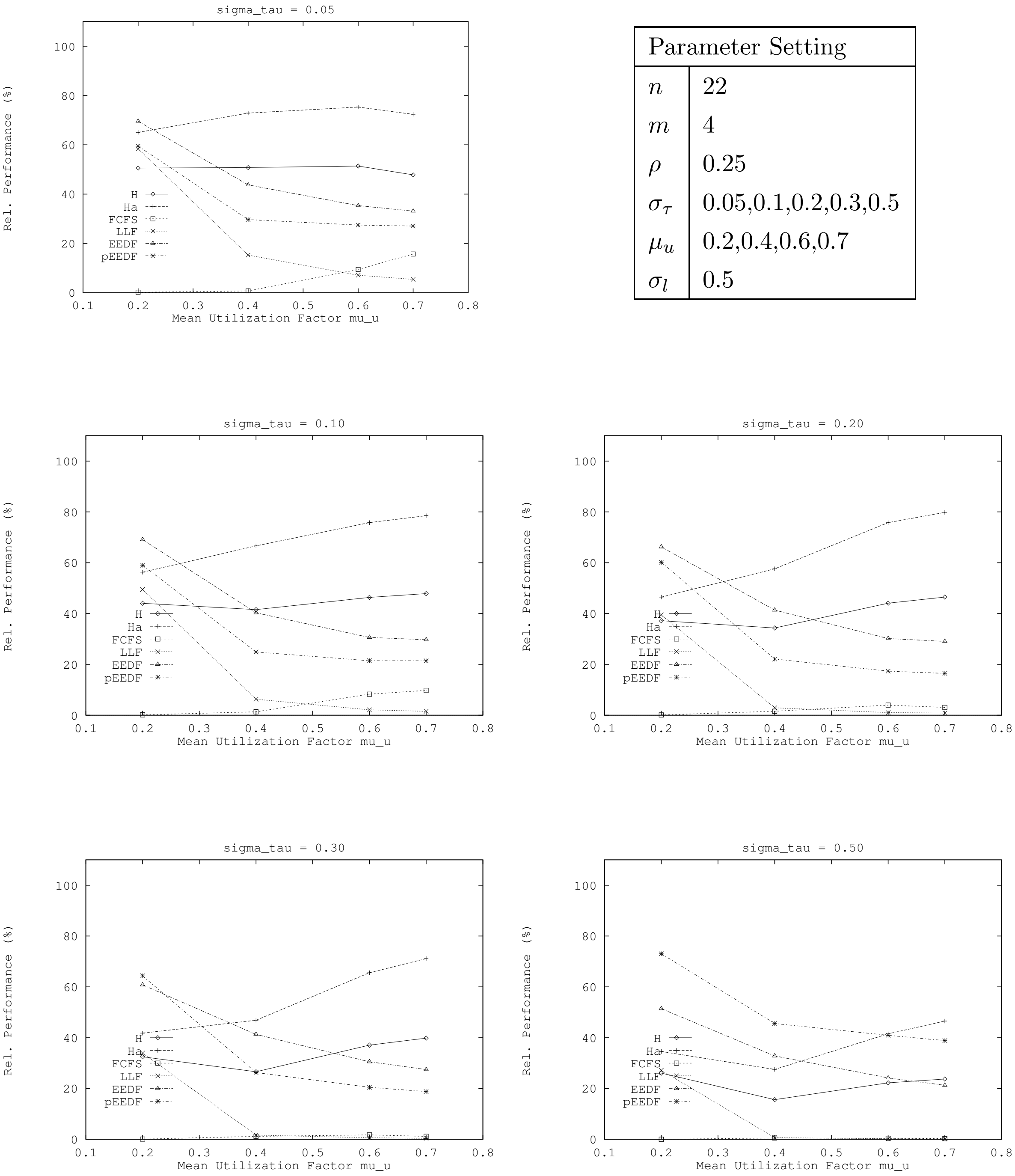


Figure B.6: Relative Performance: 14 Tasks, 22 Processors.



Parameter Setting	
n	22
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.7: Relative Performance: 22 Tasks, 4 Processors.

Parameter Setting	
n	22
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

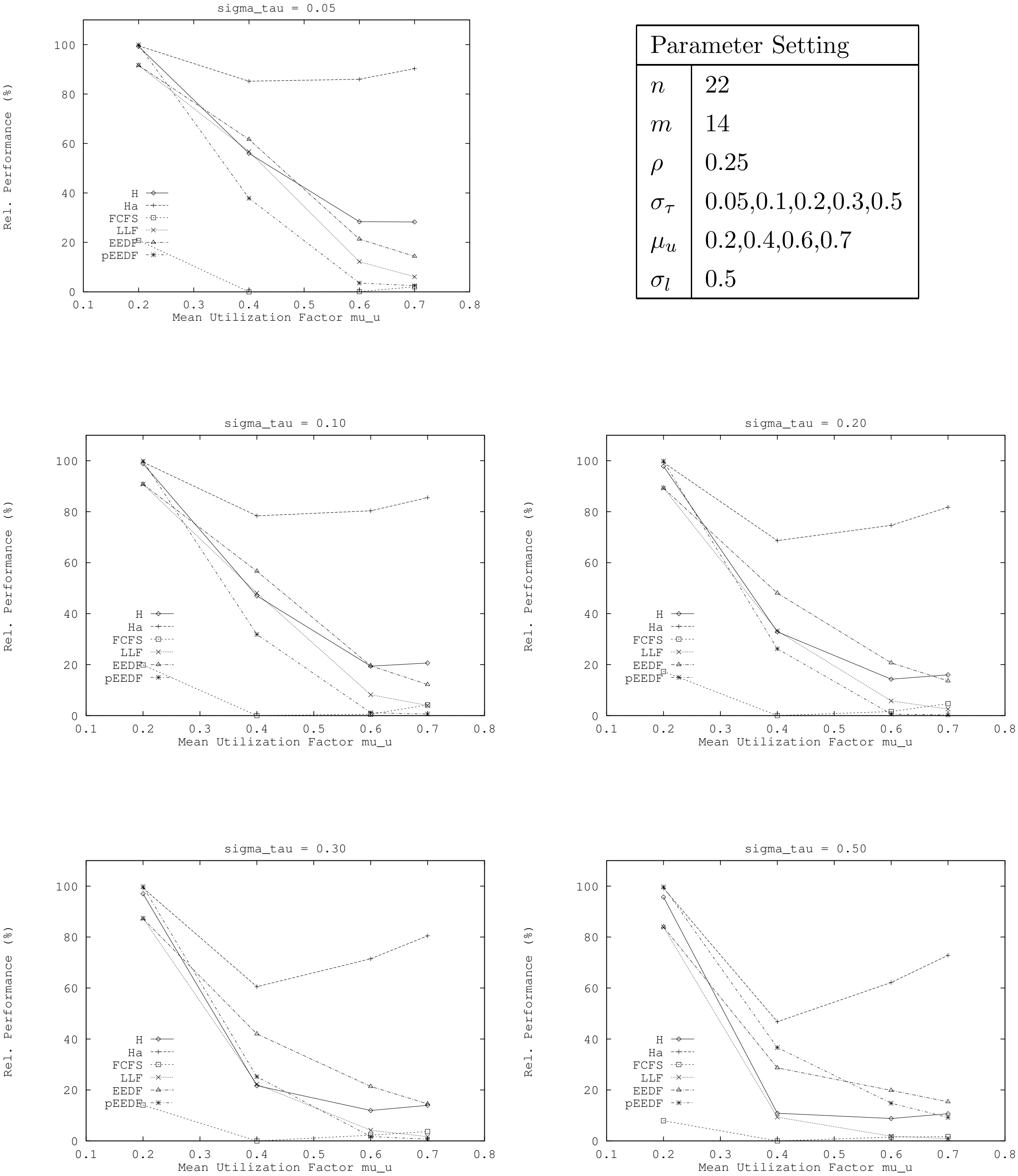
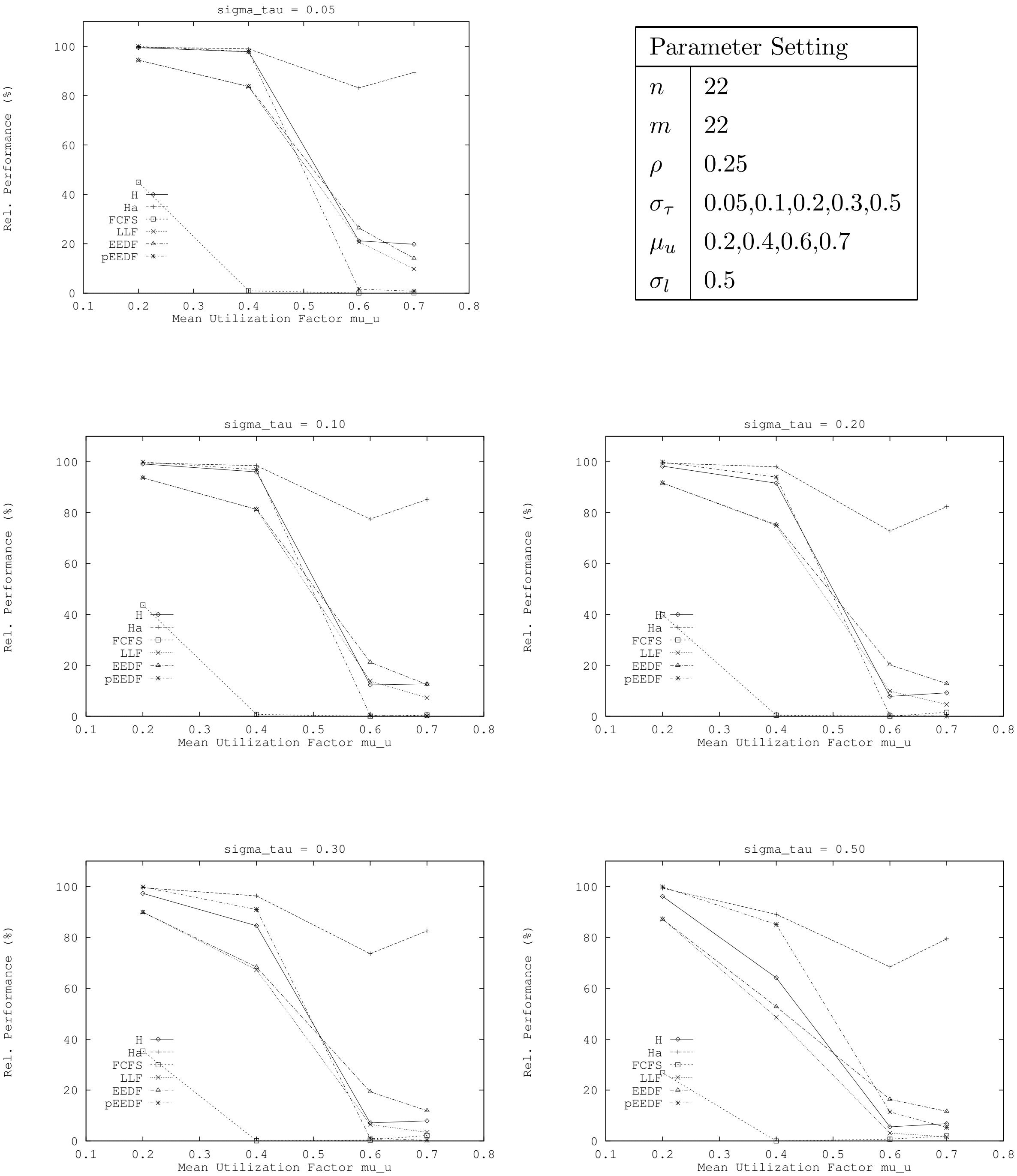
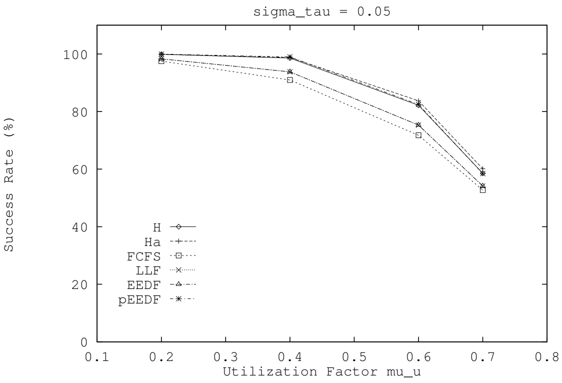


Figure B.8: Relative Performance: 22 Tasks, 14 Processors.



Parameter Setting	
n	22
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.9: Relative Performance: 22 Tasks, 22 Processors.



Parameter Setting	
n	4
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

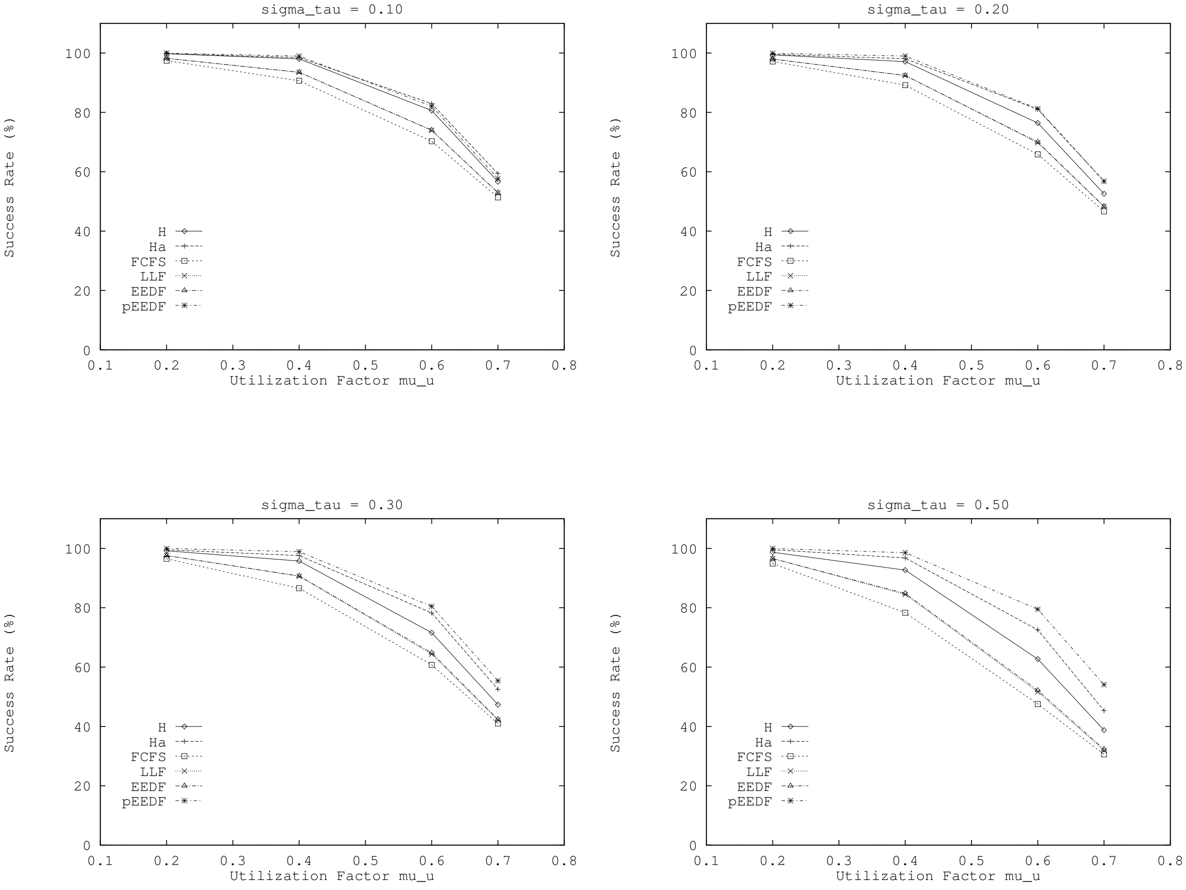


Figure B.10: Success Rate: 4 Tasks, 4 Processors.

Parameter Setting	
n	4
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

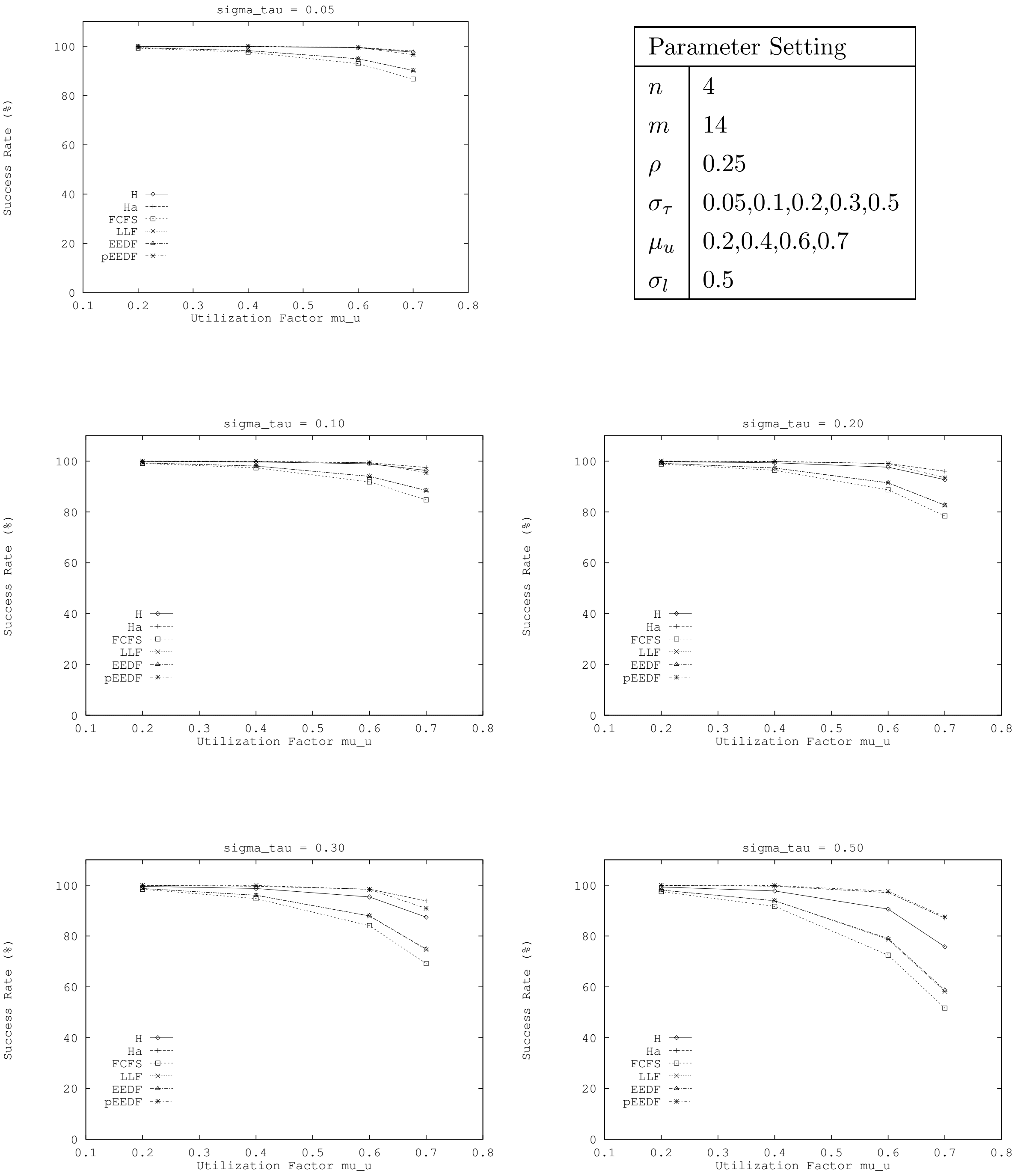
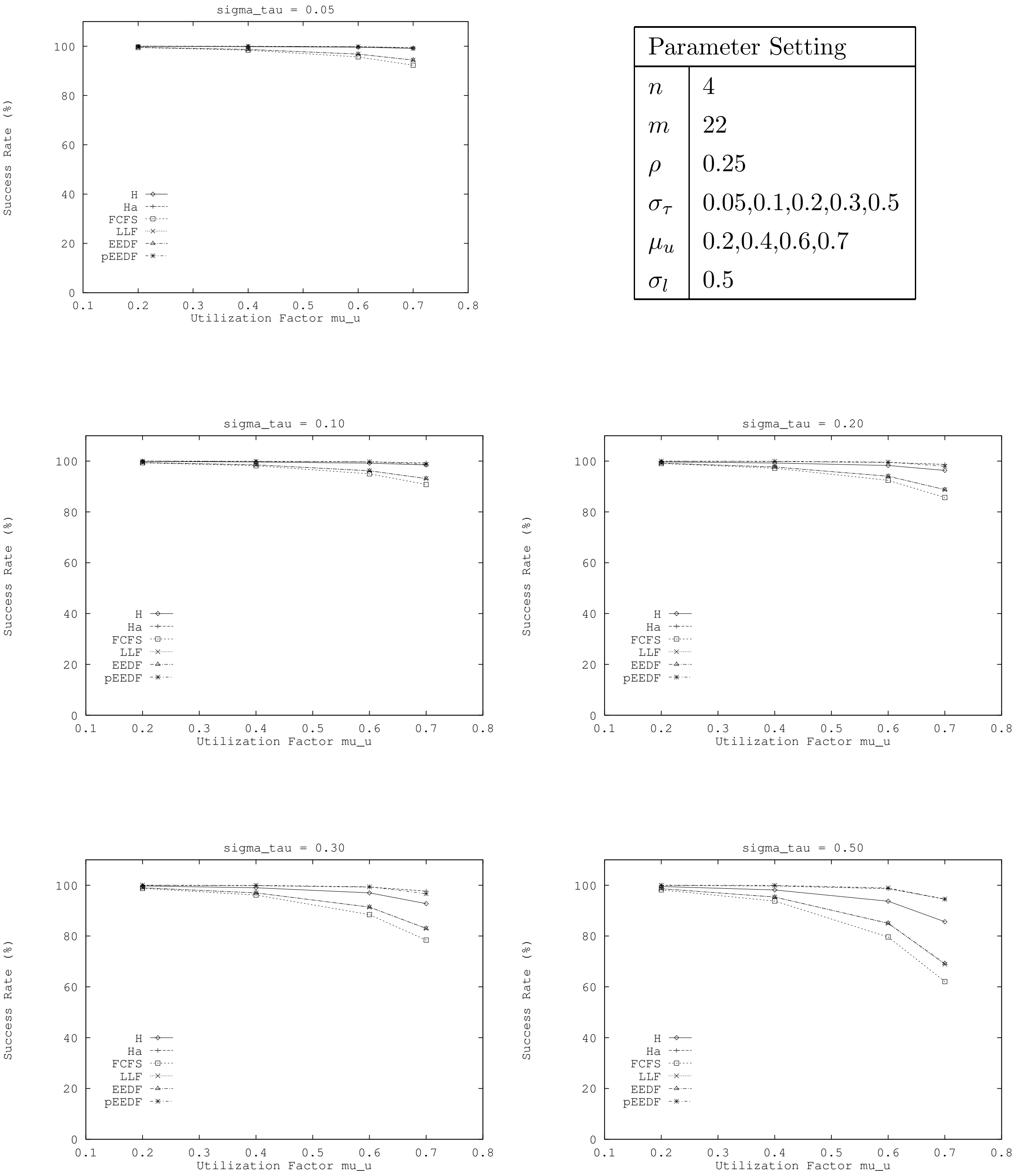
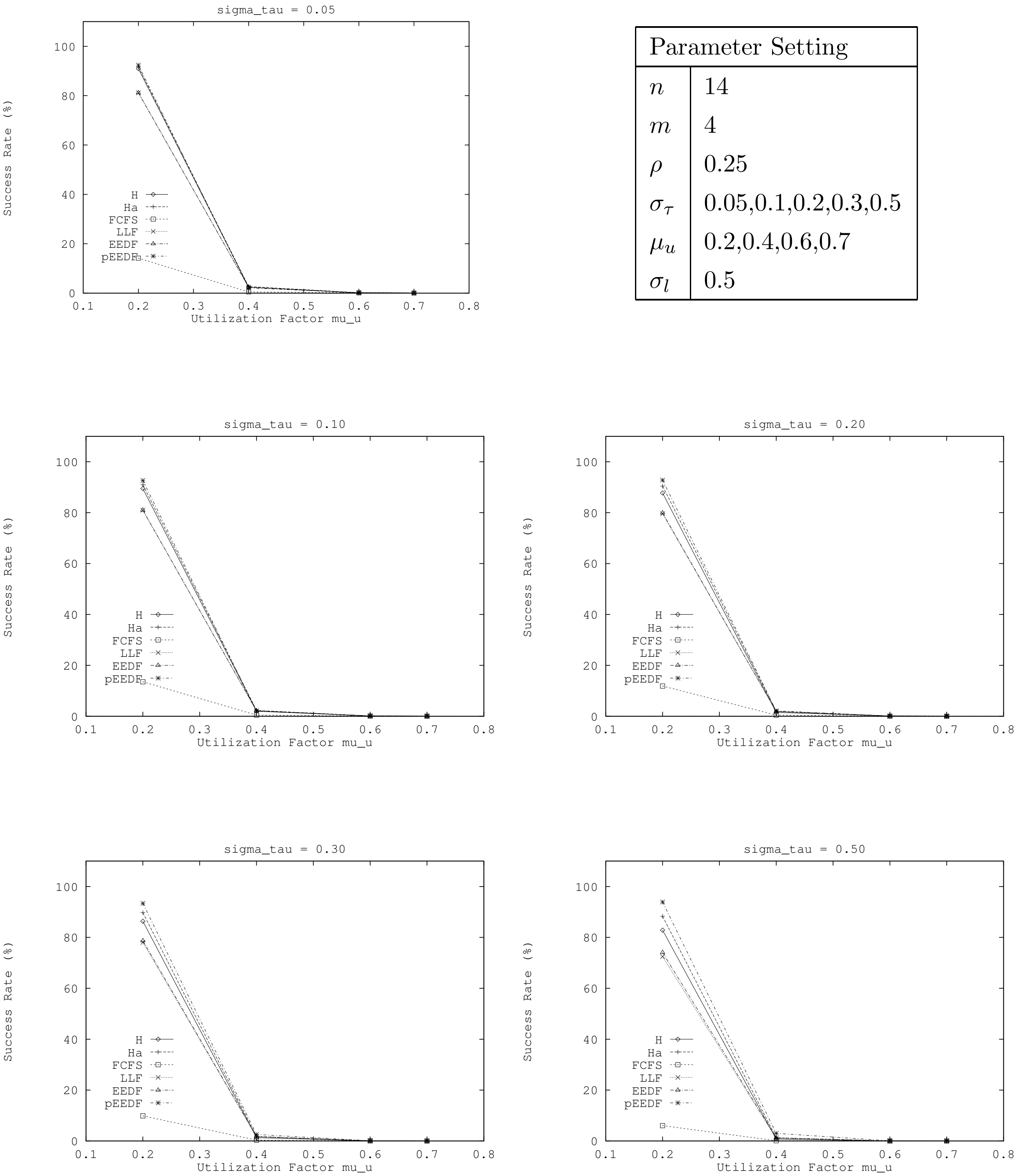


Figure B.11: Success Rate: 4 Tasks, 14 Processors.



Parameter Setting	
n	4
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.12: Success Rate: 4 Tasks, 22 Processors.



Parameter Setting	
n	14
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.13: Success Rate: 14 Tasks, 4 Processors.

Parameter Setting	
n	14
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

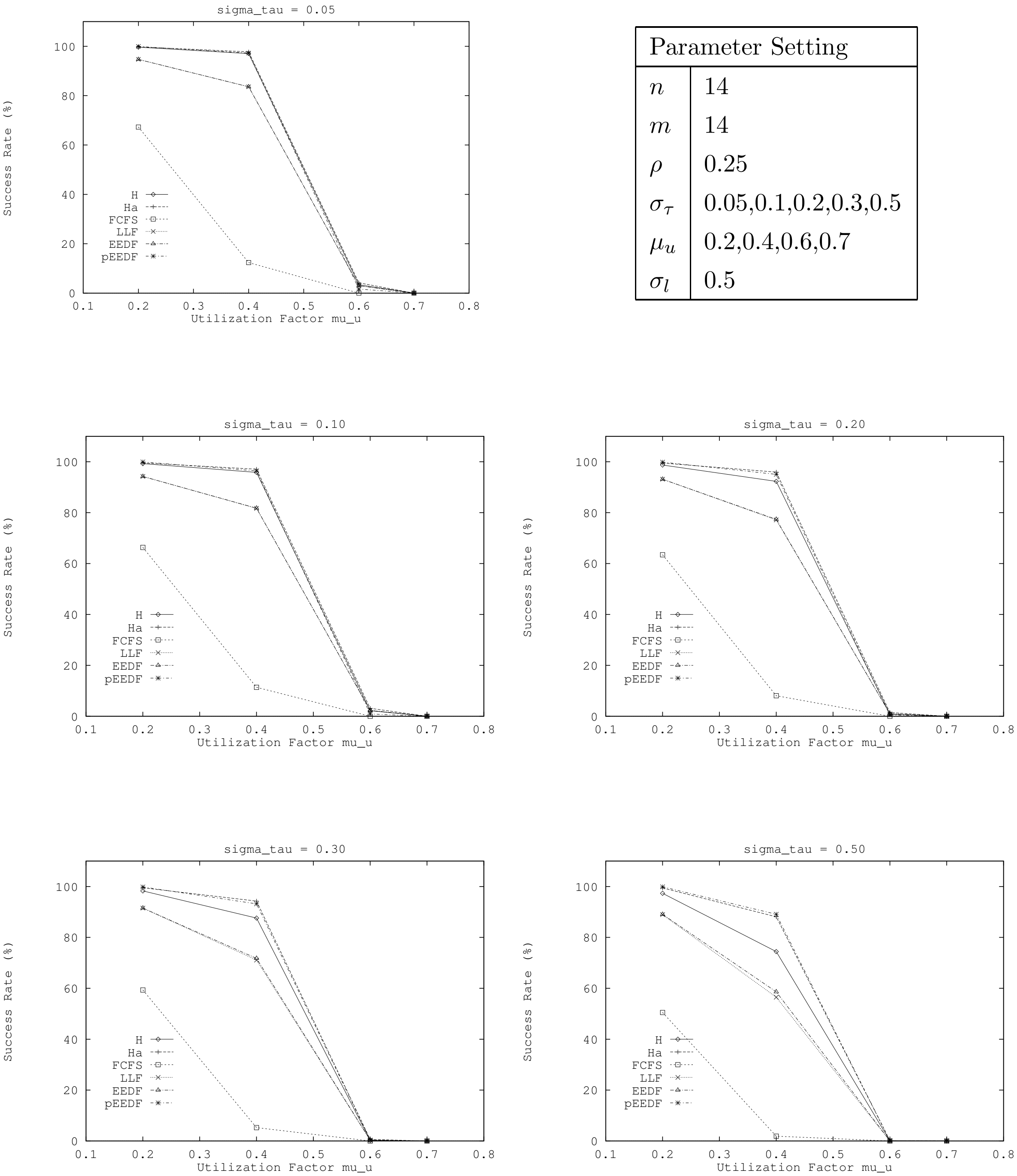
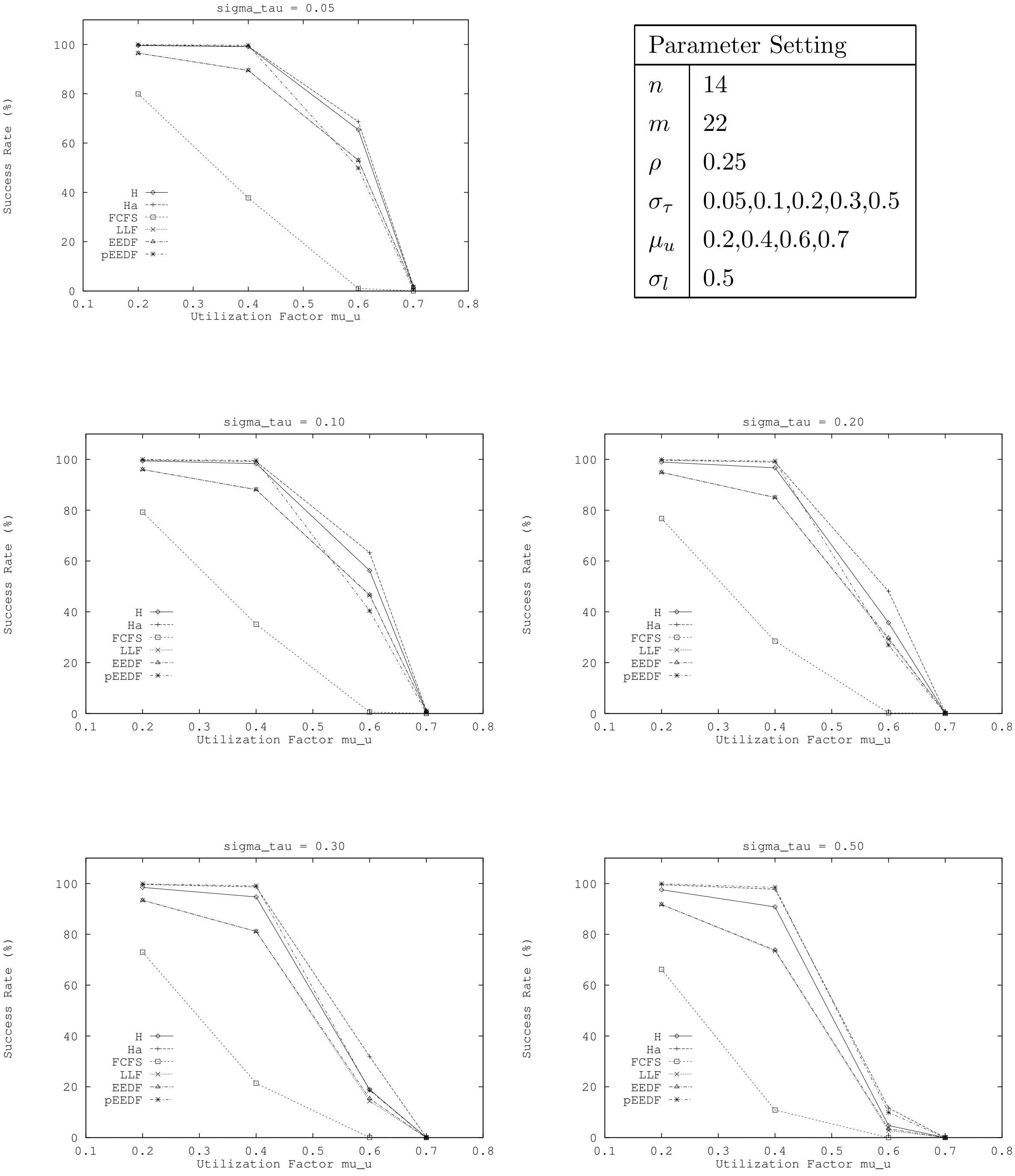
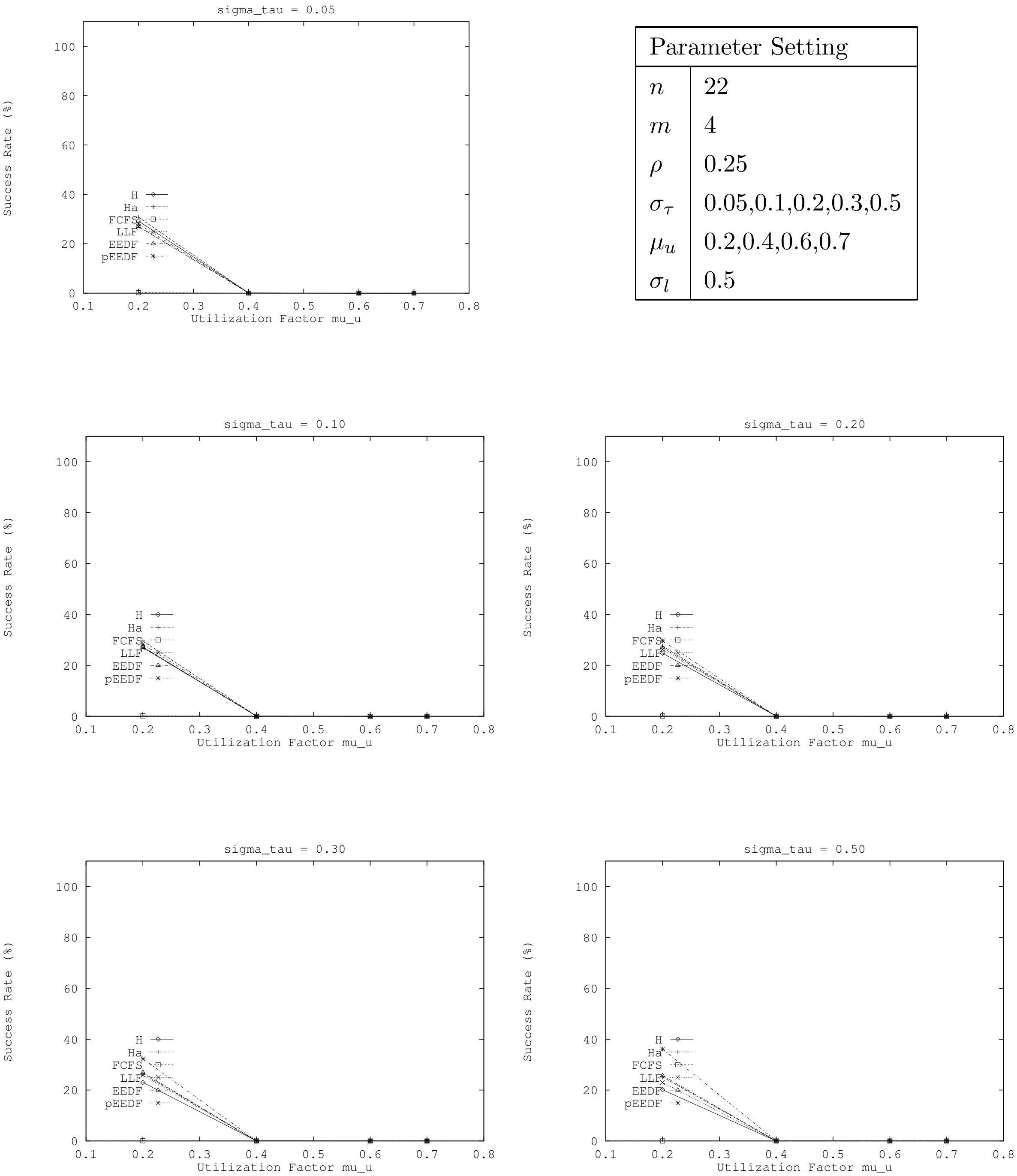


Figure B.14: Success Rate: 14 Tasks, 14 Processors.



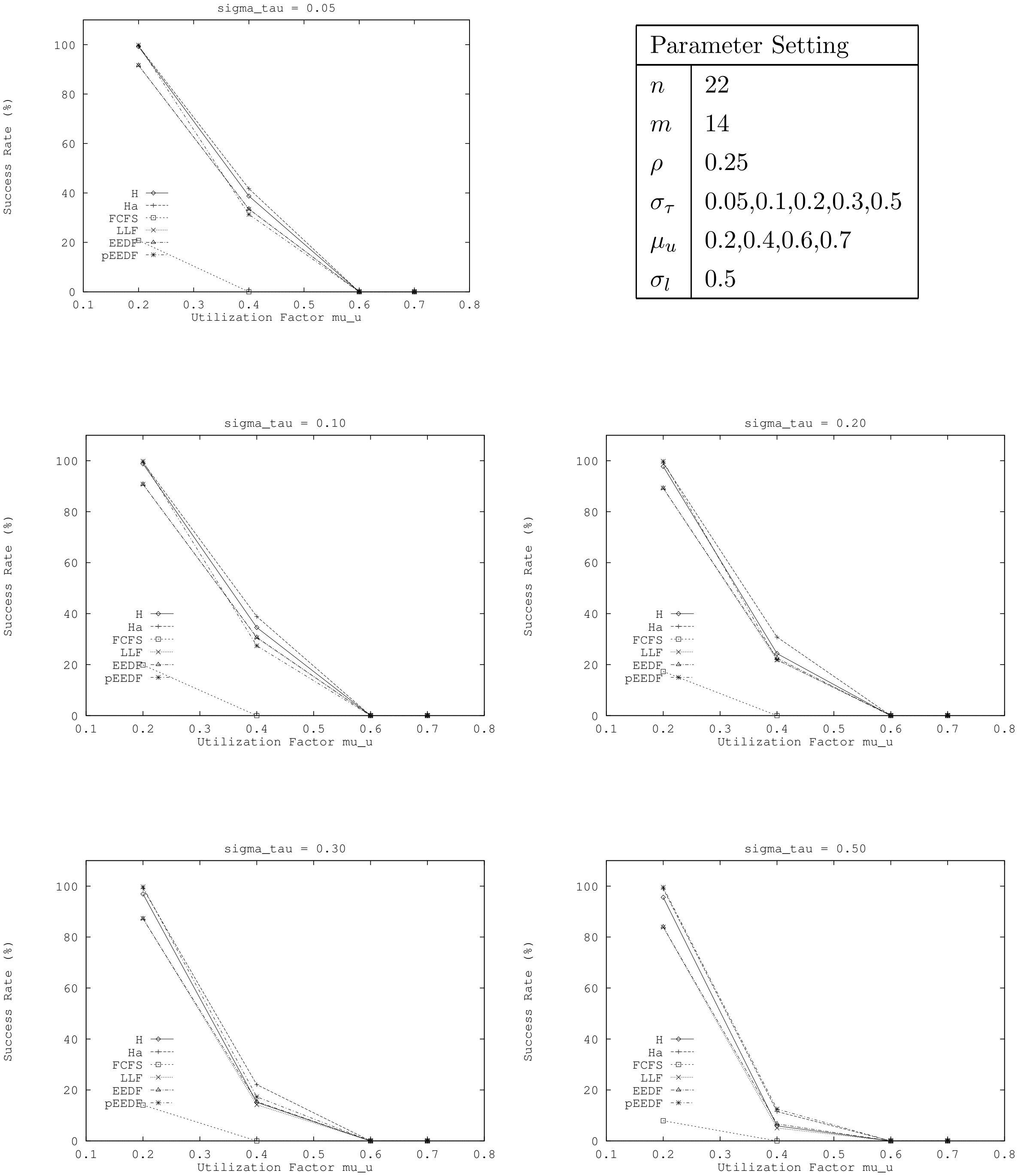
Parameter Setting	
n	14
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.15: Success Rate: 14 Tasks, 22 Processors.



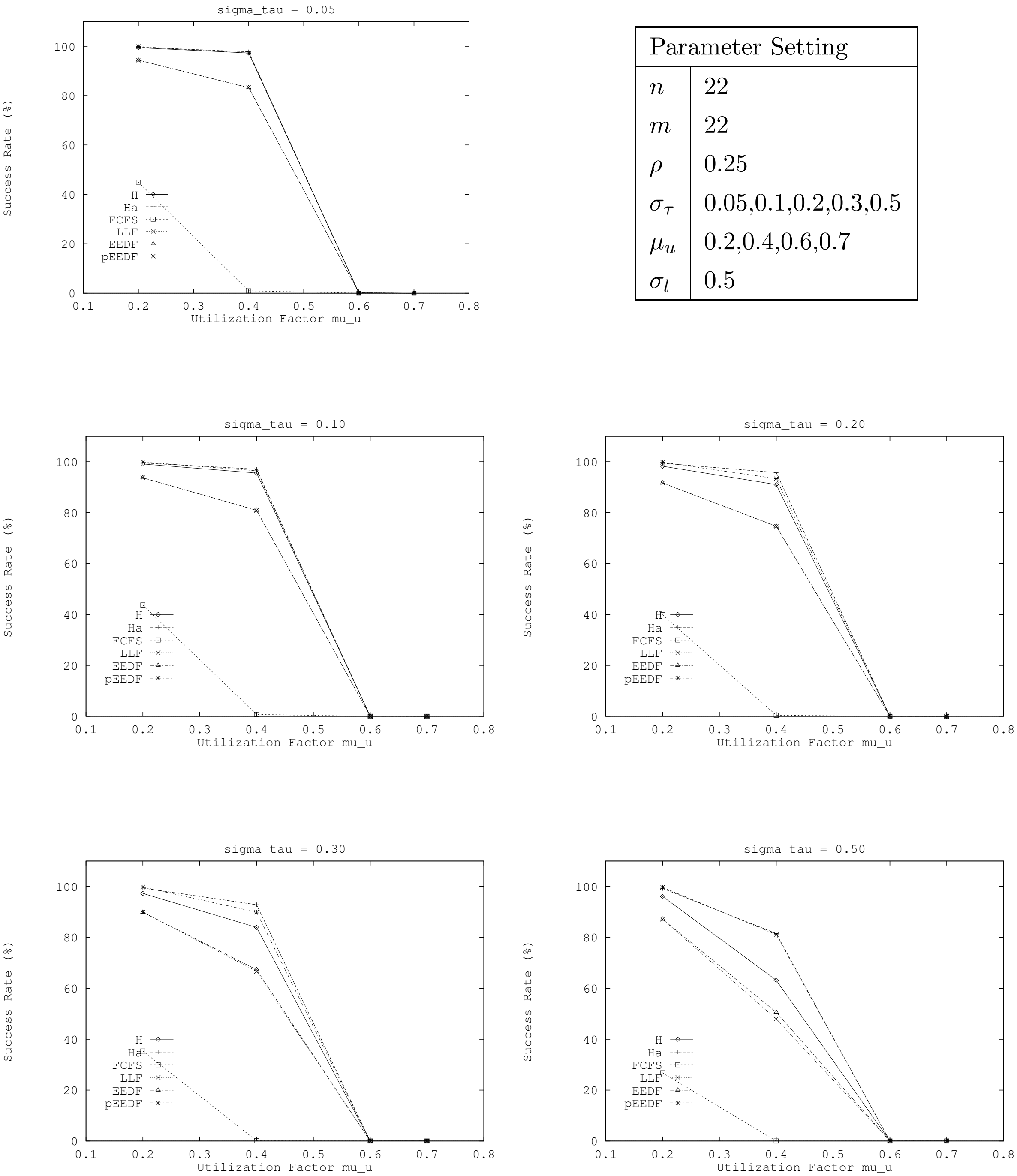
Parameter Setting	
n	22
m	4
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.16: Success Rate: 22 Tasks, 4 Processors.



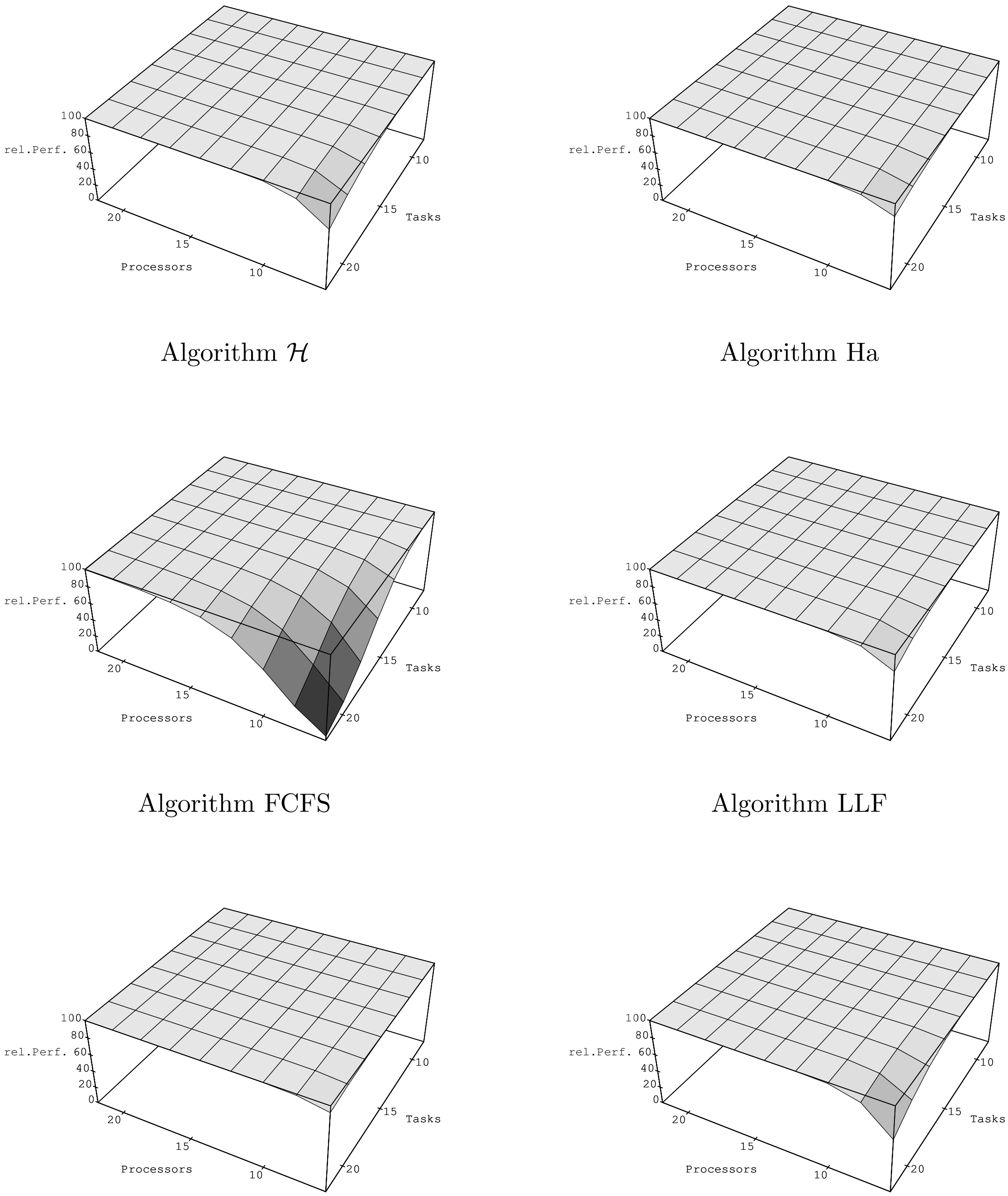
Parameter Setting	
n	22
m	14
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.17: Success Rate: 22 Tasks, 14 Processors.



Parameter Setting	
n	22
m	22
ρ	0.25
σ_τ	0.05,0.1,0.2,0.3,0.5
μ_u	0.2,0.4,0.6,0.7
σ_l	0.5

Figure B.18: Success Rate: 22 Tasks, 22 Processors.



Algorithm $\mathcal{H}$

Algorithm Ha

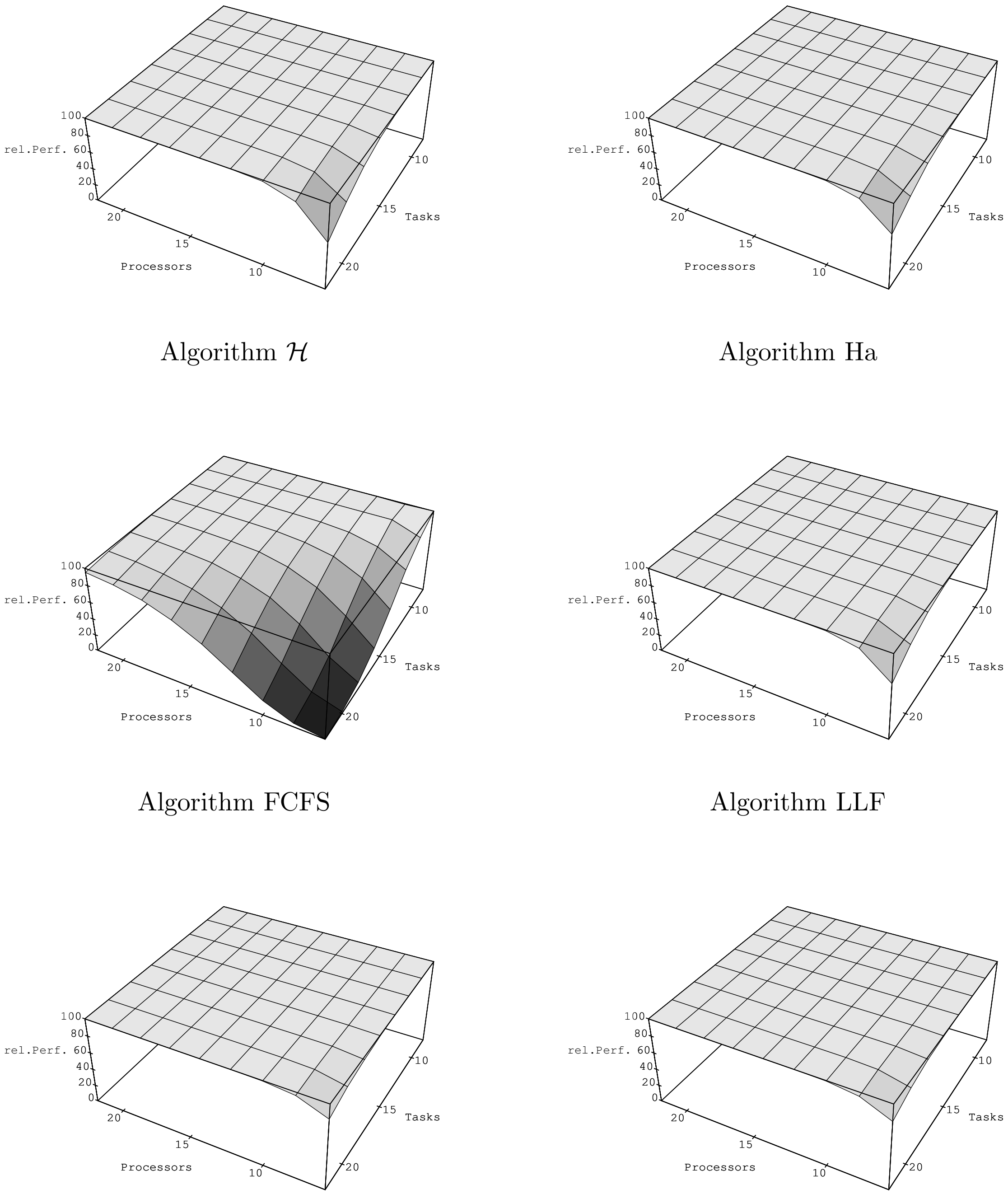
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.19: Relative Performance: $\sigma_\tau = 0.05$, $\mu_u = 0.2$.



Algorithm $\mathcal{H}$

Algorithm Ha

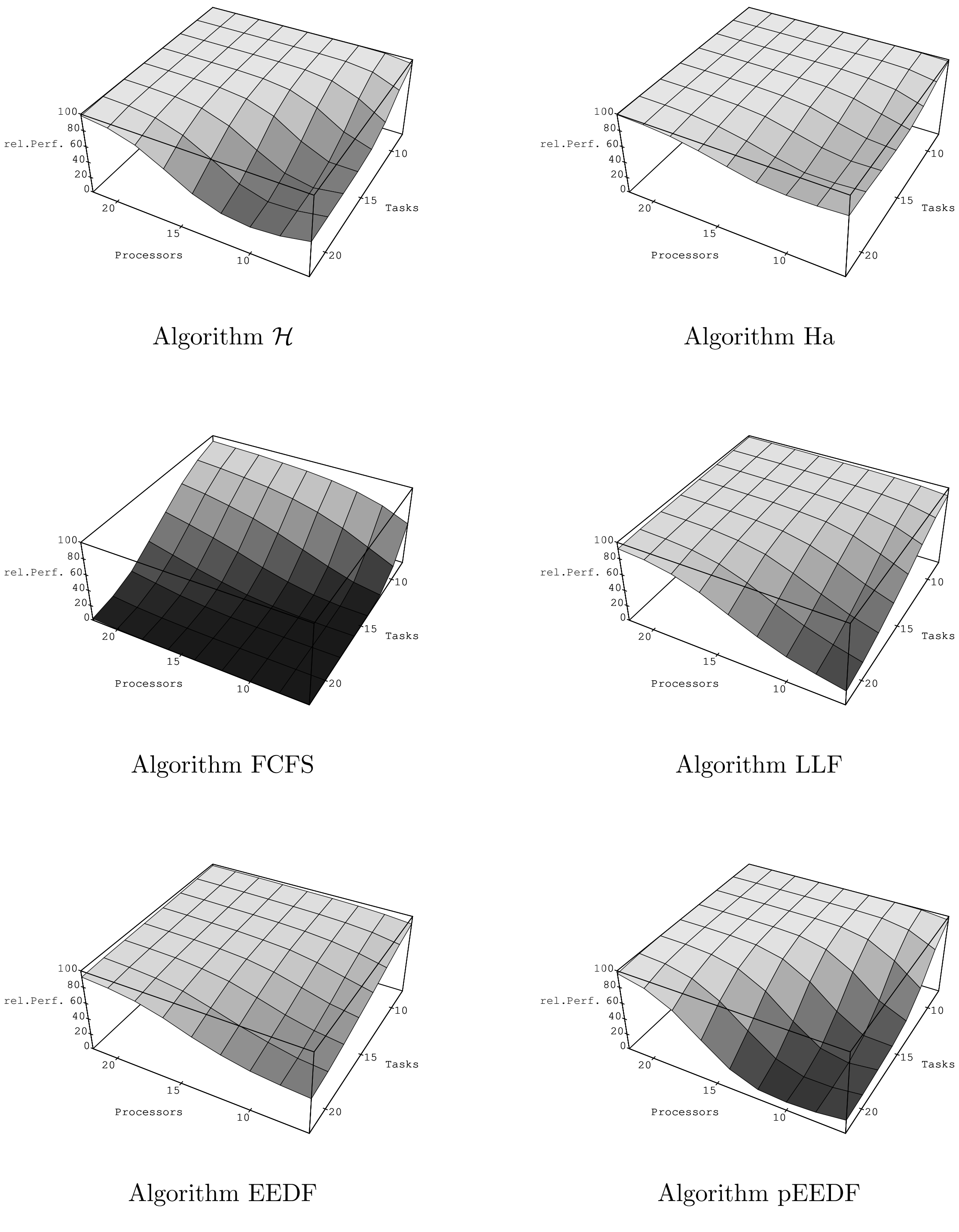
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.20: Relative Performance: $\sigma_\tau = 0.5$, $\mu_u = 0.2$.



Algorithm $\mathcal{H}$

Algorithm Ha

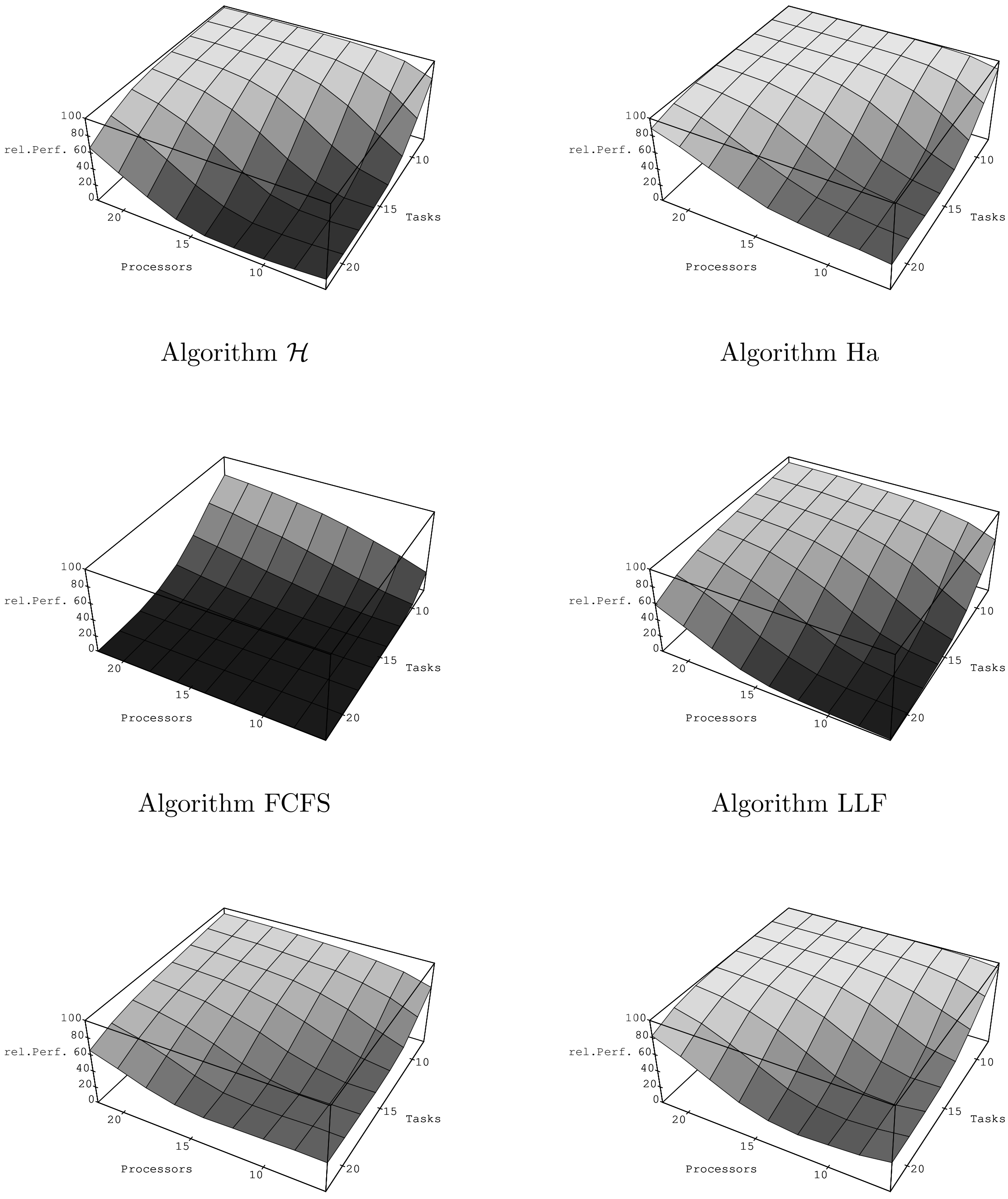
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.21: Relative Performance: $\sigma_\tau = 0.05$, $\mu_u = 0.4$.



Algorithm $\mathcal{H}$

Algorithm Ha

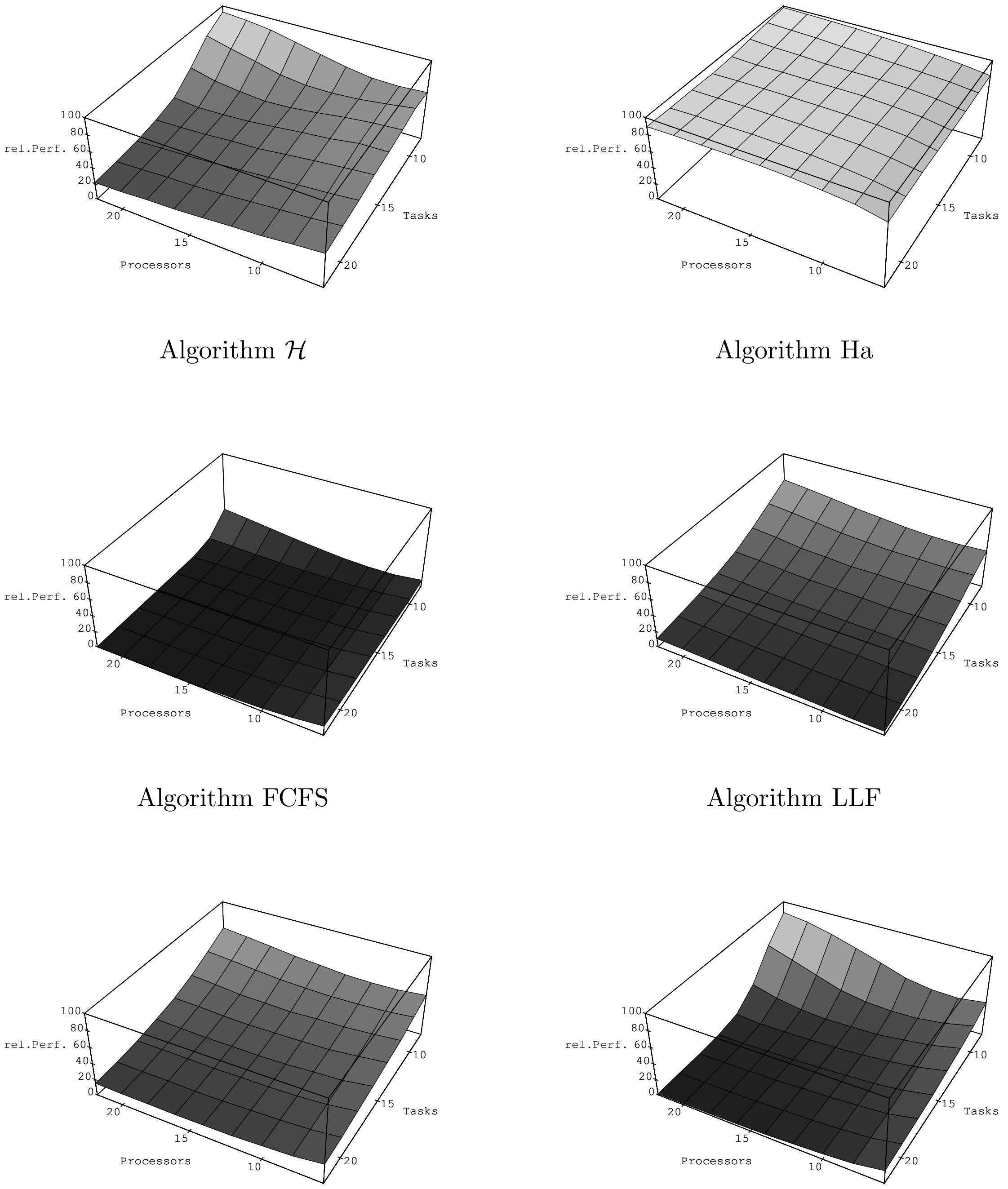
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.22: Relative Performance: $\sigma_\tau = 0.5$, $\mu_u = 0.4$.



Algorithm $\mathcal{H}$

Algorithm Ha

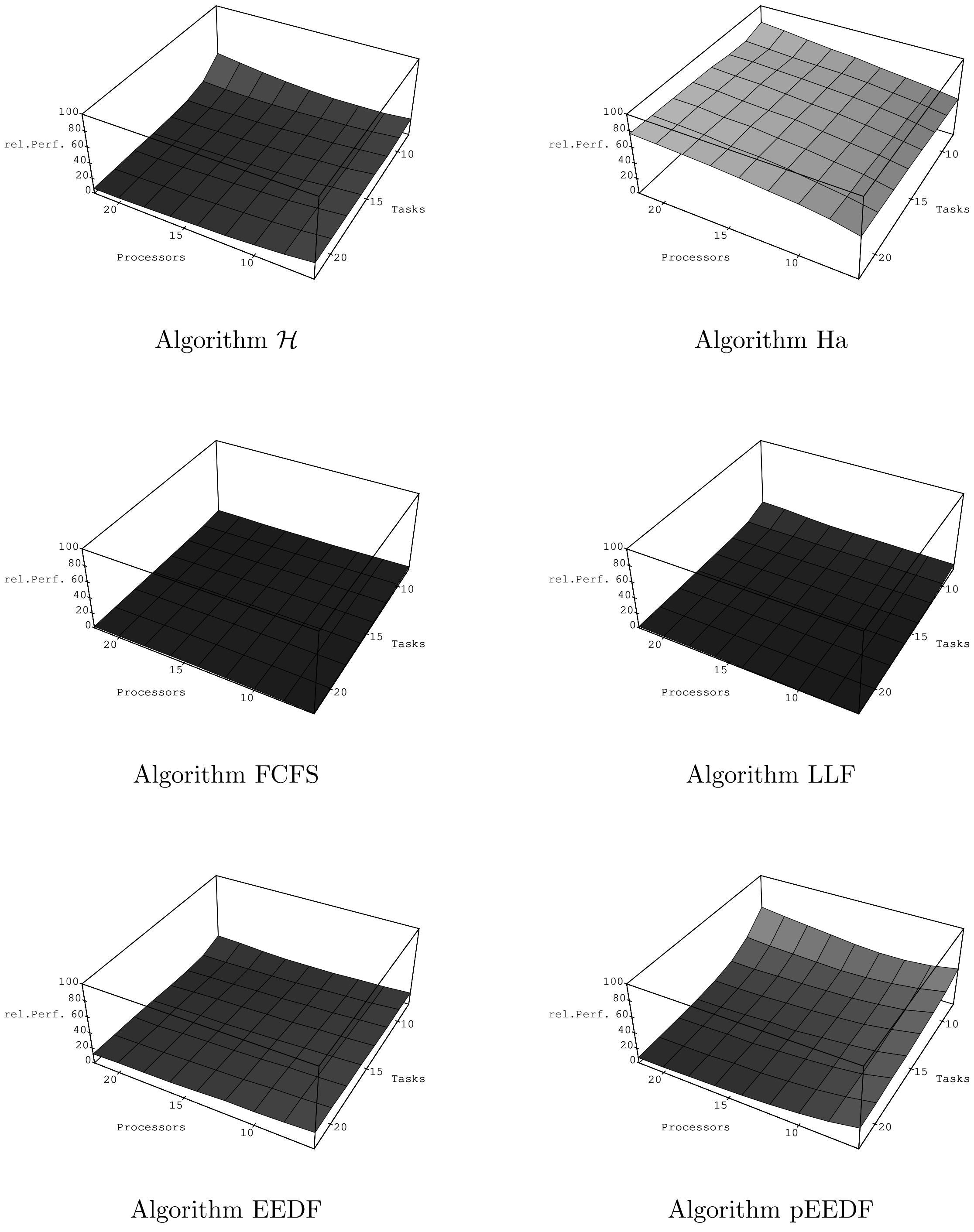
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.23: Relative Performance: $\sigma_\tau = 0.05$, $\mu_u = 0.7$.



Algorithm $\mathcal{H}$

Algorithm Ha

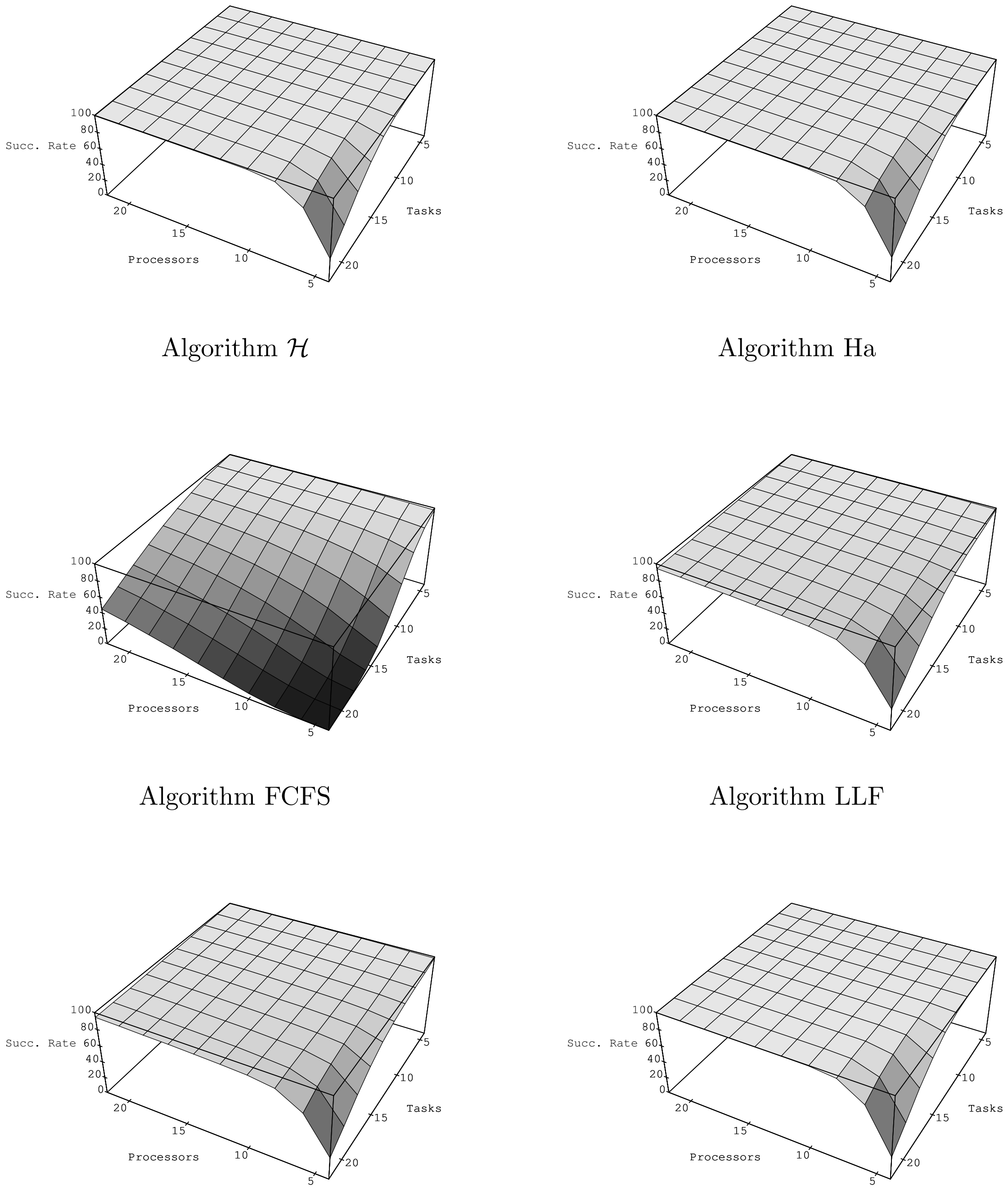
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.24: Relative Performance: $\sigma_\tau = 0.5$, $\mu_u = 0.7$.



Algorithm $\mathcal{H}$

Algorithm Ha

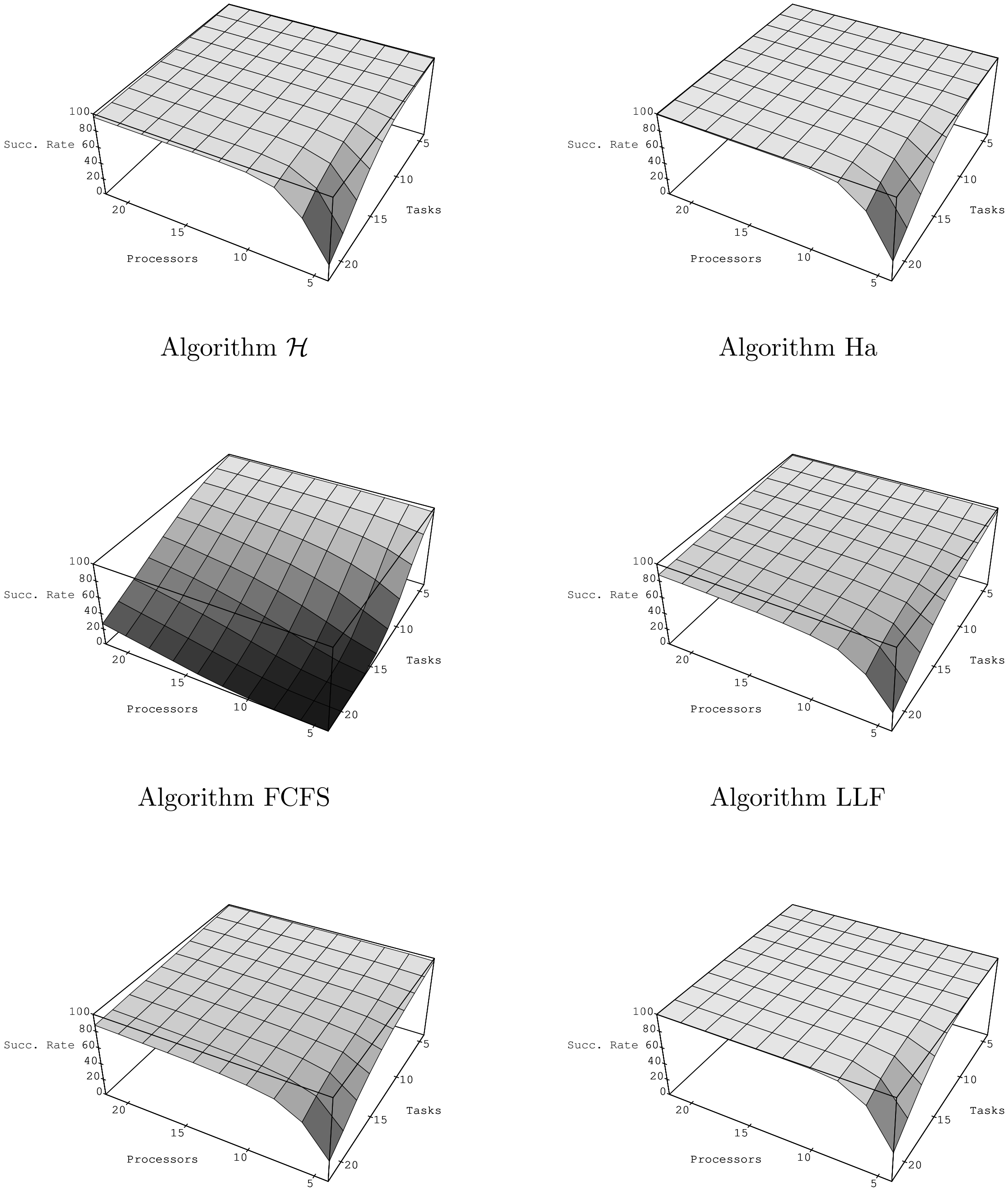
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.25: Success Rate: $\sigma_\tau = 0.05$, $\mu_u = 0.2$.



Algorithm $\mathcal{H}$

Algorithm Ha

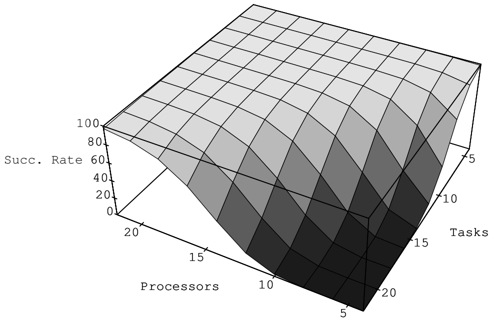
Algorithm FCFS

Algorithm LLF

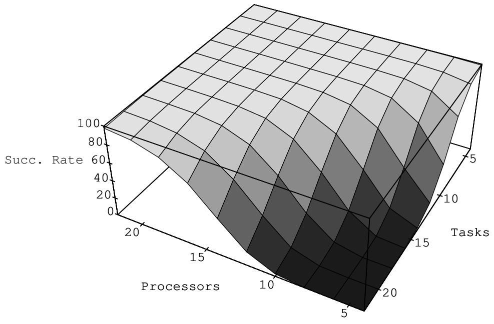
Algorithm EEDF

Algorithm pEEDF

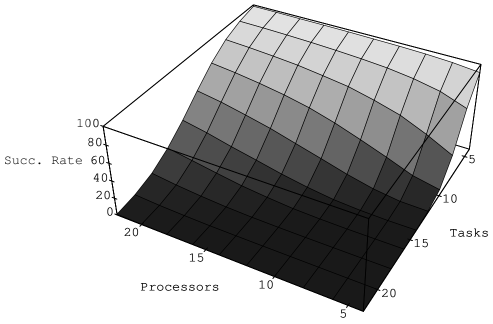
Figure B.26: Success Rate: $\sigma_\tau = 0.5$, $\mu_u = 0.2$.



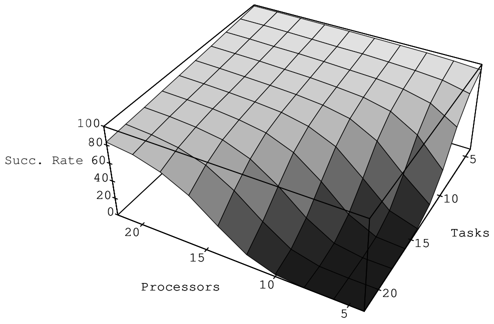
Algorithm $\mathcal{H}$



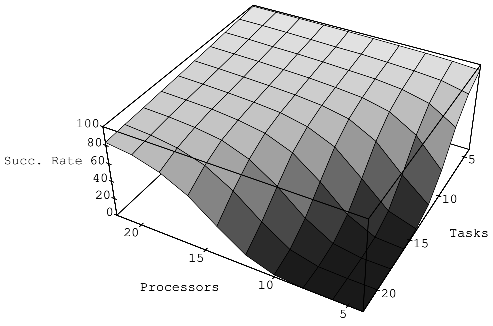
Algorithm Ha



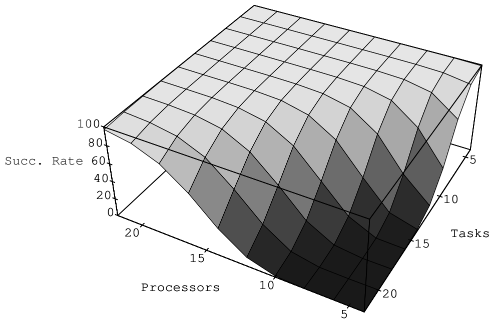
Algorithm FCFS



Algorithm LLF

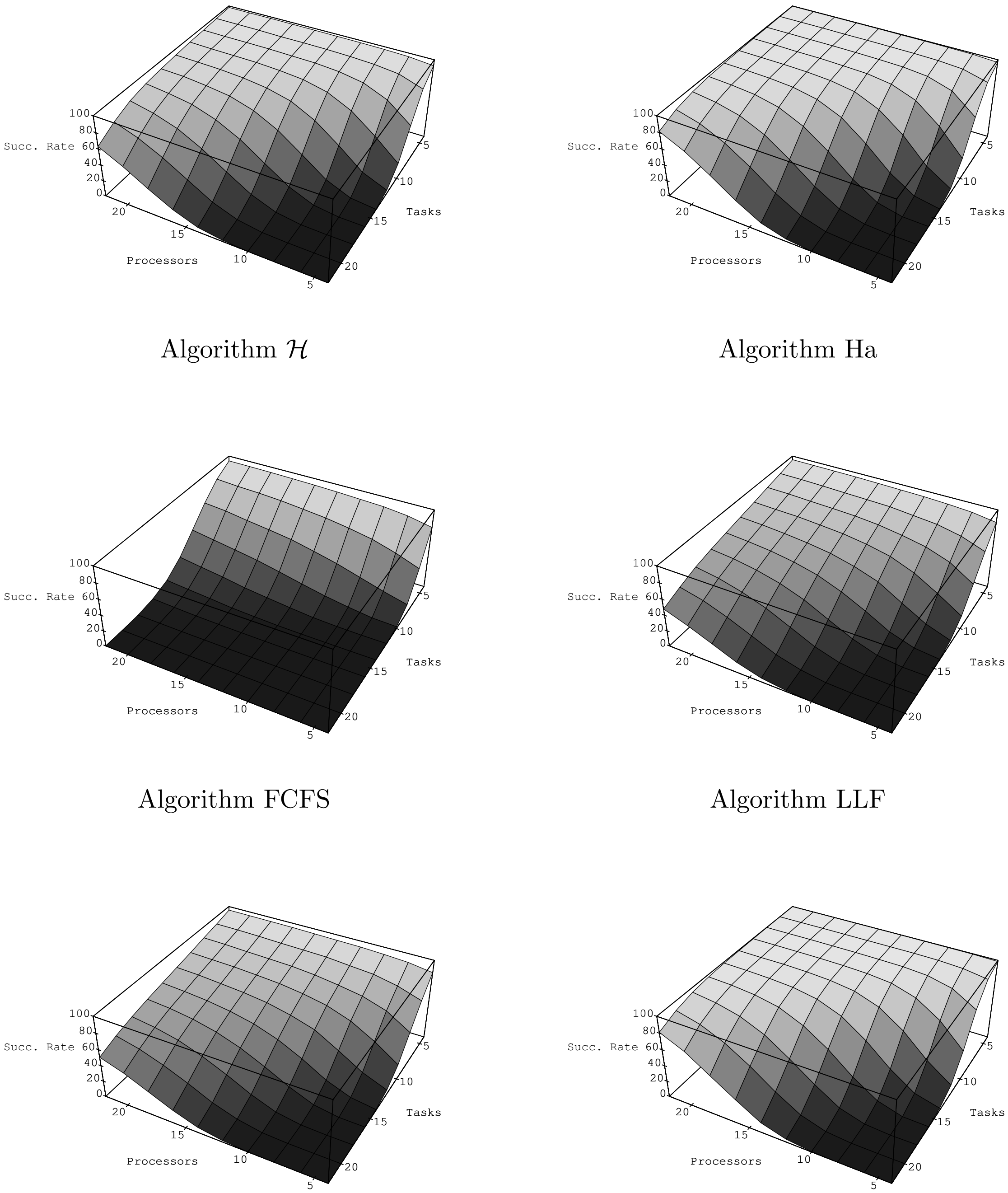


Algorithm EEDF



Algorithm pEEDF

Figure B.27: Success Rate: $\sigma_\tau = 0.05$, $\mu_u = 0.4$.



Algorithm $\mathcal{H}$

Algorithm Ha

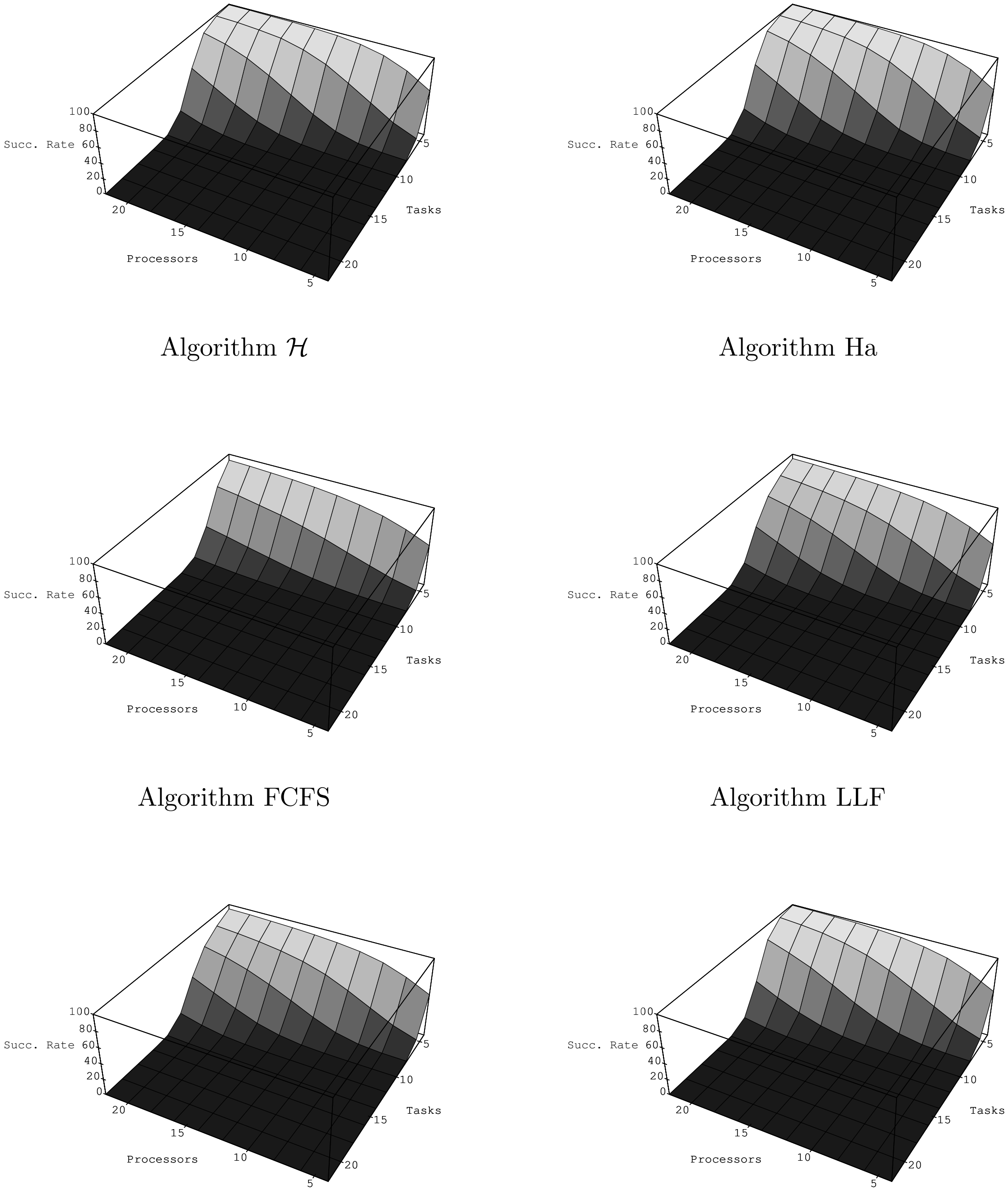
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.28: Success Rate: $\sigma_\tau = 0.5$, $\mu_u = 0.4$.



Algorithm $\mathcal{H}$

Algorithm Ha

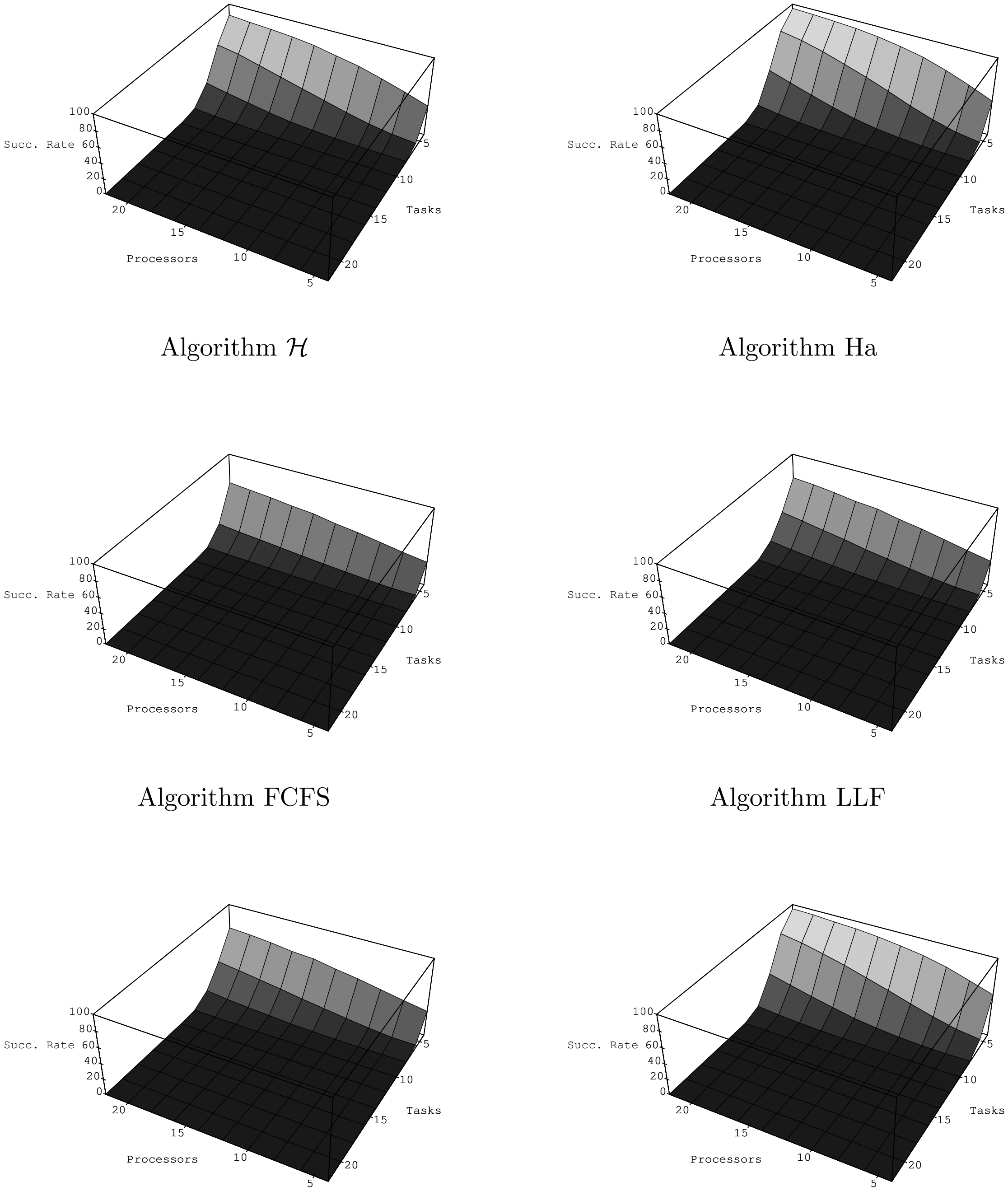
Algorithm FCFS

Algorithm LLF

Algorithm EEDF

Algorithm pEEDF

Figure B.29: Success Rate: $\sigma_\tau = 0.05$, $\mu_u = 0.7$.



Algorithm $\mathcal{H}$

Algorithm Ha

Algorithm FCFS

Algorithm LLF

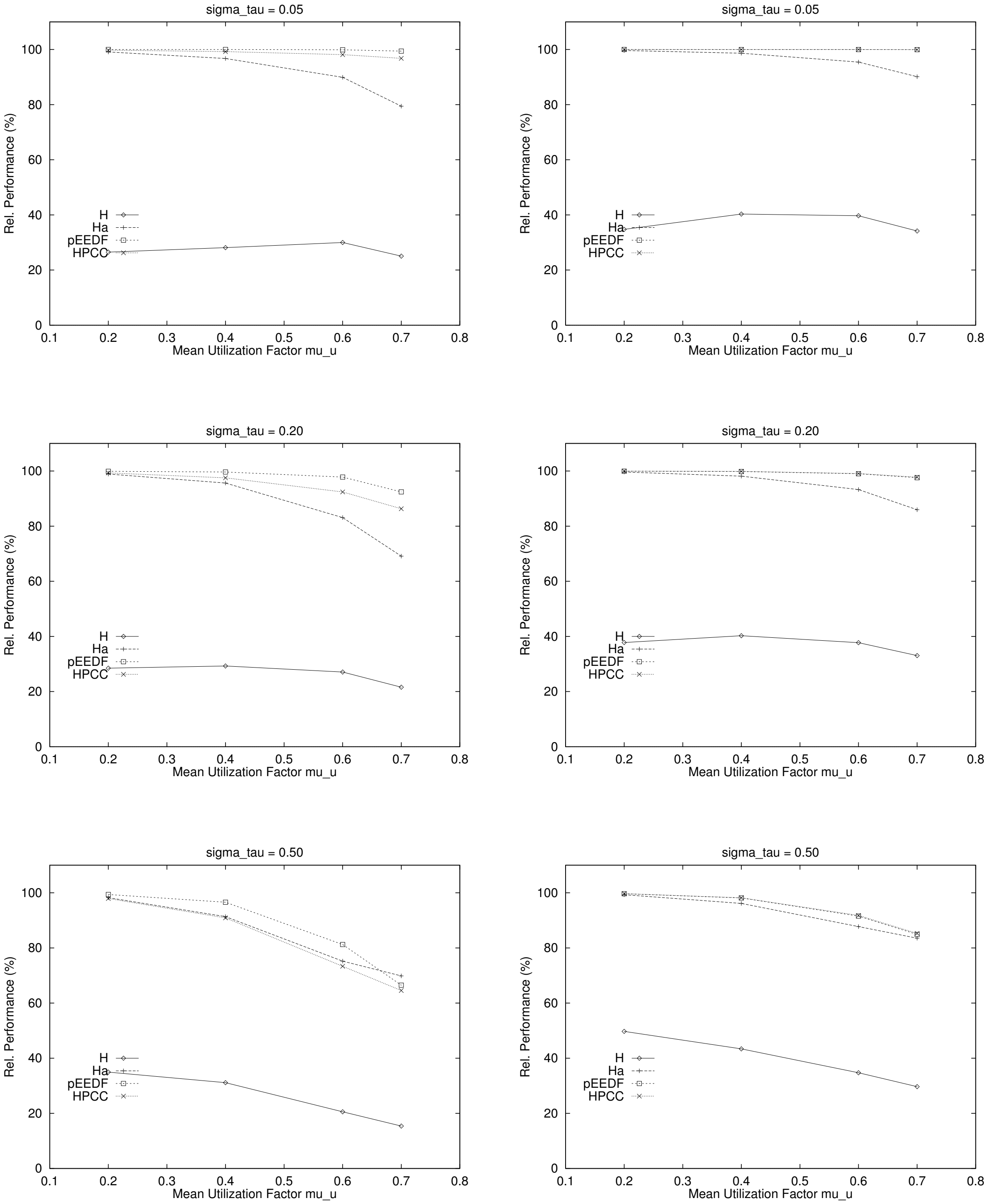
Algorithm EEDF

Algorithm pEEDF

Figure B.30: Success Rate: $\sigma_\tau = 0.5$, $\mu_u = 0.7$.

Appendix C

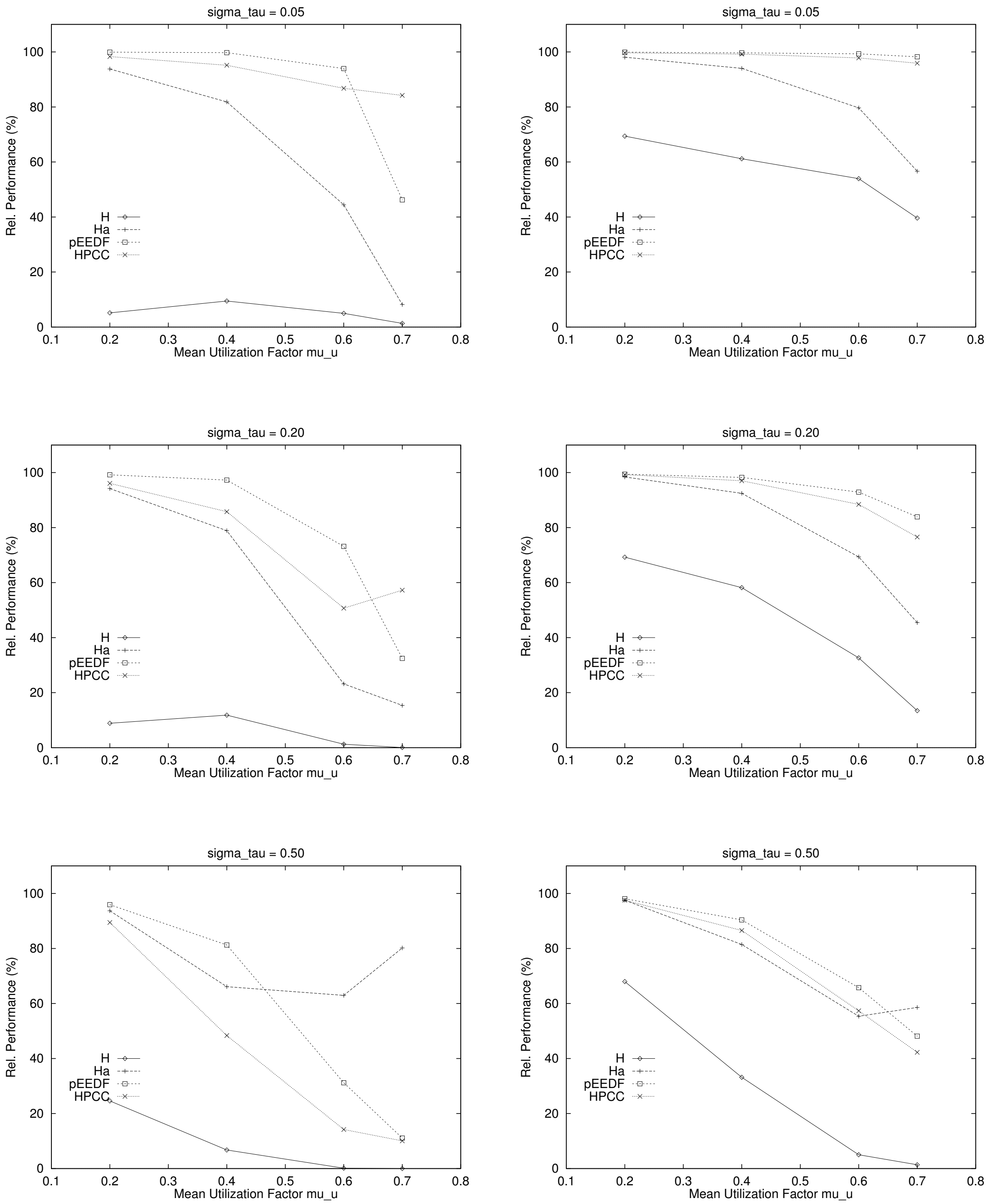
Simulation Results for Algorithm $\mathcal{HPCC}$



Few Short Tasks ($s = 0.25$).

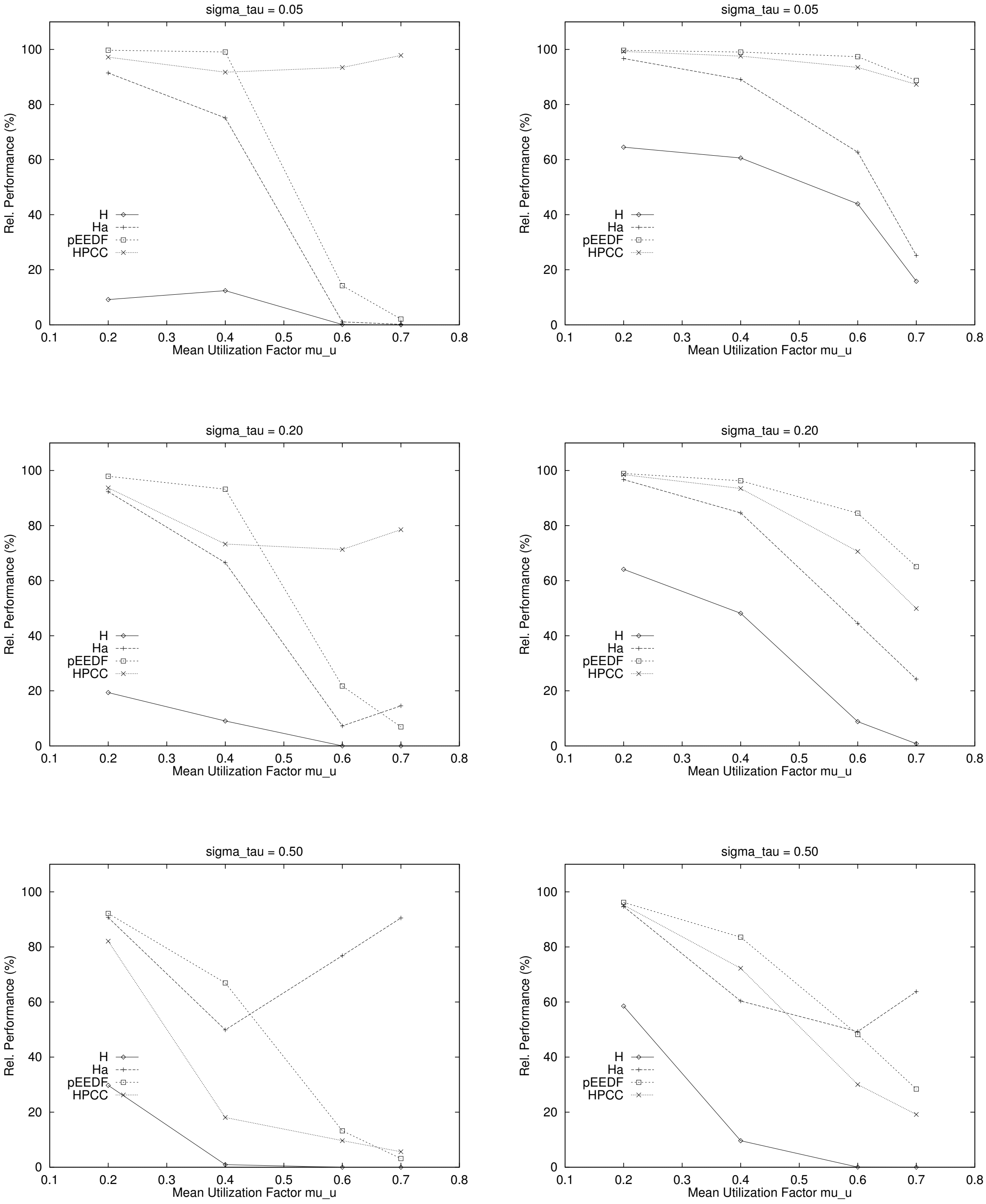
Many Short Tasks ($s = 0.75$).

Figure C.1: Relative Performance: 4 Tasks, 12 Processors.



Few Short Tasks ($s = 0.25$). Many Short Tasks ($s = 0.75$).

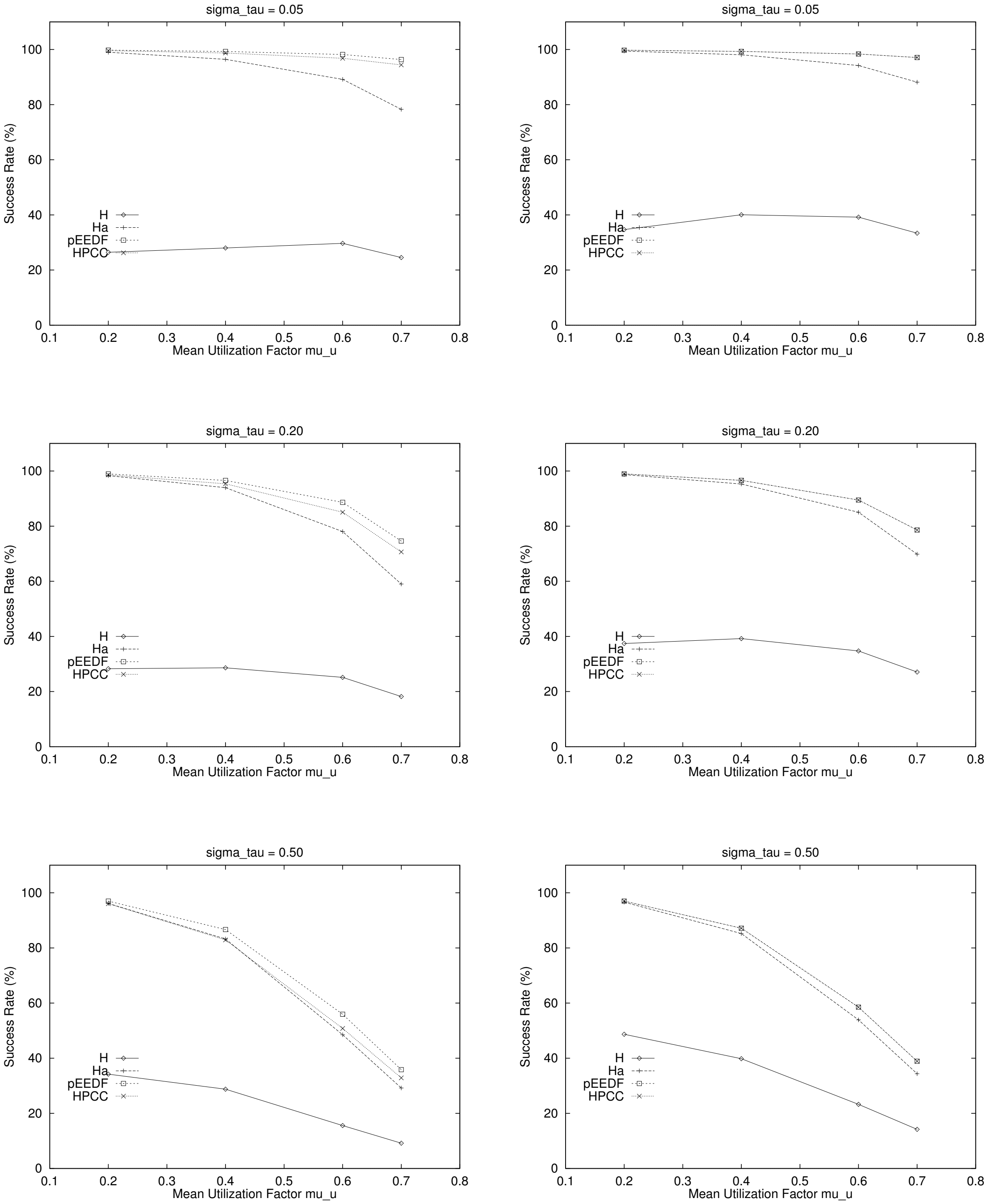
Figure C.2: Relative Performance: 12 Tasks, 12 Processors.



Few Short Tasks ($s = 0.25$).

Many Short Tasks ($s = 0.75$).

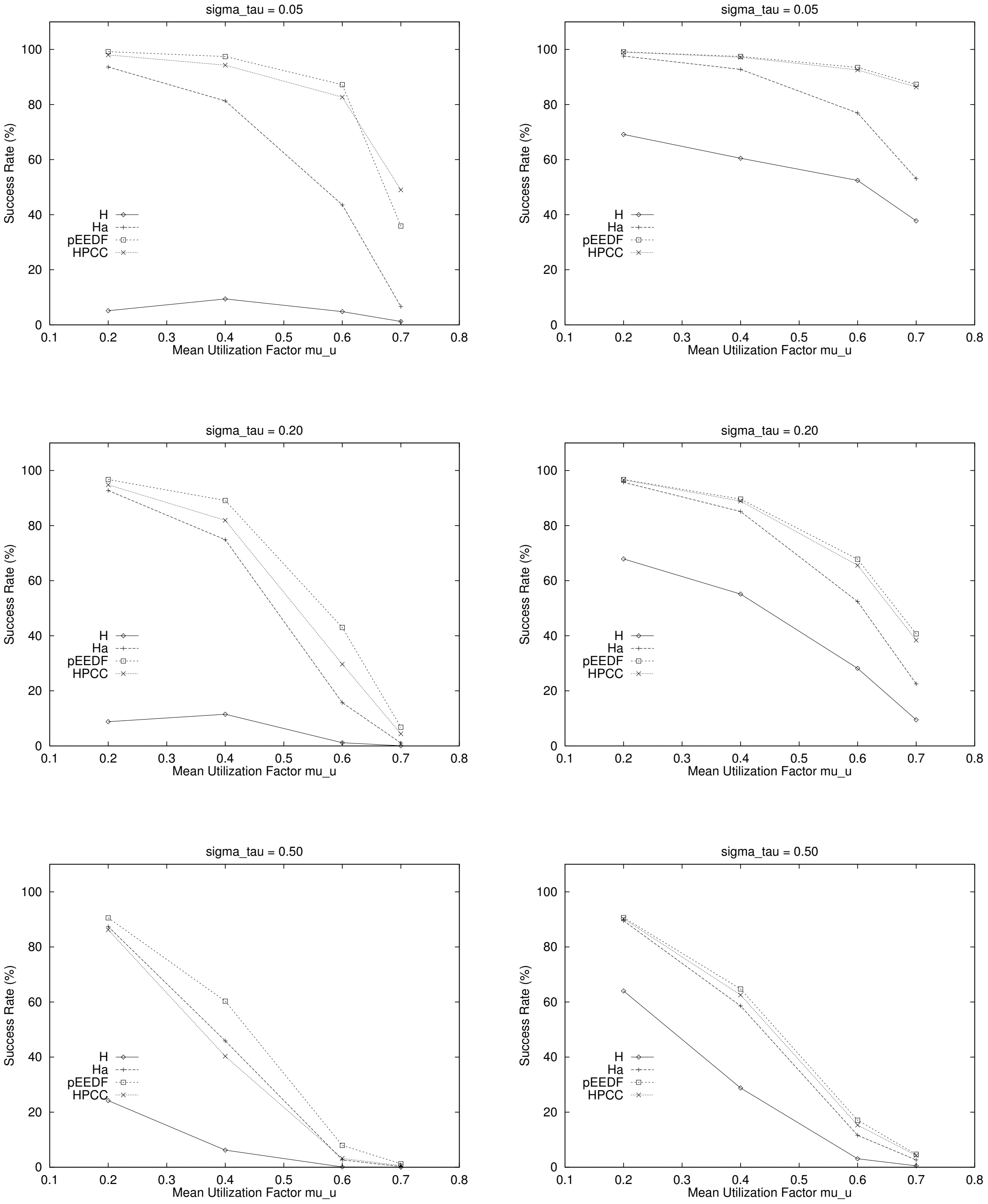
Figure C.3: Relative Performance: 20 Tasks, 12 Processors.



Few Short Tasks ($s = 0.25$).

Many Short Tasks ($s = 0.75$).

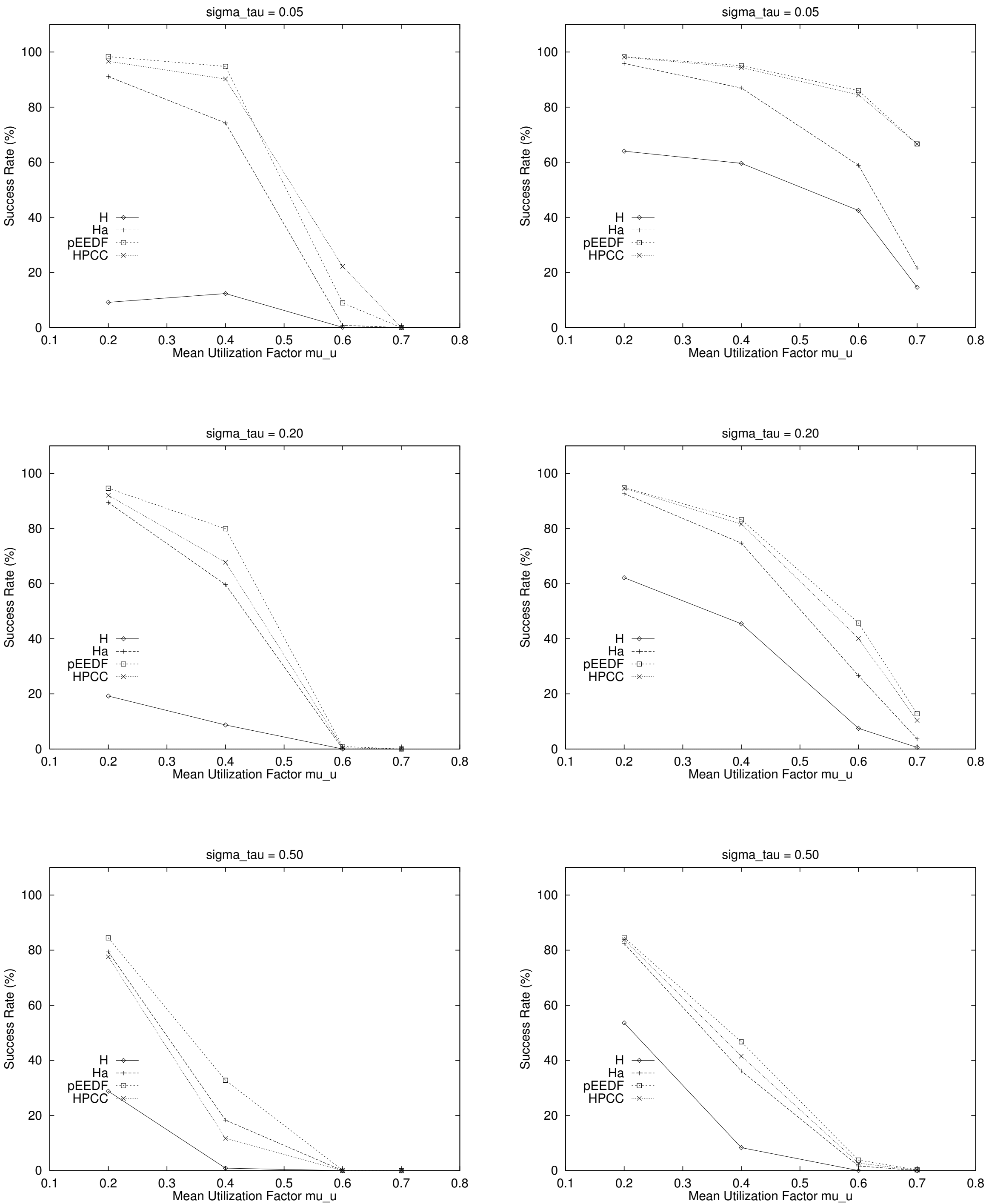
Figure C.4: Success Rate: 4 Tasks, 12 Processors.



Few Short Tasks ($s = 0.25$).

Many Short Tasks ($s = 0.75$).

Figure C.5: Success Rate: 12 Tasks, 12 Processors.



Few Short Tasks ($s = 0.25$).

Many Short Tasks ($s = 0.75$).

Figure C.6: Success Rate: 20 Tasks, 12 Processors.

Bibliography

[1] G. Agrawal, B. Chen, W. Zhao, and S. Davari. Guaranteeing synchronous message deadlines with the timed token protokol. In *Proceedings of the 12th Internatioal Conference on Distributed Computing Systems*, pages 468–475, Yokohama, Japan, June 1992.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline-monotonic approach. In *Proceedings of the Eighth IEEE Workshop on real-Time Operating Systems and Software*, May 1991.

[3] T. P. Baker. A stack-based allocation policy for realtime processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–200, December 1990.

[4] J. Bruno, J. W. Jones III, and K. So. Deterministic scheduling with pipelined processors. *IEEE Trans. Computers*, 29:120–139, 1980.

[5] A. Burns, M. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a point-to-point distributed system. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 11–20, Newport Beach, California, April 1993.

[6] H. Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operation Research*, 17-4:701–715, 1969.

[7] D. Ferrari. Real-time communication in an internetwork. Technical Report TR-92-001, International Computer Science Institute, Berkeley, January 1992.

[8] S. French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.* Wiley, 1982.

[9] M. R. Garey and D. S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *J. Assoc. Comput. Mach.*, 23:461–467, 1976.

[10] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6:416–426, 1977.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, New York, 1979.

[12] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1:117–129, 1976.

[13] M. R. Garey, D. S. Johnson, B. Simons, and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM J. Comput.*, 10-2:256–269, 1981.

[14] T. Gonzales and S. Sahni. Flowshop and jobshop scheduling: Complexity and approximation. *Operation Research*, 26-1:37–52, 1978.

[15] S. K. Goyal and C. Sriskandarajah. No-wait shop scheduling: computational complexity and approximate algorithms. *Opsearch*, 25:220–244, 1988.

[16] J. Grabowski. A new algorithm of solving the flow-shop problem. In G. Feichtinger and P. Kall, editors, *Operations Research in Progress*, pages 57–75. Reidel, Dordrecht, 1982.

[17] J. Grabowski, E. Skubalska, and C Smutnicki. On flow shop scheduling with release and due dates to minimize maximum lateness. *J. Oper. Res. Soc.*, 34:615–620, 1983.

[18] J. N. D. Gupta and S. S. Reddi. Improved dominance conditions for the three-machine flowshop scheduling problem. *Oper. Res.*, 26:200–203, 1978.

[19] C. C. Han. *Scheduling Real-time computatios with Temporal Distance and Separation Constraints and with Extended Deadlines.* PhD thesis, University of Illinois at Urbana-Champaign, June 1992.

[20] C. C. Han and K. J. Lin. Job scheduling with separation constraints. Technical Report UIUCDCS-R-1635, Department of Computer Science, University of Illinois, 1990.

[21] D. Hoitomt, P. B. Luh, and K. R. Pattipati. Job shop scheduling with simple precedence constraints. In , pages 1–6, 1991.

[22] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[23] J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple access protocols and time-constrained communication. *ACM Computing Surveys*, 16:43–70, March 1984.

[24] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline scheduling: A survey. Technical Report RJ 5738, IBM Research Division, San Jose, CA, 1987.

[25] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical report, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[26] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 1986.

[27] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In A. M. Tilborg and G. M. Koob, editors, *Foundations of Real-Time Computing, Scheduling and Resource Management*, chapter 1. Kluwer Academic Publishers, 1991.

[28] J. P. Lehoczky, L Sha, and J.K. Strosnider. Enhanced aperiodic scheduling in hard-real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, December 1987.

[29] J. Y.-T. Leung, O. Vornberger, and J. Witthoff. On some variants of the bandwidth minimization problem. *SIAM J. Comput.*, 13:650–667, 1984.

[30] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, December 1982.

[31] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. Assoc. Comput. Mach.*, 20:46–61, 1973.

[32] J.W.S. Liu, J.L. Redondo, Z. Deng, T.S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W.K. Shih. Perts: A prototyping environment for real-time systems. Technical Report UIUCDCS-R-1802, Department of Computer Science, University of Illinois, 1993.

[33] J.W.S. Liu, J.L. Redondo, Z. Deng, T.S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W.K. Shih. Perts: A prototyping environment for real-time systems. In *Proceedings of the 14th Real-Time Systems Symposium*, December 1993.

[34] J. A. McHugh. *Algorithmic Graph Theory.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[35] G.B. McMahon. *A Study of Algorithms for Industrial Scheduling Problems.* PhD thesis, University of New South Wales, Kensington, 1971.

[36] C. B. McNaughton. Scheduling with deadlines and loss functions. *Management Sci.*, 6:1–12, 1959.

[37] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment.* PhD thesis, M.I.T., 1993.

[38] J. K. Y. Ng and J. W.-S. Liu. Performance of local area network protocols for hard real-time applications. In *11th International Conference on Distributed Computing Systems*, pages 318–326, 1991.

[39] K. V. Palem and B. Simons. Scheduling time-critical instructions on risc machines. In *ACM Symposium on Principles of Programming Languages*, pages 270–280, 1990.

[40] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[41] D. T. Peng and K. G. Shin. A new performance measure for scheduling independent real-time tasks. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan, 1989.

[42] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization of multiprocessors. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 259–269, December 1988.

[43] J. L. Redondo. Schedulability analyzer tool. Master's thesis, University of Illinois at Urbana-Champaign, Feb 1993.

[44] J. Riggs. *Production Systems Planning.* Wiley, third edition, 1981.

[45] N. M. Sadeh and M. S. Fox. Variable and value ordering heuristics for hard constraint satisfaction problems: An application to job shop scheduling. Technical Report CMU-RI-TR-91-23, Carnegie Mellon University, 1991.

[46] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of Real-Time Systems Symposium*, December 1986.

[47] L. Sha, L. Rajkumar, R., and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, September 1990.

[48] L. Sha, L. Rajkumar, R., J. P. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *The Journal of Real-Time Systems*, 1:243–264, 1989.

[49] H.R. Simpson. A Data Interactive Architecture (DIA) for real-time embedded multiprocessor systems. In *Computing Techniques in Guided Flight RAe Conference*, April 1990.

[50] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[51] W. Szwarc. Optimal elimination methods in the $m \times n$ flow-shop scheduling problem. *Operation Research*, 21:1250–1259, 1973.

[52] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[53] C. M. Woodside and D. W. Graig. Local non-preemptive scheduling policies for hard real-time distributed systems. In *Proceedings of Real-Time Systems Symposium*, December 1987.

[54] Y. Xiong, N. Sadeh, and K. Sycara. Intelligent backtracking techniques for job shop scheduling. In *Proceedings of the Third Int. Conf. on Principles of Knowledge Repr. and Reasoning*, 1992.

[55] W. Zhao, J. A. Stankovic, and K. Ramamritham. A window protocol for transmission of time constrained messages. *IEEE Trans. Computers*, 39:1186–1203, September 1990.

Vita

Riccardo Bettati was born in Aarau, Switzerland on March 27, 1963. He received his diploma in informatics from the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland, in 1988. He was research assistant at the Electronics Institute at ETH until he began his Ph.D. studies in Computer Science at the University of Illinois at Urbana-Champaign in August 1988. He joined the Real-Time Systems Laboratory in 1989 and completed his Ph.D. program in 1994.