PROBABILISTIC ANALYSIS AND SCHEDULING OF CRITICAL SOFT REAL-TIME SYSTEMS

BY

MARK K. GARDNER

B.S., Brigham Young University, 1986
M.S., Brigham Young University, 1994

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, 1999

Urbana, Illinois

PROBABILISTIC ANALYSIS AND SCHEDULING
OF CRITICAL SOFT REAL-TIME SYSTEMS

Mark K. Gardner, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1999
Jane W.S. Liu, Advisor

In addition to correctness requirements, a real-time system must also meet its temporal constraints, often expressed as deadlines. We call safety or mission critical real-time systems which may miss some deadlines *critical soft real-time systems* to distinguish them from hard real-time systems, where all deadlines must be met, and from soft real-time systems which are not safety or mission critical. The performance of a critical soft real-time system is acceptable as long as the deadline miss rate is below an application specific threshold.

Architectural features of computer systems, such as caches and branch prediction hardware, are designed to improve average performance. Deterministic real-time design and analysis approaches require that such features be disabled to increase predictability. Alternatively, allowances must be made for for their effects by designing for the worst case. Either approach leads to a decrease in average performance. Since critical soft real-time systems do not require that all deadlines be met, average performance can be improved by adopting a probabilistic approach. In order to allow a trade-off between deadlines met and average performance, we have developed a probabilistic analysis technique, call Stochastic Time Demand Analysis, for determining a lower bound on the rate at which

deadlines are met in fixed priority systems.

Allowing a real-time system to miss some deadlines in exchange for better average performance increases the possibility of overload. While overload in real-time systems has been studied, the emphasis has been on hard real-time systems in which overload is an exception whose occurrence is to be minimized. In contrast, critical soft real-time systems can be repeatedly overloaded during normal operation. Therefore, we have evaluated the performance of various real-time scheduling algorithms for critical soft real-time systems, including two new classes of algorithms, on workloads with execution and inter-release time variations, both with and without dependencies.

To my parents, Mark and Wilma,
who made it possible.
To my children, Melanie, Diane and Stephanie,
who made it fun.
But most of all to my wife, Laurinda,
who makes it all worth while.

Acknowledgements

I am grateful to my advisor, Jane W.S. Liu, for her patience, encouragement and support. Not only has she taught me to be thorough and methodical, but she has pushed me to do more than my best. I am especially indebted to her for teaching me the research process and opening my eyes to its human element. I would also like to thank Lui Sha, Klara Nahrstedt, and Geneva Belford for kindly agreeing to be the other members of my committee. My work has been made better by their clear insight and helpful criticisms.

I would like to thank my colleagues in the Real-Time Systems Laboratory, especially Wu-chun Feng who taught me the ropes and served as a mentor, David Hull who always seemed to know the answers to all my questions about Unix and its software, and Mallikarjun (Arjun) Shankar for encouragement and for many pleasant conversations. I would also like to thank Scott Pakin and Geetanjali (Geta) Sampemane in the Concurrent Systems Architecture Group who loaning me the hardware to build my "simulation cluster" and for enlightening me on some of the more obscure aspects of Linux. I would also like to thank Steve Cvetko for loaning me needed computer parts. With out the help of my friends, I would have accomplished much less while taking longer to do it.

Next, I would like the thank the staff of the Department of Computer Science for their administrative and technical help. I would especially like to thank Molly Flesner, our secretary, for service above and beyond the call of duty. I would also like to thank Barb Cicone, Julie Legg, Kay Tomlin and Felice Long in the academic office. Their smiling faces and restorative powers helped me recover from the many times I did not follow proper protocol. I also owe Rich Myers, Sandra and Chuck Thompson, Lawrence Bowie and Amanda (Andy) Coyle of the Computing Resources Laboratory a debt of gratitude for maintaining the hardware and software I used to do my work and for tolerating my almost incessant pesterings.

Finally, I would like to thank my family for foregoing many of the comforts of life and allowing me to return to school. I hope they will fondly remember the experiences we have shared. I am especially grateful to my loving companion, Laurinda, without whose support this achievement would not be so sweet. Last of all, I must thank my Creator for the beauties and intricacies of life that make living so much fun.

Table of Contents

Chapter 1

Introduction

The fundamental difference between real-time and non-real-time computer systems is the requirement that a system meet its temporal constraints. The most common form of constraint is the deadline; each job (e.g., computation, data transmission or file block retrieval) must complete its execution by its deadline to have met its temporal constraints.

Traditionally, safety and mission critical real-time systems are designed to ensure that there are no missed deadlines because there are no techniques for accurately determining the probabilities that deadlines will be missed. For many systems, meeting all deadlines is an overly stringent requirement resulting in low resource utilization and poor average performance. We call safety or mission critical systems which may miss some deadlines *critical soft real-time systems* to distinguish them from hard real-time systems, where all deadlines must be met, and from soft real-time systems which are not safety or mission critical. Examples of critical soft real-time systems are found in the telecommunication, signal processing, and process control domains. As long as the rate at which deadlines are missed is below a threshold, the real-time performance of a critical soft real-time system is considered acceptable.

Critical soft real-time systems cannot be analyzed with existing (deterministic) real-time analysis techniques because the techniques indicate whether or not deadlines will be missed, not the frequency of missed deadlines. Queueing theoretic approaches used to analyze time-share systems compute the mean and variance of performance not the probability of a missed deadline. While simulations or measurements can provide the desired performance characterization, they are expensive and adequate coverage is difficult to ensure, particularly when the expected miss rate is small. Thus, techniques for bounding the probability of a missed deadline in critical soft real-time systems must be developed to allow systems to meet their temporal constraints while improving average performance.

Allowing a real-time system to miss some deadlines in exchange for better average performance increases the possibility of overload. A real-time system is overloaded when insufficient processing time is available to complete all jobs by their deadlines. The goal for critical soft real-time systems in overload is to meet as many deadlines as possible while minimizing response times. While overload in real-time systems has been studied, the emphasis has been on hard real-time systems in which overload is an exception whose occurrence is to be minimized.[1] In contrast, critical soft real-time systems by definition will be repeatedly overloaded during normal operation. Therefore, the performance of various real-time scheduling algorithms needs to be evaluated and compared specifically for critical soft real-time systems.

1.1 Motivation

Modern computer systems incorporate many architectural features designed to improve average performance. For example, it is common for CPUs to contain multiple functional units, multi-level branch prediction, and out-of-order pipelined execution. Multiple levels of caches are routinely used to reduce memory access delays. Network requests from many connections are aggregated on a shared link to maximize link utilization and minimize cost. In each of these cases, increases in average performance come at the expense of predictability and worst-case performance.

Traditional real-time design and analysis approaches require that these features (e.g., caching) be disabled to increase predictability or allowances be made for increased variation (e.g., by using worst-case memory access times). Either approach leads to a decrease in average performance. Since many safety or mission critical real-time systems do not require that all deadlines be met, average performance can be improved by adopting a probabilistic approach which accommodates the variability of modern architectural features.

As an example of the usefulness of a probabilistic analysis approach, consider the design of an automotive engine management system. One of the important

[1] A real-time system designed to meet all deadlines may still become overloaded due to unforeseen events. Rather than uncovering a flaw in the design, this indicates an error in the specification or a bug in the implementation.

functions of the system is to compute the duration of fuel injection pulses at each intake port from sensor readings of air flow rate and fuel pressure. To be of use, a duration must be available before the beginning of the appropriate intake stroke. However, some computations can miss their deadlines without adverse effects because the amount of fuel an engine requires changes slowly from one cycle to the next. Suppose that a deterministic real-time analysis of the engine management system shows that some of the duration computations miss their deadlines on a target processor. Based on the analysis, a faster processor must be selected to ensure the proper operation of the engine because insufficient information is available to decide otherwise. In contrast, suppose that it can be ascertained that the computation will meet its deadline at least 99% of the time when executed on the target processor. It is likely that missing a deadline one percent of the time will have little effect on the performance of the engine. Hence the target processor can be used. To enable such trade-offs, probabilistic analysis techniques need to be developed.

Designing a system for good average utilization causes the potential for overload. It is well known that tasks in an overloaded system miss deadlines in a predictable manner when scheduled on a fixed priority basis. In contrast, deadline-driven systems behave unpredictably when overloaded. It would appear that fixed priority scheduling is preferable to deadline-driven scheduling except for the fact that the maximum schedulable utilization of a deadline-driven system is 100% while the maximum schedulable utilization of a fixed priority system can be significantly less. To facilitate the design of critical soft real-time systems which experience repeated overload due to execution or inter-release time variations, the performance of various schedulers under overload needs to be characterized.

1.2 Objectives and Contributions

The objective of this thesis is to develop techniques for analyzing and scheduling critical soft real-time systems. Specifically, this work

1. extends the periodic task model [1] to describe systems of hard, soft and non-real-time tasks in a uniform manner,

2. develops an analysis method for bounding the frequency of missed deadlines for three classes of fixed priority systems: 1) uniprocessor systems

with independent tasks, 2) uniprocessor systems with shared resources, and 3) distributed systems of independent tasks with end-to-end deadlines,

3. proposes two classes of algorithms for scheduling job overruns and evaluates their performance by comparison with the class of existing hard real-time scheduling algorithms on workloads with and without dependencies,

4. evaluates the performance of the previous two classes of scheduling algorithms in relation to the algorithms in the base class on workloads with release time jitter and execution time variations, both with and without dependencies.

The extended periodic task model allows systems of hard, soft and non-real-time tasks to be described in a uniform and seamless manner. It provides a framework for developing analysis and scheduling techniques for heterogeneous systems. The new method allows fixed priority critical soft real-time systems to be designed with good resource utilization and average performance. It allows the effect of performance enhancing features, such as multi-level memory hierarchies in processors or statistical multiplexing in networks, to be accounted for in determining the probability that jobs of a task will meet their deadlines. Finally, the information obtained through the evaluation of various real-time scheduling algorithms aids designers of critical soft real-time systems in selecting the appropriate scheduling algorithm for the best performance.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 presents related work and discusses background material concerning the analysis and scheduling of real-time systems. In particular, the chapter summarizes deterministic real-time analysis and scheduling techniques and their applicability in the context of critical soft real-time systems. Building upon the periodic task model, Chapter 3 presents an extended model of real-time systems which allows hard, soft and non-real-time computations to be described in a uniform and convenient manner. The extended periodic task model is the basis for the analysis and scheduling results in subsequent chapters.

Chapter 4 presents a statistical analysis technique for bounding the frequency of missed deadlines in fixed-priority systems. The technique, called *Stochastic*

Time Demand Analysis, is first developed for systems of independent tasks on a single processor. It is then extended to systems in which tasks access shared resources within non-preemptable critical sections. Finally, it is applied to distributed systems with end-to-end deadlines.

In Chapter 5, the performance of three classes of algorithms for scheduling real-time systems in which jobs overrun are compared for workloads in which the execution times of jobs in a task are independent. The performance of the algorithms on workloads in which the execution times of jobs in a task exhibit dependencies is also presented.

In Chapter 6, we initially consider the effect of release time variations with execution times held constant. Then we look at the performance of scheduling algorithms in the presence of both release time variations and overrun.

Finally, Chapter 7 summarizes the contributions of this work and points to directions of future research.

Chapter 2

Related Work

Substantial work has been done on deterministic models for the analysis of hard real-time systems. This work is summarized in Sections 2.1 and 2.2. Section 2.3 summarizes existing probabilistic analysis techniques. Section 2.4 discusses approaches for the scheduling of overloaded systems. Since the behavior of real-time systems can also be described using queueing theory, Section 2.5 presents applicable queueing theoretic results and discusses why we have chosen to extend deterministic real-time results rather than taking a queueing theoretic approach to the analysis of real-time systems.

2.1 Periodic Task Model

The *periodic task model* [1], with various extensions [2–29], is the foundation for state-of-the-art techniques for characterizing the behavior of hard real-time systems. According to the periodic task model, a real-time system consists of a set of *tasks*, each of which is a stream of computations or communications called *jobs*. We denote the ith task of the system by T_i and the jth job of the task (or the jth job since some time instant) by $J_{i,j}$. The execution time of a job is the amount of time the job takes to complete if it executes alone. All the jobs in a task have common minimum and maximum execution times denoted E_i^- and E_i^+. We will refer to the actual execution time of $J_{i,j}$ as $e_{i,j}$. Jobs in a task are released for execution (i.e., arrive) with a common minimum inter-release time. The minimum inter-release time (or inter-arrival time) is greater than zero and is called the *period* of the task, P_i^-. A job $J_{i,j}$ becomes ready for execution at its release time, $r_{i,j}$. It must complete execution by its absolute deadline, $d_{i,j}$, or it is said to have missed its deadline. Figure 2.1 shows these quantities in the context

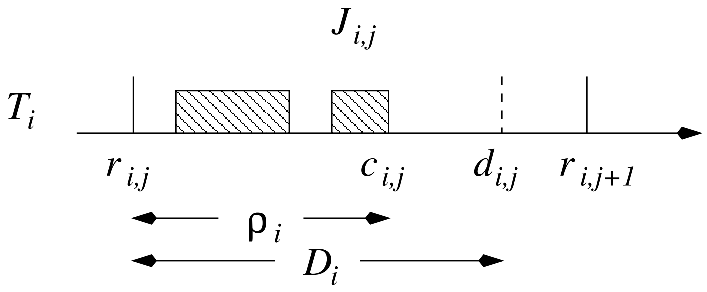


Figure 2.1: Time-line for Task T_i.

of a time-line. The length of time $D_i = d_{i,j} - r_{i,j}$ between the release time and absolute deadline of every job in each task T_i is called the *relative deadline* of the task. The completion time of $J_{i,j}$ is $c_{i,j}$ and the response time is $\rho_{i,j} = c_{i,j} - r_{i,j}$.

The maximum utilization U_i^+ of a task is the ratio of the maximum execution time to the minimum inter-release time (period), i.e.,

$$U_i^+ = \frac{E_i^+}{P_i^-}$$

The maximum utilization of a system of n tasks is

$$U^+ = \sum_{1 \le i \le n} U_i^+$$

In a similar manner, the average utilization of the system $\bar{U}$ is defined as

$$\bar{U} = \sum_{1 \le i \le n} \frac{\bar{E}_i}{\bar{P}_i} = \sum_{1 \le i \le n} \bar{U}_i$$

Finally, the release time of the first job in a task is called the *phase* of the task. We say that tasks are *in-phase* when they have identical phases.

In modern real-time systems, tasks are scheduled in a priority driven manner. At any point in time, the ready job with the highest priority executes. Most systems use a fixed priority assignment according to which all jobs in a task have the same priority. The priority of task T_i is denoted ϕ_i. For convenience and without loss of generality, we assume that priorities are distinct in a fixed priority system and arrange the tasks in order of non-increasing priority $T_i \preceq T_{i+1}$ such that T_i has a higher priority than T_{i+1} for all i. Examples of fixed priority policies are *Rate Monotonic* (RM) [1] or *Deadline Monotonic* (DM) [30]. The priority of a task under RM is inversely proportional to the period of the task. The priority of a task under DM is inversely proportional to the relative deadline of the task. Priorities may also be assigned dynamically in the which case the priority of job $J_{i,j}$ is $\phi_{i,j}$. The most common dynamic priority scheduling policy is *Earliest Deadline First* [1] (EDF) which assigns priorities to jobs in order of their absolute deadlines, i.e. the earlier the deadline the higher the priority.

2.2 Deterministic Schedulability Analysis

A task in a system is said to be *schedulable* if all jobs in the task meet their deadlines. A system of tasks is schedulable if all the tasks in the system are schedulable. Deterministic real-time theory determines the schedulability of a system based on the maximum execution times and minimum inter-release times of its tasks. In order to be schedulable, each task is allocated a portion of the processor bandwidth equal to its maximum utilization. In other words, the system is designed with sufficient capacity to enable it to meet the peak time demand of all tasks simultaneously. The maximum utilization for which a specific scheduling algorithm is guaranteed to schedule all jobs of an arbitrary system without missing a deadline is called the *schedulable utilization* of the scheduling algorithm.

It was shown by Liu and Layland in [1] that a system of n tasks scheduled on a RM basis is schedulable if the maximum utilization of the system satisfies the inequality

$$U^+ \leq n \left(2^{\frac{1}{n}} - 1\right)$$

The expression on the right hand side of the inequality is often called the Liu and Layland bound. In the limit, the Liu and Layland bound approaches $\ln 2 \approx 0.693$. Thus the schedulable utilization of RM is $\ln 2$. We note that the Liu and Layland bound is a sufficient condition. A system may be schedulable according to the RM priority assignment policy even though its maximum utilization exceeds the Liu and Layland bound.

Consider the system of three fixed priority tasks in Table 2.1. The tasks are arranged in order of decreasing RM priority. The Liu and Layland bound is computed for task T_i by considering the set of tasks with priority equal to or higher than T_i. In the example, the maximum utilization of the set of tasks consisting of $\{T_1\}$ is less than the Liu and Layland bound, hence task T_1 is schedulable. Likewise, task T_2 is schedulable because the maximum utilization of the set of tasks

Table 2.1: Parameters of the Tasks.

T_i	ϕ_i	P_i^-	E_i^+	D_i	U_i	U^+	$n(2^{\frac{1}{n}}-1)$
T_1	1	300	100	300	0.333	0.333	1.000
T_2	2	400	100	400	0.250	0.583	0.828
T_3	3	600	200	600	0.333	0.917	0.780

$\{T_1, T_2\}$ is less than the Liu and Layland bound. However, the Liu and Layland bound does not allow us to determine if task T_3 is schedulable when executed with the other two tasks.

One of the limitations of the Liu and Layland bound is that it assumes the worst-case execution time for every job in a task and hence may be overly pessimistic when execution times vary widely. In their paper on the Multiframe Model [31], Mok and Chien observed that the execution times of jobs in many tasks vary according to a fixed repeating pattern. For example, a task which decodes a MPEG video stream executes longer to decode I frames than it does to decode P or B frames. Because the frames in a video stream follow a fixed pattern, e.g., I-B-P-B, the execution times of jobs in the task vary according to the same pattern. For each position in the pattern, the worst-case execution time of jobs in that position is determined. In general, a task T_i has a sequence of worst-case execution times $\{E_{i,1}^+, \ldots, E_{i,N_i}^+\}$ of length N_i. In the example above, the first element of the sequence is the worst-case execution time of all I frames. Mok and Chien derived the following schedulability bound when each of the n tasks in the system has a fixed sequence of worst-case execution times and the first execution time in the sequence is larger than the others

$$U^+ \leq rn\left(\left(\frac{r+1}{r}\right)^{\frac{1}{n}} - 1\right)$$

where $r = \min_{i=1}^{n}(E_{i,1}^+/E_{i,2}^+)$ is the smallest of the ratios of the first two worst-case execution times of the tasks. In systems with a pattern of length one, r is one, and the multiframe bound reduces to the Liu and Layland bound. In spite of the increased precision the technique affords, we still cannot determine the schedulability of a task if the total utilization of the task and higher priority tasks is greater than the multiframe bound. In addition, some systems do not have a fixed pattern of worst-case execution times.

The Time Demand Analysis (TDA) method [9] provides a more accurate and general characterization of the schedulability of arbitrary fixed-priority systems than the Liu and Layland bound. For a system of independent periodic tasks in which the tasks are scheduled preemptively on a fixed-priority basis and every job completes before the next job in the task is released, the worst-case response time of a the task occurs when one of its jobs is released along with a job from every task of equal or higher priority [1]. The release time of such a job is called a *critical instant* of T_i. To determine if all jobs in T_i meet their deadlines, it suffices for us to look at a job in T_i that is released at a critical instant. We call this job $J_{i,1}$. The *time demand function* of T_i, denoted $w_i(t)$, is the maximum processor time demanded by $J_{i,1}$, as well as all the jobs that complete before $J_{i,1}$, as a function of time t since the release of $J_{i,1}$

$$w_i(t) = \sum_{1 \leq k < i} \left\lceil \frac{t}{P_k^-} \right\rceil E_k^+$$

It is a function which increases by the maximum execution time E_k^+ every time a higher priority job $J_{k,l}$ is released. If there is sufficient time before the deadline of $J_{i,1}$ such that $w_i(t) \leq t$ is satisfied, then no job in T_i will miss its deadline.

Figure 2.2 shows the time demand function for each of the tasks in Example 1. There is sufficient time for tasks T_1, T_2 and T_3 to complete by 100, 200 and 600 respectively. Thus the system is schedulable in spite of the fact that the maximum utilization of the system is greater than the Liu and Layland bound. A schedule of the system with the initial job in each task released at a critical instant is shown in Fig. 2.3. Even though the processor is idle from 1100–1200, it is clear that increasing the maximum execution time of any task will result in the potential for $J_{3,1}$ to miss its deadline at 600.

The description of TDA given above works only when all jobs complete by the release of the next job in the task, which is the case for the example. To determine whether all jobs in T_i meet their deadlines when a job in some task of the set $\{T_1, T_2, \ldots, T_i\}$ may be released before the previous job in the same task completes, we must compute the worst-case bounds on the response times of all jobs in T_i executed in a *in-phase level-ϕ_i busy interval* that begins at an instant when a job $J_{i,1}$ in T_i is released at the same time with a job in every higher priority task.[1] A level-ϕ_i busy interval is an interval of time which begins the instant when

[1]This instant is still called a critical instant in the literature even though $J_{i,1}$ may not have the worst-case response time among all jobs in T_i.

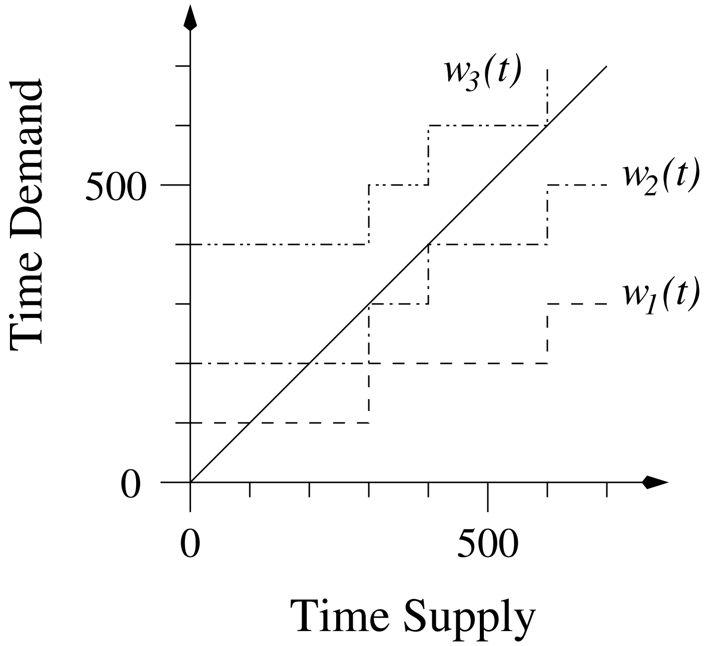


Figure 2.2: Time Demand Analysis of the Example System.

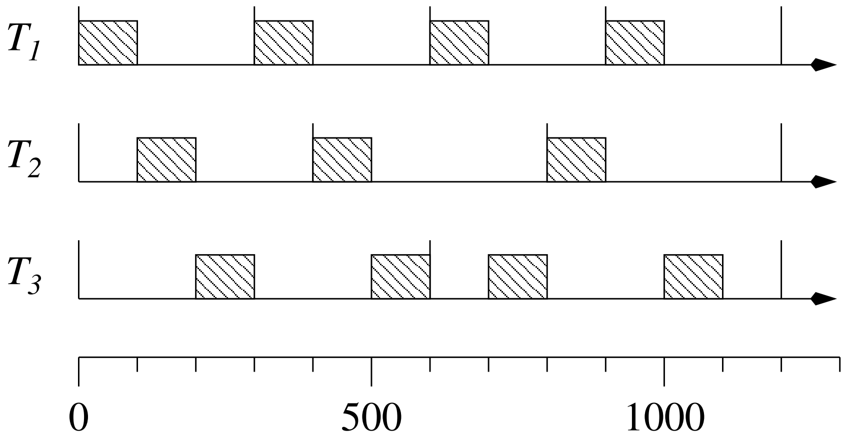


Figure 2.3: Schedule of the Example System.

a job in T_i or a higher priority task is released and immediately prior to the instant no job in those tasks is ready for execution. It ends at the first time instant t at which all jobs in T_i and higher priority tasks released before t have completed.

Analogous to the critical instant analysis in [1], it has been shown in [10] that it suffices for us to consider only an in-phase level-ϕ_i busy interval for the following reasons.

1. If a job in T_i is ever released at the same time as a job in every higher priority task, that instant is the beginning of an in-phase level-ϕ_i busy interval (i.e., the system has no backlog at that instant).

2. The length of an in-phase level-ϕ_i busy interval is longer than a level-ϕ_i busy interval that is not in-phase and hence more jobs in T_i are released in an in-phase level-ϕ_i busy interval.

3. The response time of every job in a level-ϕ_i busy interval that is not in phase is no greater than the response time of the corresponding job in an in-phase level-ϕ_i busy interval.

For these reasons, if all jobs in an in-phase level-ϕ_i busy interval meet their deadlines, the task is schedulable [10]. We call this Generalized Time Demand Analysis (GTDA).

We know from TDA that the system of tasks in Table 2.1 is schedulable. However, suppose that a significantly less expensive processor is available which is half as fast. Just as in the automotive engine management system example in Section 1.1, the profitability of the product would be greatly enhanced if the slower processor could be used. Using the slower processor, the execution time doubles but the periods do not change because they are determined by the environment. Thus the system utilization is doubled, as shown in Table 2.4. The deterministic analysis techniques discussed earlier can only tell us that task T_1 is schedulable and that tasks T_2 and T_3 are not. They cannot tell us how often deadlines will

Figure 2.4: Tasks of Example 1 on Slower Processor.

T_i	ϕ_i	P_i^-	E_i^+	D_i	U_i^+	U^+	$n(2^{\frac{1}{n}} - 1)$
T_1	1	300	200	300	0.67	0.67	1.00
T_2	2	400	200	400	0.50	1.17	0.83
T_3	3	600	400	600	0.67	1.83	0.78

be missed. Although we may be willing to trade occasional missed deadlines for the use of the slower processor, we are unable to do so based on the information obtained from deterministic real-time techniques.

Our work is based on an extension of the periodic task model in which a guaranteed execution time, which may differ from the maximum execution time, is specified for each task in the system. Likewise, the guaranteed inter-release time specified for a task may differ from the minimum inter-release time of task. Unlike the Liu and Layland or Multiframe bounds, the objective of the Stochastic Time Demand Analysis (STDA) described in Chapter 4 is to derive a lower bound on the percentage of jobs in a task that meet their deadlines. While the bound obtained by STDA can be used to determine the schedulability a system, it also allows us to determine if the frequency of missed deadlines is acceptable when the system is not schedulable. STDA is similar to Generalized Time Demand Analysis in that it also considers the jobs of an in-phase level-ϕ_i busy interval. Unlike GTDA, it is not restricted to systems in which the maximum utilization is less than 1.0. It does this by taking a probabilistic rather than a deterministic approach.

2.3 Probabilistic Schedulability Analysis

There are only two other real-time techniques that exploit information about the statistical behavior of periodic tasks to analyze real-time systems: Probabilistic Time Demand Analysis (PTDA) [32] and Statistical Rate Monotonic Scheduling (SRMS) [33].

Like the method proposed in Chapter 4, PTDA attempts to provide a lower bound on the probability that jobs in a task will complete in time. It is a straight forward extension to TDA in which the time demand is computed by convolving the probability density functions of the execution times instead of summing the maximum execution times. PTDA assumes that the relative deadline of all tasks are less than or equal to their periods and computes a lower bound on the probability that jobs in a task complete in time by determining the probability that the time supply equals or exceeds the time demand at the deadline of the first job in the task. The assumption is not valid when the average utilization of the system approaches one and hence Stochastic Time Demand Analysis was developed.

SRMS is an extension to classical Rate Monotonic scheduling. Its primary goal is to schedule tasks with highly variable execution times in such a way that the portion of the processor time allocated to each task is met on the average. Variable execution times are “smoothed” by aggregating the executions of several jobs in a task and allocating an execution time budget for the aggregate (which may be proportional to the original). A job is released only if its task contains sufficient budget to complete in time and if higher priority jobs will not prevent its timely completion. All other jobs are dropped. The analysis given in [33] can only be used to compute the percentage of jobs in each task that will be released for execution (and hence complete in time). Moreover, it is applicable only when the periods of the tasks are related in a harmonic way, i.e., each larger period P_j^- is an integer multiple of every smaller period P_i^-. Recent extensions [34] generalize SRMS to non-harmonic systems. The STDA method presented in Chapter 4 seeks to provide a lower bound on the percentage of jobs which meet their deadlines when all jobs are released, and it is not restricted to the RM scheduling policy.

2.4 Overload Handling

Since processor bandwidth allocations in a critical soft real-time system are often less than the amount necessary to guarantee that the system is schedulable, the scheduler must be able to accommodate overload. The scheduling of overload systems was considered by Tia *et al.* [32] who proposed the Task Transform Method (TTM). The Overrun Server Method (OSM) proposed in Chapter 5 is both a simplification and an extension of the TTM. Like the TTM, the OSM also transforms a job into a mandatory periodic task, whose maximum execution time is the guaranteed execution time in our model, and a request to a server for the execution of the remaining portion. Under fixed priority scheduling, OSM and TTM both execute the remaining portion by a Sporadic Server [20]. Under an EDF scheduler, OSM executes requests by either a Constant Utilization Server [35] or a Total Bandwidth Server [21] rather than a Slack Stealer [11] as is the case with TTM.

The Overload Server Method is also similar to the work by Chung *et al.* [36]; the main difference being that the remaining portions of all jobs execute to completion under OSM instead of being terminated when deadlines cannot be met. Chung also investigated several policies for assigning priorities to remaining portions and compared how well the policies minimized the average error of the system based on a simple non-linear error function. (The error in the result computed by a job is a function of the difference between the result obtained by executing for less than the required time and the result obtained by executing to completion.) They found that assigning priorities to the remaining portions on the basis of the deadlines of the jobs generally resulted in a lower average error. Our results also suggest that deadline-driven scheduling of the remaining portions gives better performance even though we used the average deadlines met and the average response times as our metrics instead of average error.

The Isolation Server Method (ISM), proposed in Chapter 5, is similar to the Proportional Share Resource Allocation algorithm (PSRA) [37]. Both assign a portion of the processor bandwidth to a task. Whereas the PSRA algorithm allocates the assigned portion to jobs in discrete-sized time quanta, the ISM allocates the portion in variable sized chunks. The difference between the portion of processor bandwidth a task receives under the PSRA algorithm and the ideal is bounded by a constant equal to the quantum size. The ISM provides the ideal portion precisely. Both algorithms allow the integration of real and non-real-time processing, are easy to implement and prevent ill-effects of overrunning jobs on jobs in other tasks.

Both the Overrun Server Method and the Isolation Server Method require tasks to be assigned to servers. For the fixed priority case, Katcher *et al.* [38] consider the problem of assigning n fixed priority tasks to m servers, where $m \leq n$ and give an exponential time algorithm for determining the assignment that gives the smallest response time while ensuring that the system remains schedulable, if such an assignment exists. However, the systems under consideration in this thesis are, for the most part, not schedulable according to deterministic real-time scheduling theory. In addition, we relax the assumption that jobs are served first-come-first-served and consider the behavior of Sporadic Servers with a fixed priority queue discipline (DM) and a queue discipline designed to minimize response times. We also consider the assignment of tasks to servers under EDF scheduling. As will be shown in Chapter 5, our results indicate that the behavior of a system with multiple servers is bounded by the assignment of all tasks to a single server and the assignment of each task to its own server.

In [6], Ghazalie and Baker consider the performance of several aperiodic servers in a deadline-driven environment. Their focus is on scheduling aperiodic tasks while our focus is on scheduling overruns using servers. One of the servers they consider is a variation of the Sporadic Server, adapted to a dynamic priority environment, while we use Sporadic Servers for fixed priority scheduling and

Constant Utilization Servers [35], Total Bandwidth Servers [21] or Weighted Fair Queueing Servers [39] (also called Packet-by-Packet Generalized Processor Sharing [40]) for dynamic priority scheduling. They observe that the average response time of an aperiodic task decreases with increases in Sporadic Server server period while we observe that increasing the server period can either increase or decrease the average response time depending on the execution time distributions involved. Some of the difference may be due to fixed versus dynamic priority, but it is also likely that the difference comes from averaging the behavior of many systems instead of observing the behavior of a single system. We also consider three disciplines for prioritizing the server ready queue while it appears that their server executes requests in order of arrival. Finally, we also consider dependencies between the execution times of consecutive jobs.

A complimentary work to ours is the Open Systems Environment (OSE) described in [35]. Similar to the deadline-driven version of the ISM with a server per task, the OSE ensures that the behavior of a task does not interfere with the ability of other tasks to meet their deadlines. The primary motivation of the OSE is to allow real-time applications to be developed and validated independently by assuming that each application runs alone on a slow processor and then are executed together on a fast processor without causing missed deadlines. The primary motivation of the OSM and the ISM is to accommodate overrun. Thus, the OSM and the ISM are complimentary to the OSE.

Ramanathan [41] reduces the load on an overloaded system by selectively discarding jobs in a task according to the (m, k)-firm deadline model of [42]. According to the approach, the stream of jobs from a task is partitioned into sequences of k consecutive jobs. Out of the k consecutive jobs, m jobs in a sequence are declared mandatory and are scheduled at their nominal priority while the remaining jobs are declared optional and given a priority lower than any real-time task. Thus, optional jobs are only scheduled if sufficient time is available and at least m out of k consecutive jobs in each task are guaranteed to meet their deadlines. The error introduced by discarding jobs can be compensated for by modifying the control law computation [41]. The approach assumes that priorities are fixed and that some jobs can be dropped. It also assumes that jobs can be complete out of order since optional jobs may complete after subsequent mandatory jobs due to being given a lower priority. Our work assumes that all jobs must be executed and they must complete in order. We allow either fixed or dynamic priority assignments.

The work by Chu and Nahrstedt on the Dynamic Soft Real-Time scheduling framework (DSRT) [43, 44] is also closely related. Their middleware, which requires conformance to the POSIX 1003.1b standard for real-time support, requires no modifications to the kernel in order to co-schedule hard, soft, and aperiodic real-time workloads (along with time-share workloads) on a set of processors. Under DSRT, the processing capacity of a system is divided into fixed sized real-time, overrun and time-share partitions. The scheduler task which chooses the next job to execute based on an EDF policy is given the highest priority in DSRT. The selected job is assigned a real-time priority below the scheduler priority so that it will run when the scheduler finishes. In this way, DSRT implements EDF scheduling using fixed priority services which conform to the POSIX real-time standard. Each real-time task submits a reservation for processor bandwidth, with DSRT performing admission control to prevent hard real-time jobs from missing their deadlines. To assist the user in deciding upon an appropriate bandwidth reservation for a task, the framework provides a "smart probing" feature which executes a specified number of jobs in the task without real-time guarantees and returns a suggested reservation based upon the processor usage of the jobs. As part of the reservation, the task can specify an adaptation policy which allows the system to modify the reservation as the required bandwidth of the jobs in the task change. If a job overruns, the remainder of its execution will take place in the overrun partition. In this respect, the scheduling of overrunning jobs most resembles our Overrun Server Method with a single server for all tasks scheduled according to the EDF algorithm.

Most of the differences between DSRT and the overload scheduling techniques we present stem from differences in purpose and scope. DSRT is designed as a complete solution for supporting soft real-time workloads on general purpose operating systems and therefore contains mechanisms and policies, such as adaptation, which are useful in that environment. We focus on issues related to overrun scheduling in support of critical soft real-time systems where real-time performance is of primary importance. In spite of the different goals, there are many similarities. Like DSRT, the OSM partitions the processor bandwidth. However, the partitions in OSM are sized dynamically based on the requirements of the admitted tasks rather than being statically determined by an administrator. Furthermore, the processor bandwidth in OSM is not allocated in discrete-sized time quanta as it is in DSRT. Instead of treating all overrunning jobs equally as the OSM does, DSRT classifies overrunning jobs into one of two categories based on the amount and frequency of overruns and schedules the jobs accordingly. Our approach does not preclude a multi-level overrun classification policy from being implemented but the policy implemented in DSRT is undesirable for scheduling critical real-time systems because of its emphasis on fairness at the expense of meeting deadlines. We note that DSRT can approximate the Isolation Server Method by letting the guaranteed execution times of tasks (execution time reservations) be zero and setting the burst tolerances appropriately. However, DSRT uses round robin scheduling within an overrun class to ensure fairness whereas the OSM uses priority-based scheduling to achieve better real-time performance. The adaptive statistical characterization in Appendix C is similar to the "smart probing" feature of DSRT with the primary difference being that we are after distributions while [43, 44] seeks some simple descriptive statistics in order to form reservations. Finally, as mentioned earlier, DSRT provides support for adapting the resource reservation of a task to accommodate changing processor demands while OSM does not. Adaptation is not required in most critical soft real-time systems.

The work described in this thesis differs from recent approaches (such as [45]) by relaxing hard real-time constraints in a controlled manner rather than attempting to add support for real-time tasks to time-shared operating systems without compromising fairness. The latter is unable to provide any form of hard real-time guarantee under overload by virtue of an insistence on being fair. In contrast, the baseline and OSM algorithms discussed in this thesis are able to make real-time guarantees and thereby may sacrifice the ability to be fair to all tasks. However, fairness within non-real-time tasks can achieved without sacrificing the ability to make guarantees by executing non-real-time tasks within a server which implements a traditional time-share scheduler. Thus the baseline and OSM algorithms described in this thesis are able to guarantee the deadlines of hard real-time tasks (and critical soft real-time tasks whose jobs do not exceed the guaranteed execution time) while scheduling non-real-time tasks fairly within the portion of the processing time allocated for them.

2.5 Queueing Theory

The statistical analysis of real-time systems described in Chapter 4 is similar to the analysis of queueing systems in many ways. Indeed, real-time systems can be described by a queuing theoretic model rather than a periodic task model. However, a very simple real-time system is still a complex queueing system.

A key assumption underlying most analytical results in queueing is the "memoryless property" of the arrival time and/or execution time distributions. For example, M/M/1 and M/G/1 queues have Poisson arrivals; M/M/1 and G/M/1 queues have exponential execution times [46, 47]. The memoryless property makes the analysis of otherwise complex queueing systems tractable. On the other hand, jobs of a real-time task arrive more or less periodically, implying an arbitrary inter-arrival time distribution. Likewise, the execution times of jobs in a task are likely from distributions for which there are no elegant mathematical description. To further complicate attempts to achieve an analytical solution, real-time systems are scheduled preemptively according to priority. Although queueing theoretic analysis techniques exist for priority scheduled systems (see [47] Chapter 3), combining priority scheduling with general inter-arrival time and execution time distributions that do not have the memoryless property makes analytical solutions of average performance difficult.

Recently, Lehoczky has developed a queuing theoretic technique called Real-Time Queueing Theory which is better suited to real-time systems [48–50]. Given a scheduling algorithm and a distribution of the deadlines of jobs, analytical expressions for the lead-time distribution of jobs in each task are derived as a function of the average number of jobs in the queue for the FIFO, EDF and processor sharing scheduling algorithms. (The lead-time of a job is the time remaining until its deadline.) The formulation for the lead-time distributions of tasks is based upon the assumption of a high system load in order to approximate the behavior of the system by a diffusion process.

While the technique shows promise, results published to date have been limited to the above mentioned scheduling algorithms and tasks with either Poisson arrivals or exponential execution times. Besides the lack of expressions for fixed-priority schedulers with arbitrary execution time and inter-arrival time distributions, Real-Time Queueing Theory yields the average number of deadlines met rather than a lower bound on the percentage of deadlines met, the latter being more important for critical soft real-time systems. In addition, the diffusion approximation upon which the approach is based requires the average number of jobs in the queue to be large in order to be accurate. (The results for a 95% average system utilization and uniformly distributed execution times reported in [48] were obtained with an average queue length of 50 jobs. In contrast, the system of three tasks scheduled by a Total Bandwidth Server discussed in Figure 5.8 of Chapter 5 has a maximum queue length of two.) Because of these difficulties, the approach taken in this thesis is to extend real-time system results through a statistical treatment rather than extending queueing theory to account for real-time constraints.

It should be noted that any statistical approach, based on either real-time or queueing theory, must deal with the practical problem of summing random variables. The primary difficulty is in computing the resulting probability density function from the density functions of the summands. Analytical results for sums of random variables exist only for certain special cases. In the general case, the probability density function of the sum must be computed by convolution. Convolution, however, has remained computationally expensive in spite of efforts to reduce its cost. (See [51], for example). The fastest known method for performing convolution is to compute the Fast Fourier Transforms of the density functions, multiply, and and compute the inverse transform. This is still an expensive operation, however. It is natural to wonder if the effort required to compute the probability distribution of a sum of random variables can be reduced by approximations derived from well known mathematical or statistical results (see Appendix D). Sadly, the bounds obtained by this approach are so loose as to completely discourage their use in analyzing real-time systems. Because of this, we compute probability distributions via convolution in this work.

Chapter 3

Extended Periodic Task Model

Deterministic real-time theory determines the schedulability of a system based on the maximum execution times and minimum inter-release times of tasks in the system. In order to ensure that the system is schedulable, the processor bandwidth set aside for each task is equal to its maximum utilization. Because the execution times or inter-release times of jobs in many real-time systems vary widely, designing a critical soft real-time system using deterministic real-time theory often yields a system whose average utilization is unacceptably low.

In systems where the execution time varies, we require that a *guaranteed execution time*, E_i^*, be specified for each task instead of a maximum execution time as in the original periodic task model. The guaranteed execution time of a task is zero for non-real-time tasks, equal to the maximum execution time of any job of the task for hard real-time tasks, and somewhere in between for soft real-time tasks. To account for inter-release time variations, we require that a *guaranteed inter-release time*, $P_i^* > 0$, be specified for each task instead of a minimum inter-release time. The guaranteed inter-release time of a task is infinite for non-real-time tasks, equal to the minimum inter-release time of any job in the task for hard real-time tasks, and somewhere in between for soft real-time tasks. (We define the *guaranteed utilization* in the expected way, $U_i^* = E_i^*/P_i^*$.) We require that systems be schedulable according to deterministic real-time theory on the basis of the guaranteed execution and inter-release times of tasks. Modifying the periodic task model in this manner allows systems containing hard, soft and non-real-time tasks to be described and treated in a simple, unified manner.

Allowing the guaranteed execution time of a task to be less than its maximum execution time increases the potential for jobs to miss their deadlines. A job is said to *overrun* when it executes for more than its guaranteed execution time.

Depending on the amount of time available, a system may be able to schedule the remaining portion of an overrunning job so that it completes by its deadline. Likewise, allowing the guaranteed inter-release time of a task to be greater than its minimum inter-release time increases the potential for jobs to miss their deadlines. We say that a job has a *jittered release* when the time between its release and the release of its predecessor differs from the guaranteed inter-release time. A system may be able to schedule a job with jittered release so that no deadlines are missed, depending upon the load on the system.

We say that a system is *overloaded* when it is not schedulable according to deterministic real-time theory on the basis of maximum execution times and minimum inter-release times and hence some jobs may miss their deadlines. Jobs in a system may overrun or have jittered releases without the system being overloaded. However, an overloaded system implies that a job will overrun or that some release was jittered. In Chapter 5, we consider ways to schedule systems in which jobs overrun so as to guarantee that jobs which do not overrun will meet their deadlines. For jobs with execution times in excess of their guaranteed execution times, the objective is to minimize the response times of the jobs. In Chapter 6, we consider the scheduling of systems in which the release times of jobs are jittered, both with and without overrun. For jobs with inter-release times less than their guaranteed inter-release times, the objective is also to maximize the number of deadlines met and minimize response time.

We note that specifying an execution time for a task less than the maximum is not new. Under the Processor Capacity Reserve model of Mercer *et al.* [52], a task is guaranteed to execute for at least its guaranteed execution time each period. However, we appear to be the first to use the specification of a guaranteed execution time equal to or less than the maximum to characterize hard, soft and non-real-time tasks in a uniform manner. In contrast, we know of no study in which the guaranteed inter-release time of a task is purposefully specified as greater than the minimum inter-release time.

Chapter 4

Stochastic Time Demand Analysis

This chapter presents the *Stochastic Time Demand Analysis* technique (STDA) for computing a lower bound on the percentage of deadlines that a fixed priority periodic task meets in the presence of execution time variations. We compute a lower bound on the probability that deadlines are met by jobs in a simple system and compare the bounds with the percentage of deadlines met obtained by simulation. We then extend the technique to account for blocking caused by non-preemptability. We also apply STDA to computing the probability of meeting end-to-end deadlines in distributed systems.

4.1 Computing Lower Bounds

We now focus on a task T_i in the system. Let $J_{i,j}$ be the jth job in T_i released in a level-ϕ_i busy interval. To simplify the discussion and without loss of generality, we take as the time origin the beginning of the busy interval. The response time $\rho_{i,j}$ of job $J_{i,j}$ is a function of the execution times of all jobs which can execute in the interval $(r_{i,j}, c_{i,j}]$. Since the execution times of jobs are random variables, the response times of jobs are also random variables. Our analysis assumes that the execution time E_i of a job in T_i is statistically independent of other jobs in T_i and of jobs in other tasks. We further assume that the variations in inter-release times are negligible and use the minimum inter-release time in our analysis. Because a job may not complete by the release of a subsequent job in the same task, we must consider all jobs in a level-ϕ_i busy interval. The length of a level-ϕ_i busy interval is also a random variable. Determining when busy intervals end is key to STDA. First we show how to compute the response time distribution of jobs in task T_i.

4.1.1 Computing Response Time Distributions

Let $w_{i,j}(t)$ denote the time demand of all jobs that execute in the interval $(r_{i,j}, t]$. Job $J_{i,j}$ completes when there is sufficient time to meet the demand $w_{i,j}(t) = t$. Let $W_{i,j}(t) = \mathcal{P}[w_{i,j}(t) \leq t]$ denote the probability that the time demand up to t is met by t, given that the busy interval has not ended. $W_{i,j}(t)$ is the probability that the response time of $J_{i,j}$ is less than or equal to t. The probability that $J_{i,j}$ meets its deadline is therefore at least $W_{i,j}(d_{i,j})$.

We now turn our attention to computing $W_{i,j}(t)$. The response time distribution $W_{i,j}(t)$ is computed by conditioning on whether or not a backlog of work from equal or higher priority tasks exists when $J_{i,j}$ is released. If no backlog exists, a level-ϕ_i busy interval starts at the release of $J_{i,j}$, which we relabel $J_{i,1}$, and

$$W_{i,1}(t) \quad = \quad \mathcal{P}[w_{i,1}(t) \leq t] \tag{4.1}$$

If backlog exists, the response time distributions for the subsequent jobs of T_i in the busy interval are computed in order of their release by

$$W_{i,j}(t) \quad = \quad \mathcal{P}[w_{i,j}(t) \leq t \mid w_{i,j-1}(r_{i,j}) > r_{i,j}] \tag{4.2}$$

For the highest priority task, the response time distribution of the first job in a busy interval is the same as its execution time distribution. The response time distribution of the subsequent job in the busy interval is computed by convolving the execution time distribution of the task with the distribution of the backlog obtained by conditioning. This process continues until the end of the busy interval.

We now compute $W_{i,j}(t)$ for $j > 1$. Clearly jobs with a priority higher than ϕ_i can execute in the interval $(r_{i,j}, c_{i,j}]$. Jobs among $J_{i,1}, J_{i,2}, \ldots, J_{i,j-1}$ that complete after $r_{i,j}$, as well as $J_{i,j}$, also execute in this interval. The effect of jobs which are eligible for execution at $r_{i,j}$ is already taken into account in the conditioning process. To compute $W_{i,j}$(t), we must still take into account the time demand of jobs of higher priority tasks released in the interval $(r_{i,j}, c_{i,j}]$. This is done by dividing $(r_{i,j}, c_{i,j}]$ into sub-intervals separated by releases of higher priority jobs and conditioning on whether a backlog of work exists at the start of each sub-interval. For example, suppose that only one higher priority job $J_{k,l}$ is released in the interval $(r_{i,j}, c_{i,j}]$. Its release time $r_{k,l}$ divides the interval into two sub-intervals, $(r_{i,j}, r_{k,l}]$ and $(r_{k,l}, c_{i,j}]$. The probability that $J_{i,j}$ completes by time

t in the first sub-interval $(r_{k,l}, r_{k,l}]$ is

$$W_{i,j}(t) = \mathcal{P}[w_{i,j}(t) \leq t \mid w_{i,j-1}(r_{i,j}) > r_{i,j}] \tag{4.3}$$

and completes by time t in the second sub-interval $(r_{k,l}, r_{i,j+1})$ is

$$\begin{aligned} W_{i,j}(t) \quad = \quad & \mathcal{P}[w_{i,j}(t) \leq t \mid w_{i,j-1}(r_{i,j}) > r_{i,j}, w_{i,j}(r_{k,l}) > r_{k,l}] \\ & \mathcal{P}[w_{i,j}(r_{k,l}) > r_{k,l}] \end{aligned} \tag{4.4}$$

The probability that a job completes by its deadline is determined by systematically computing $W_{i,j}(t)$ for t in successive sub-intervals until the sub-interval containing $d_{i,j}$ has been considered.

Equations 4.1 and 4.2 allow the response time distributions of jobs in a level-ϕ_i busy interval to be computed for any combination of initial release times. In order to compute a lower bound on the probability that jobs complete by their deadlines, the worst-case combination of release times needs to be identified. As discussed previously, an upper bound on the response time of jobs from T_i is obtained by computing the response times of jobs executed in an in-phase level-ϕ_i busy interval according to deterministic TDA. Sadly, we note that it is not longer sufficient for us to consider an in-phase busy interval in general. The proof that no backlog exists at the instant when a job is released simultaneously with the release of a job of every higher priority task requires that the maximum total utilization of the system is no greater than one, as assumed by deterministic TDA. STDA requires only that the average utilization of the system is less than one hence some systems may not meet the condition. It is not clear what relationship between the release times of the first jobs in a level-ϕ_i busy interval causes some job in T_i to have the maximum possible response time and hence the smallest probability of completing in time. For now, we assume that the first jobs in all tasks are released in-phase and discuss the rationale for this assumption later.

4.1.2 Determining End of Busy Interval

We now turn our attention to the matter of determining when a busy interval ends. We note that since there is a single task per priority level, a level-ϕ_i busy interval ends if some job $J_{i,j}$ in T_i completes before the next job $J_{i,j+1}$ is released. Thus we know that the busy interval has surely ended if, for some j,

$\mathcal{P}[w_{i,j}(r_{i,j+1}) \leq r_{i,j+1}] = 1.0$. (When multiple tasks have the same priority, jobs from the same priority level must have their response time distributions computed in order of increasing release times. The busy interval will have ended if all jobs with equal or higher priority released before time t have completed by t with probability 1.0.)

For systems with a maximum utilization greater than 1.0, the probability that $J_{i,j}$ completes before $J_{i,j+1}$ is released may be strictly less than 1.0. In other words, the probability that a level-ϕ_i busy interval ends is less than 100%. Even though a busy interval may not end, the lower bound on the minimum probability that jobs from T_i will meet their deadlines as computed according to STDA is a correct lower bound. To see this, let $\{J_{i,1}, J_{i,2}, \ldots J_{i,n}\}$ be the sequence of jobs of T_i in the level-ϕ_i busy interval and let $\{\mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \ldots \mathcal{P}_{i,n}\}$ be the sequence of probabilities that the corresponding jobs of T_i in the busy interval complete by their deadlines, as computed by the process in Section 4.1.1. The monotonically non-increasing function $\mathcal{L}(n) = \min\{\mathcal{P}_{i,1}, \mathcal{P}_{i,2}, \ldots, \mathcal{P}_{i,n}\}$ represents the development of the lower bound on the probability that jobs in $\{J_{i,1}, J_{i,2}, \ldots, J_{i,n}\}$ complete by their deadlines, as computed by STDA. When the busy interval never ends, $\mathcal{L}(\infty)$ is at least zero since the probability that a job completes by its deadline lies in the range $[0, 1]$. Clearly, a lower bound of zero is a correct lower bound on the minimum probability that jobs in T_i meet their deadlines. The tightness of the lower bound computed by STDA depends on the maximum utilization of the system. For maximum utilizations less than 1.0, the bound is quite good. As the maximum utilization becomes increasingly greater than 1.0, the bound becomes increasingly conservative.

As a practical matter, the process of computing $\mathcal{L}(n)$ as $n \to \infty$ can be terminated when the change between two successive values becomes acceptably small. We have found that the computation converges quite rapidly, with the rate of convergence depending on the maximum utilization of the system.

4.1.3 Computing Bounds for a Simple System

As an example, we now use STDA to bound the percentage of deadlines met in a system of two tasks shown in Table 4.1. The execution time of each task is uniformly distributed with parameters chosen to accentuate the potential for missed deadlines. The worst-case utilization of the system is 1.41 and the mean utilization of the system is 0.71. Consequently, we would expect that some jobs

Table 4.1: Parameters of the Tasks.

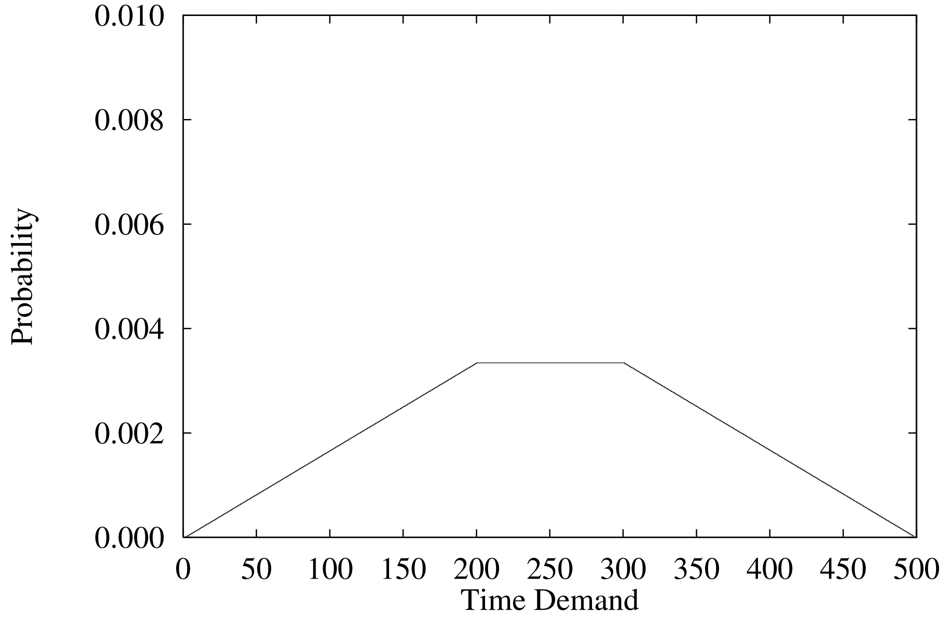
T_i	P_i^-	D_i	E_i^-	$\bar{E}_i$	E_i^+	U_i^-	$\bar{U}_i$	U_i^+
T_1	300	300	1	100	199	0.0033	0.333	0.663
T_2	400	400	1	150	299	0.0025	0.375	0.748
Total						0.0058	0.708	1.411

will miss their deadlines. To determine the probability of jobs in each of the tasks missing their deadlines, we apply the procedure outlined above. Because its maximum utilization is less than 1.0, we know that T_1 will not miss any deadlines. Therefore we begin the analysis with T_2.

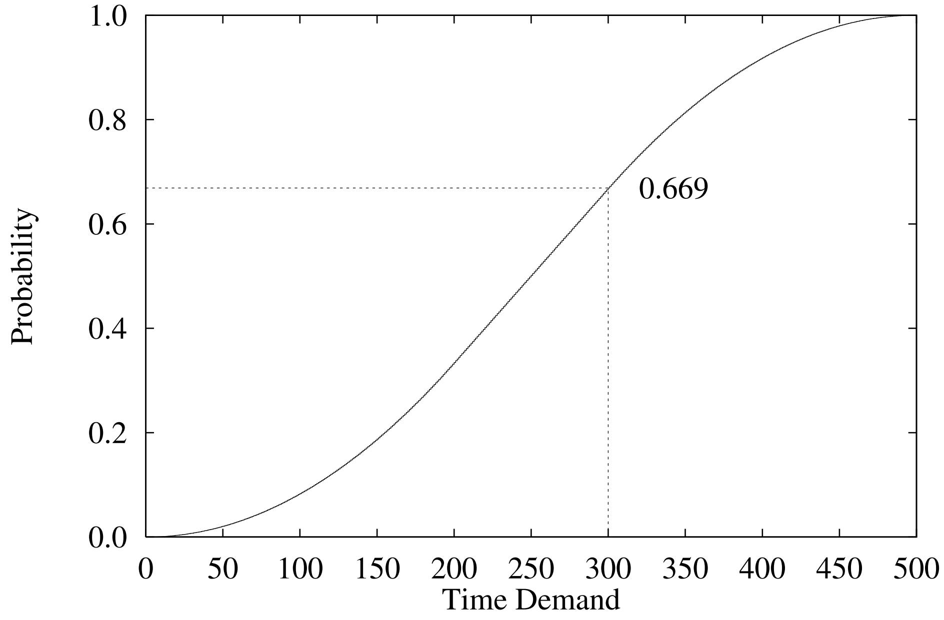
It is apparent that the maximum time demand of T_2 in the interval $(0, 400]$ exceeds the time supply because the sum of the maximum utilizations of the two tasks exceeds one. Because $J_{2,1}$ may not have complete by the time $J_{2,2}$ is released, the response time of $J_{2,2}$ may be greater than that of $J_{2,1}$. At the very least we need to compute the response time distributions for $J_{2,1}$ and $J_{2,2}$. To compute the probability that $J_{2,1}$ completes by its deadline, the interval $(0, 400]$ is divided into sub-intervals $(0, 300]$ and $(300, 400]$ by the release time of $J_{1,2}$ at 300. In the first interval, the time demand includes only the execution times of $J_{1,1}$ and $J_{2,1}$. The time demand of the second interval includes the execution time of $J_{1,2}$, as well as the work remaining from the first interval. The probability that the time demand is no greater than a particular value is conditioned on whether or not $J_{2,1}$ completes before $J_{1,2}$ is released. We first consider the interval $(0, 300]$. The probability that $J_{2,1}$ will finish by 300 is $\mathcal{P}[w_{2,1}(300) \leq 300]$, where $w_{2,1}(t)$ for $0 \leq t \leq 300$ is the sum $E_1 + E_2$ and has the density function and distribution shown in Fig. 4.1. The probability that $J_{2,1}$ completes by 300 is 0.669.

We now compute $\mathcal{P}[w_{2,1}(400) \leq 400 \mid w_{2,1}(300) > 300]$ for t in the interval $(300, 400]$. Because $J_{2,1}$ may not have completed by time 300, there are between 0 and 198 time units of work remaining when $J_{1,2}$ is released. The density function for the backlog is the density function of Fig. 4.1(a) in the range 300–498, normalized so that the cumulative probability is 1.0 at time 498 as is implied by statistical conditioning. The random variable for the backlog is then added to the random variable for the execution time of $J_{1,2}$. The resulting density and distribution are given in Fig. 4.2. The probability that $J_{2,1}$ completes by 400, given that it did not complete by 300, is 0.209 as shown in Fig. 4.2(b).

Combining the results of analyzing the two sub-intervals gives us the distri-

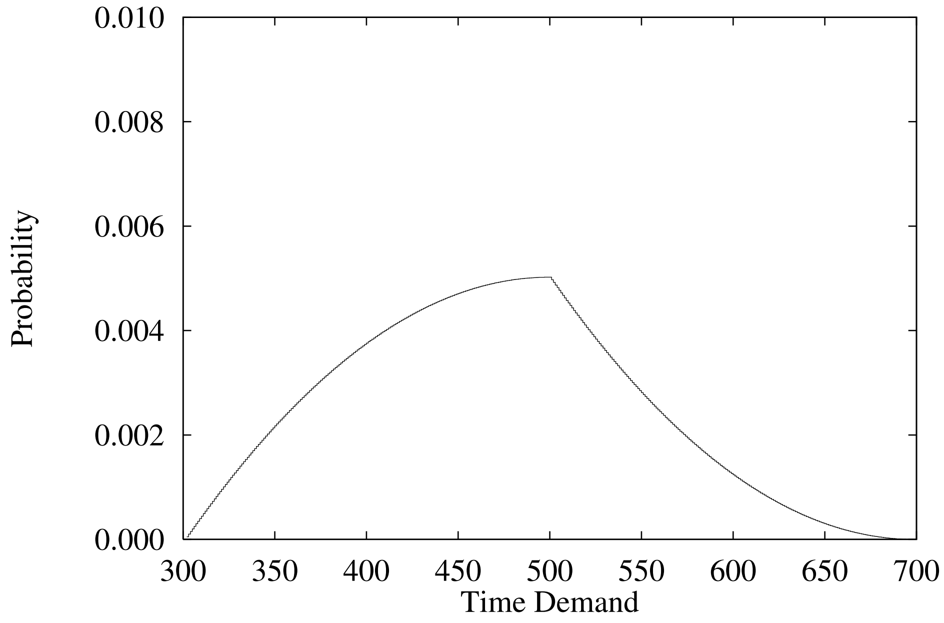


(a) Density

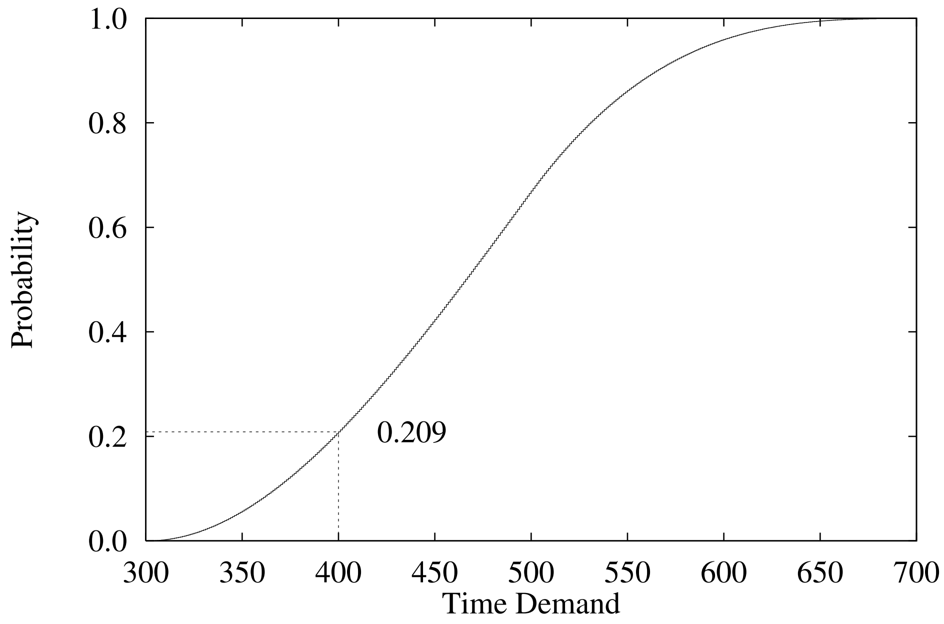


(b) Distribution

Figure 4.1: Time demand of $J_{2,1}$ over interval $(0, 300]$.

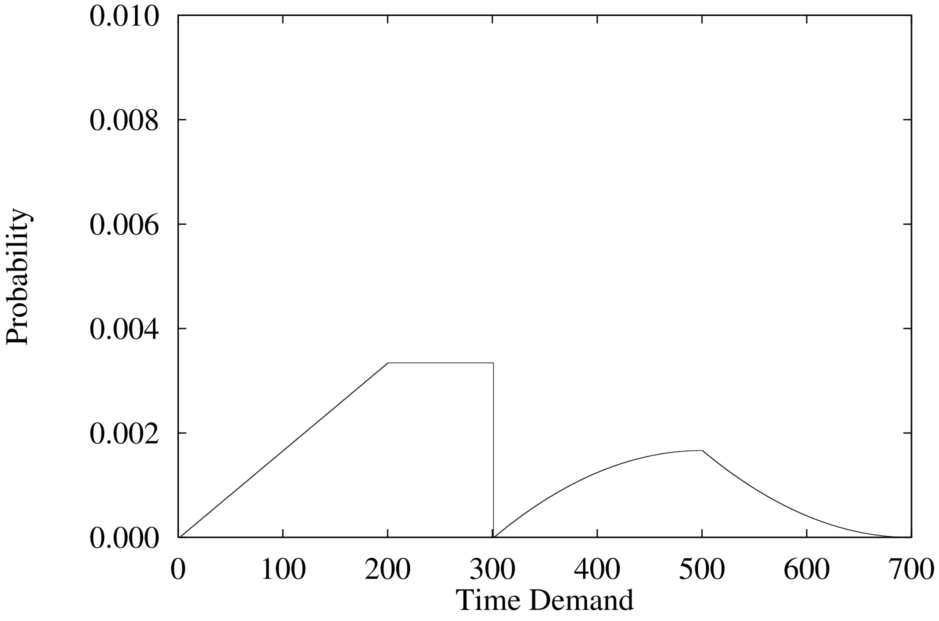


(a) Density

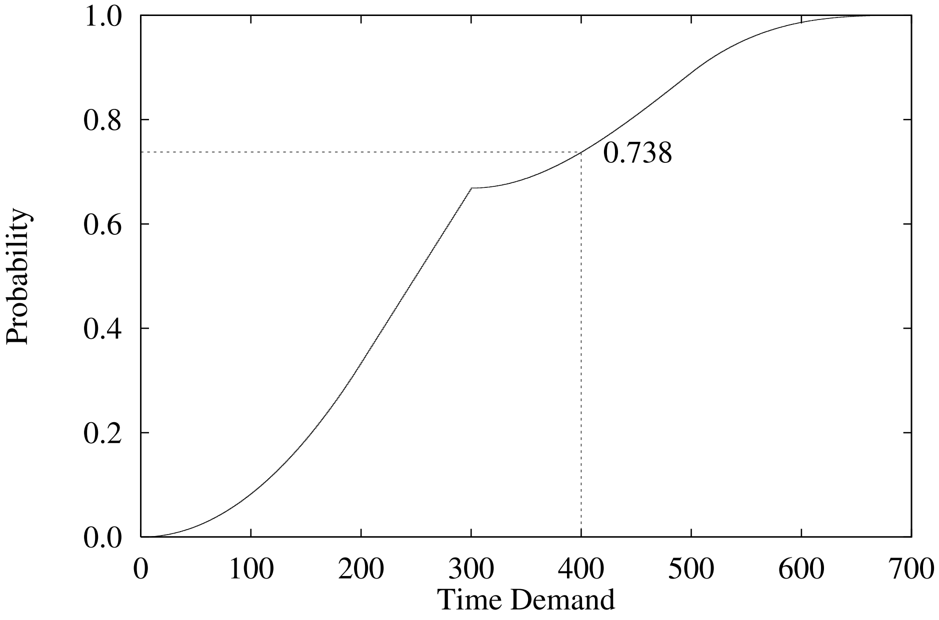


(b) Distribution

Figure 4.2: Time demand of $J_{2,1}$ over interval $(300, 400]$.



(a) Density



(b) Distribution

Figure 4.3: Time demand of $J_{2,1}$ over interval $(0, 400]$.

bution of the response time of $J_{2,1}$ and thus the probability that $J_{2,1}$ completes by 400 and meets its deadline

$$\mathcal{P}[w_{2,1}(400) \leq 400] = (0.669) + (0.209)(0.331) = 0.738 \tag{4.5}$$

The density and distribution functions of the response time of $J_{2,1}$ over the entire interval $(0, 400]$ are given in Fig. 4.3.

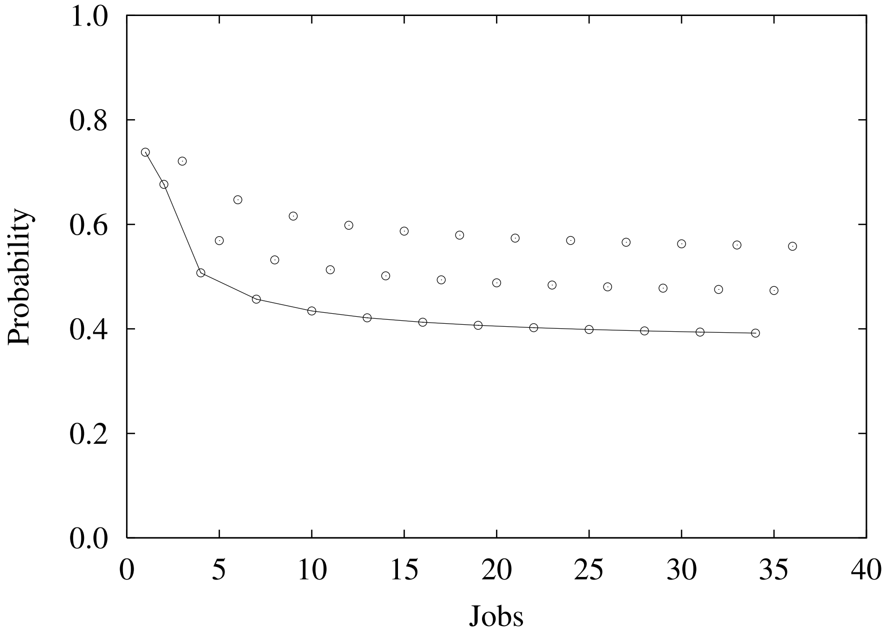
We note that the probability that $J_{2,1}$ will not complete before $r_{2,2}$ is 0.262 so it is also necessary to compute the probability that $J_{2,2}$ completes by its deadline. The analysis proceeds following the same pattern until the busy interval ends or the lower bound converges to the final value. Figure 4.4(a) shows the convergence history for this heavily loaded system. The data points indicate the probability of timely completion of jobs in T_2 as computed by STDA while the line indicates the minimum probability of timely completion. For this system, the lower bound is 73.8% after the first job, 43.4% after 10 jobs, 40.2% after 22 jobs, and 39.2% after 34 jobs. Based on these results, we let the lower bound on the percentage of deadline met be 39.2%.

Figure 4.4(b) shows the rate of convergence for the same system of two tasks (with periods of 300 and 400 and mean execution times of 100 and 200, respectively) but with a minimum execution time of 25 rather than 1. The maximum utilization is 1.27. While the lower bound starts at 76.4% with the first job, it quickly decreases to 53.2% after 10 jobs and 51.1% after 19 jobs. As can be seen by comparing Figure 4.4(a) to Figure 4.4(b), decreasing the maximum utilization causes the probability of meeting deadlines to increase. With a minimum execution time of 70 rather than 1 the maximum utilization is 1.01 and the busy interval ends after the third job.

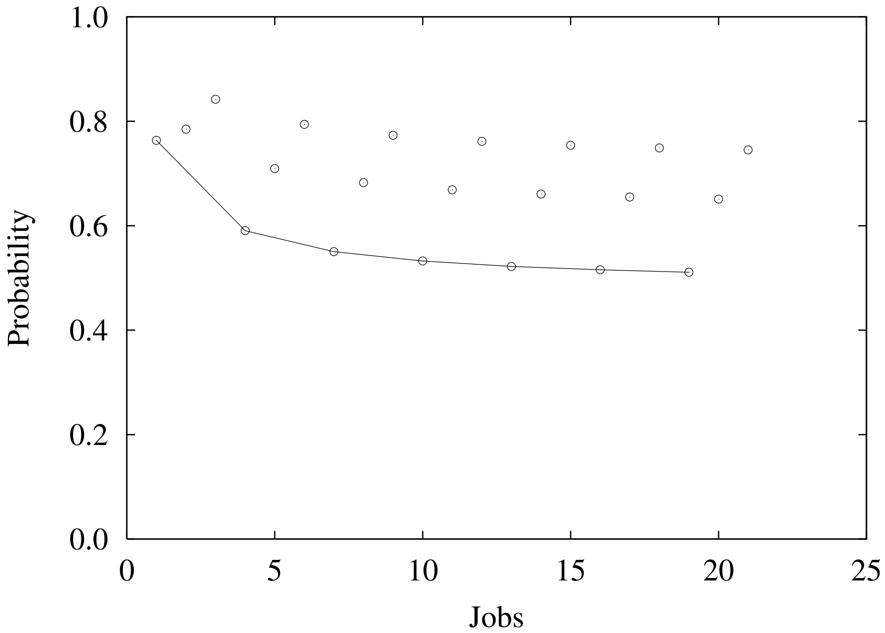
4.1.4 Determining Worst Case Phase

We now return to the choice of initial phases for tasks. When the maximum utilization is no greater than one, the worst-case response time occurs when the first job in each task in a busy interval is released in phase [10]. The result rests upon the following lemma which establishes that a new busy interval starts if jobs are ever released in-phase. (See [10] for proof that an in-phase release results in the worst-case execution time.)

Lemma 4.1.1 *There is no backlog at the start of a in-phase busy interval if the*



(a) 141% Utilization



(b) 127% Utilization

Figure 4.4: Rate of convergence for system of two tasks.

maximum utilization is no greater than 1.0.

Proof: We prove the lemma by contradiction. Suppose a job in every task is released at time t which is in the middle of a busy interval. In other words, that backlog exists at t. This implies

$$\sum_{i=1}^{n} \left\lfloor \frac{t}{P_i^-} \right\rfloor E_i^+ > t$$

Since $\lfloor x \rfloor \leq x$

$$\sum_{i=1}^{n} \frac{t}{P_i^-} E_i^+ > t$$
$$\sum_{i=1}^{n} \frac{E_i^+}{P_i^-} > 1$$

and hence

$$U^+ > 1$$

which contradicts the assumption that the maximum utilization is less than 1.0. □

Because no backlog can exist at the time jobs are released in phase when $U^+ \leq 1$, we are assured that the busy interval will end before the jobs are again released in phase. We now prove that the lower bound on deadlines met computed by STDA is one if the system is schedulable.

Lemma 4.1.2 *If a system is schedulable, the lower bound on deadlines met computed by STDA is 100%.*

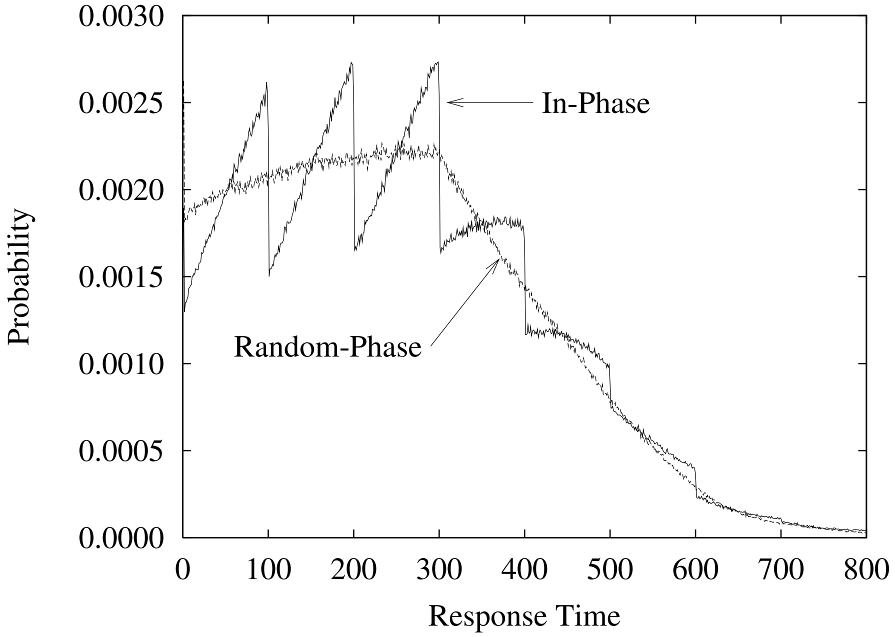
Proof: The maximum value of the response time density function computed for a job using STDA is identical to the sum of the maximum execution times computed as the worst-case response time of the job using GTDA. Hence, if the job is determined to be schedulable using GTDA, it will surely be determined schedulable using STDA. Thus, the lower bound on deadlines met computed by STDA is 100%. □

This result makes STDA particularly useful for analyzing systems in which the maximum utilization of the system exceeds the Liu and Layland bound but is less than unity. It is recommended that STDA be used instead of TDA (or GTDA) since STDA not only determines schedulability, but also computes a lower bound on the deadlines met if the system is not schedulable.

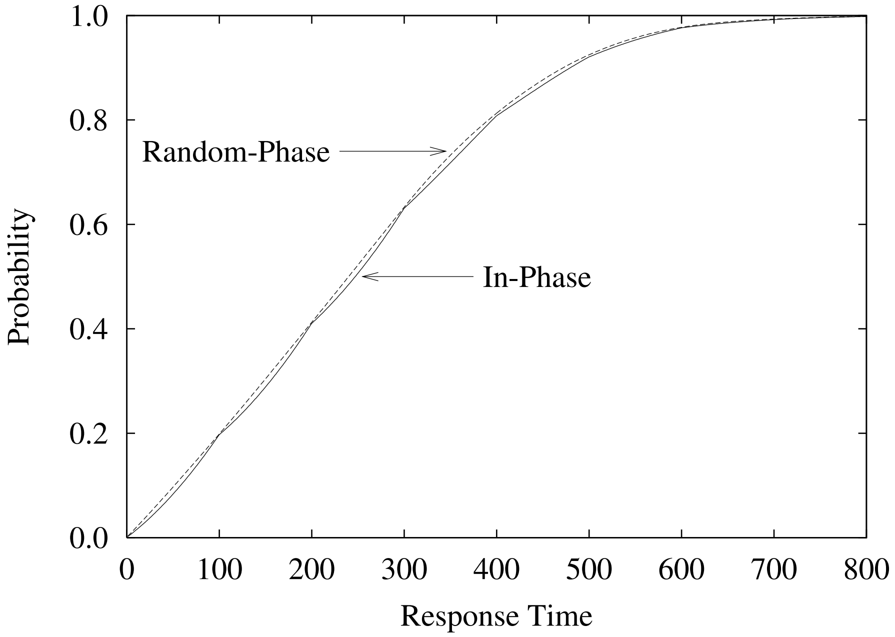
For $U^+ > 1$, we do not know what phasing causes jobs to have their worst-case response times. Some combinations of release times of the first jobs may lead to a larger maximum response time of for a task. We hypothesize that these combinations occur infrequently enough so that the lower bound computed from response time distributions of jobs in an in-phase busy interval is sufficiently accurate. To test this hypothesis, we performed a series of simulation experiments on a number of systems. For each system, we determine the behavior of the system when each task T_i has a randomly distributed phase in the range $(-P_i^-, P_i^-)$ and when all tasks have equal phases, i.e., are released at time 0. (We call a unique combination of phases and actual execution times of the tasks a *run*.) For each run, a histogram of the response times of a large number of jobs in each task is computed. The histograms of all the runs are averaged, bin by bin, to obtain a histogram representing the average behavior of the tasks of the system. The histograms for in-phase and random-phase releases are then compared.

For the tasks in Example 4.1, we performed 100 runs for both in-phase and random-phase releases, each run containing the release of at least 8,000 jobs in each task. The width of the 95$\%$ confidence interval on the profile of the histogram was $\pm 5\%$ of the mean value or less except in the tail of the density function where the probability was small to begin with. Figure 4.5 shows the histograms for task T_2 from our example.

As Fig. 4.5(b) shows, the response time distribution for in-phase releases bounds the distribution for random-phase releases from below. The response time density function, Figure 4.5(a), exhibits a saw-tooth behavior for in-phase releases. The behavior is caused by the fixed relationship between the release times of T_1 and T_2. This relationship causes the completion of jobs in T_2 to be delay by jobs in T_1 in a periodic manner. The linearly rising shape of each tooth is due to the uniform distribution of the execution time of T_1 while the general shape of the curve results from combined effect of the execution time distributions of both T_1 and T_2. Figure 4.6 compares the histograms for tasks with the same parameters as our example but with exponential distribution times. Once again, the distribution obtained when the initial jobs were released in-phase bounds from below the distribution obtained when the first jobs were released with random-phase. Also, the in-phase release curve exhibits a similar saw-tooth shape. However, each

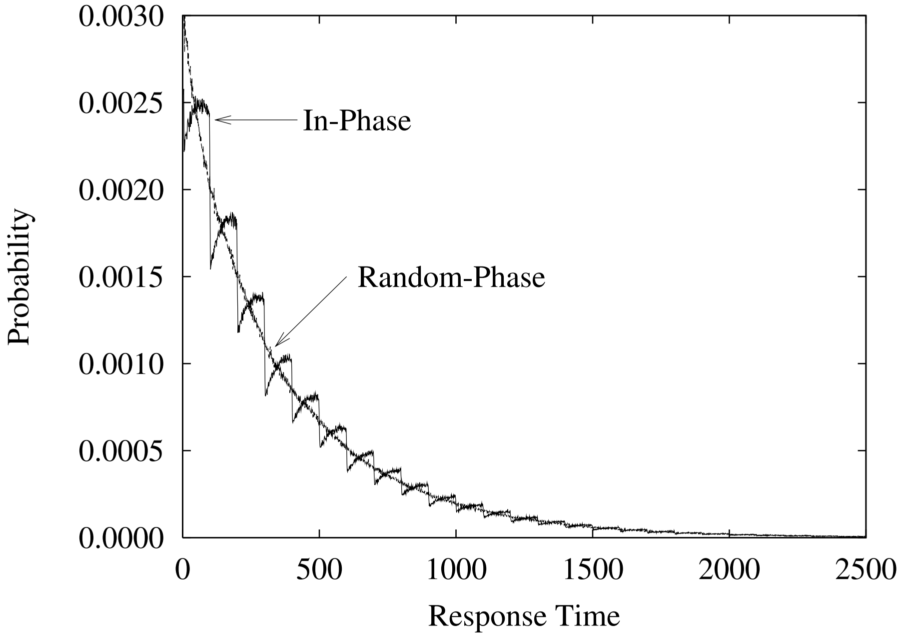


(a) Density

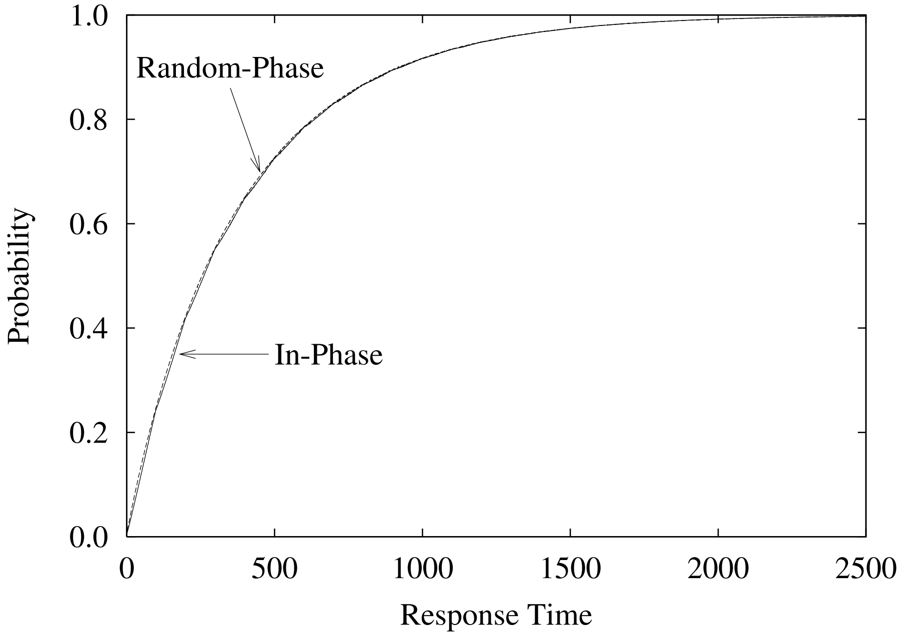


(b) Distribution

Figure 4.5: Response time distributions of T_2: Uniform, 71% Utilization.



(a) Density



(b) Distribution

Figure 4.6: Response time distributions of T_2: Exponential, 71% Utilization.

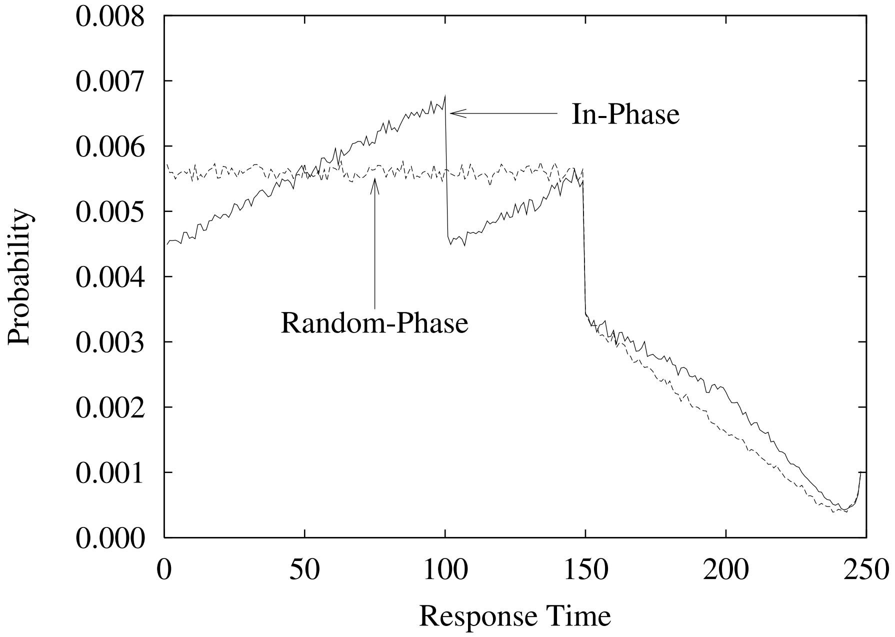
Table 4.2: Parameters of systems with different average utilizations.

System	Minimum Jobs	T_i	P_i^-	$\bar{E}_i$	$\bar{U}_i$	$\bar{U}$	U^+
1	1,000	T_1	300	50	0.167	0.354	0.703
		T_2	400	75	0.188		
2	8,000	T_1	300	100	0.333	0.708	1.411
		T_2	400	150	0.375		
3	32,000	T_1	300	134	0.447	0.949	1.893
		T_2	400	201	0.503		

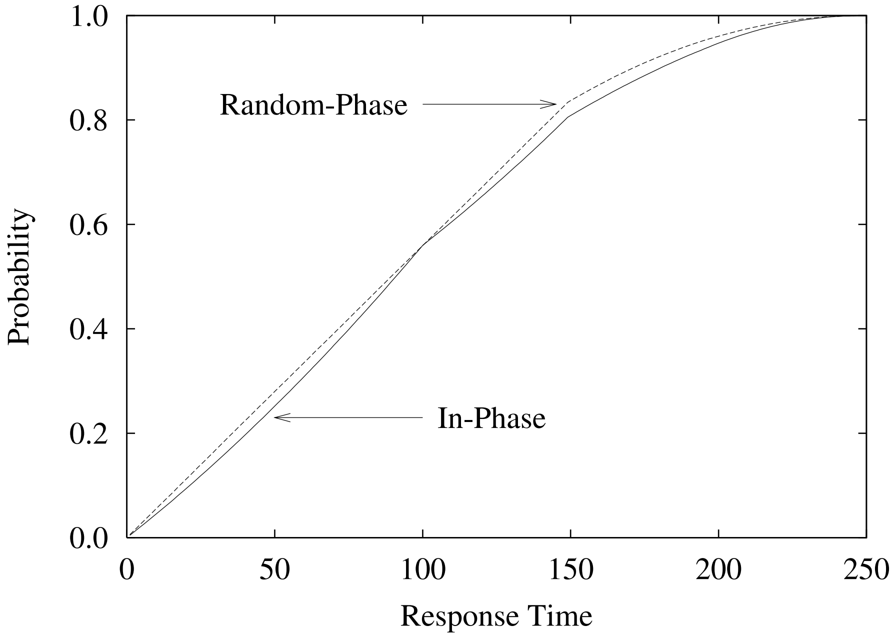
tooth has a more rounded shape due to the exponential distribution of T_1. Finally, the asymptotically decreasing shape of the density curves indicates the combined effect of the execution time distributions of both tasks.

Next we consider the effect of average system utilization on the hypothesis that in-phase response time distributions bound their random-phase release counterparts from below. Mean utilizations of 35% and 95% were obtained by scaling the mean execution times of the system in Table 4.1 by 0.5 and 1.34, respectively. The parameters of the systems are given in Table 4.2. For ease of comparison, the parameters of the previous system are duplicated as system #2. Figures 4.7 and 4.8 give the response time distributions for Systems 1 and 3, with 35% and 95% average utilizations, when execution times are distributed uniformly. The response time distributions when execution times are distributed exponentially are given in Figures 4.9 and 4.10. As can be seen, the response time distributions obtained by examining the response times of jobs in an in-phase busy interval bounds from below those obtained from examining the response times of jobs in a random phase busy interval. At high average utilizations, the curves become indistinguishable.

Despite the large number of systems simulated, we have not observed a case where tasks in which the first jobs are released with arbitrary phases have a lower percentage of deadlines met than the same tasks in which the first jobs are released in-phase. We therefore use in-phase busy intervals in computing a lower bound on the average completion rate using STDA.

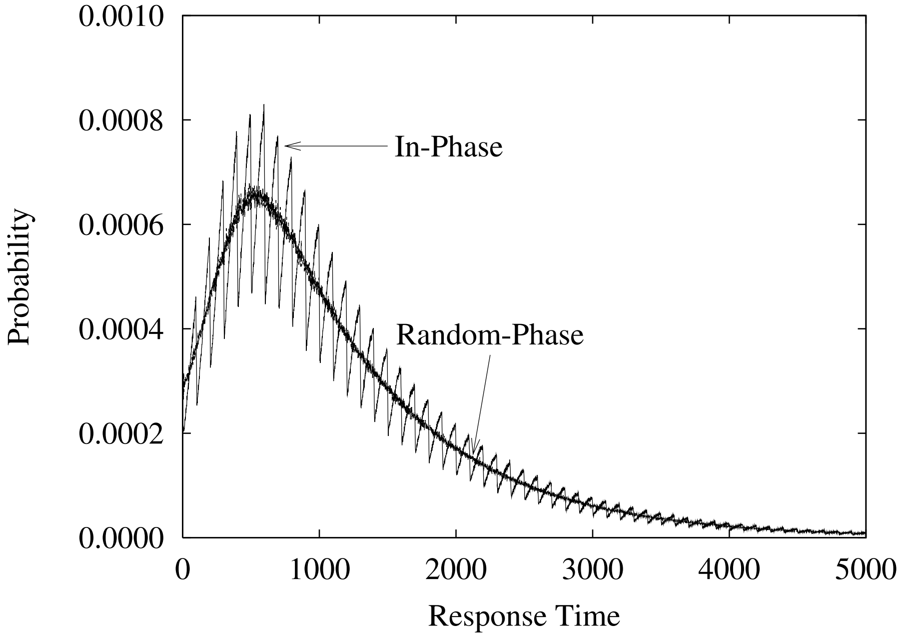


(a) Density

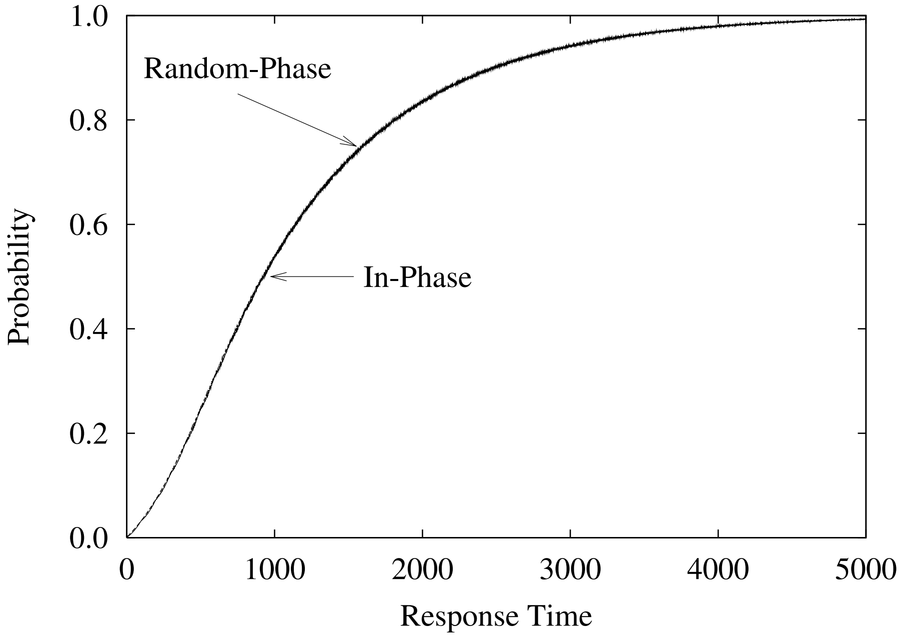


(b) Distribution

Figure 4.7: Average response times of T_2: Uniform, 35% Utilization.

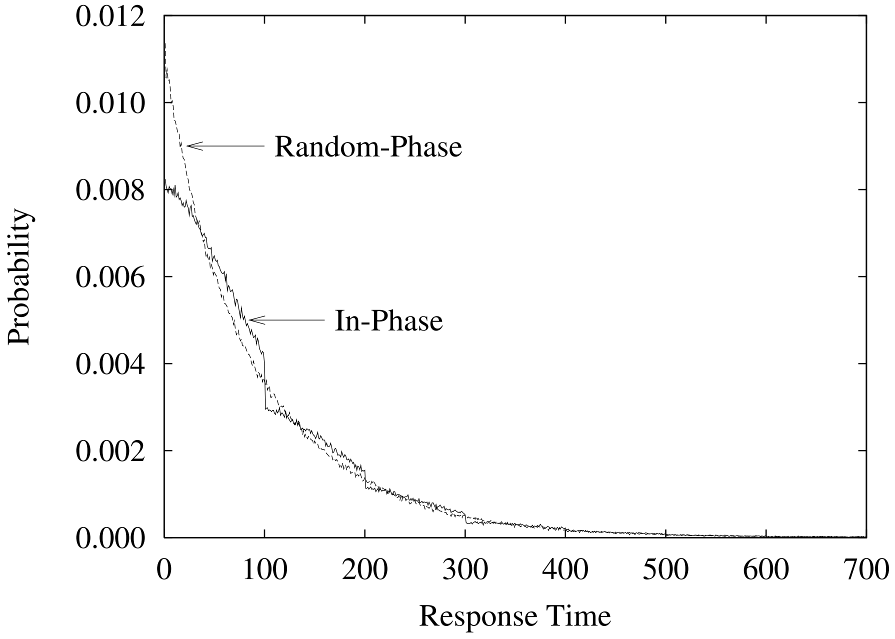


(a) Density

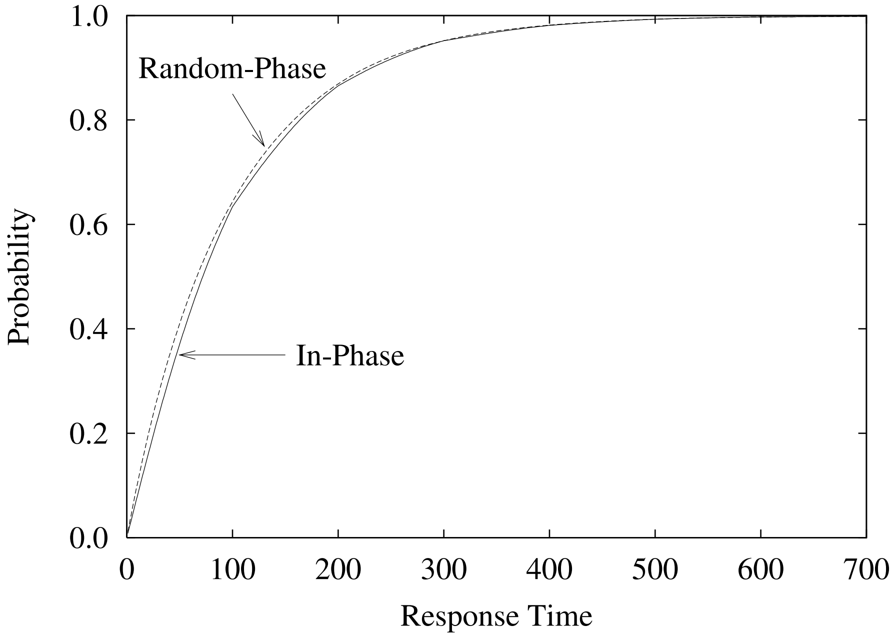


(b) Distribution

Figure 4.8: Average response times of T_2: Uniform, 95% Utilization.

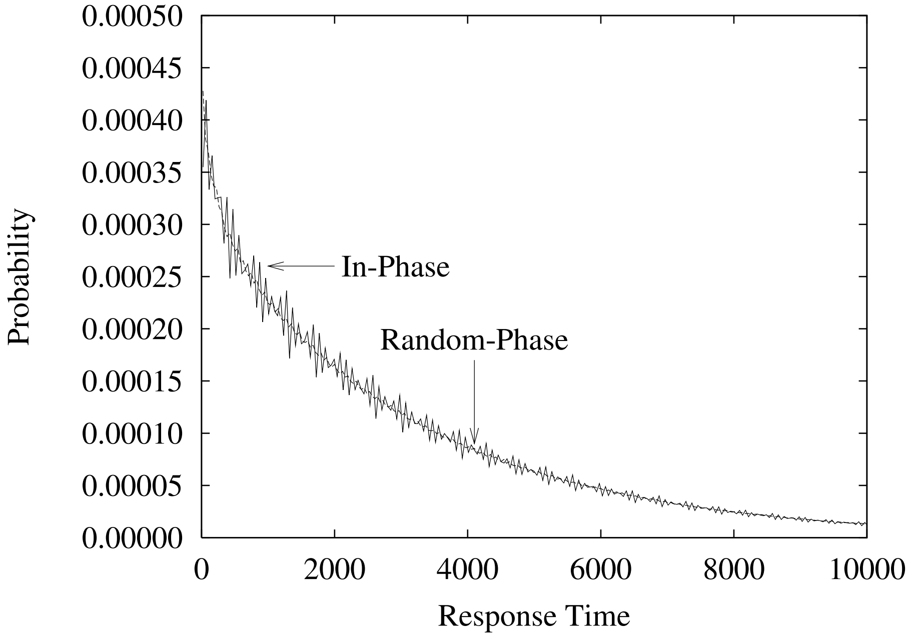


(a) Density

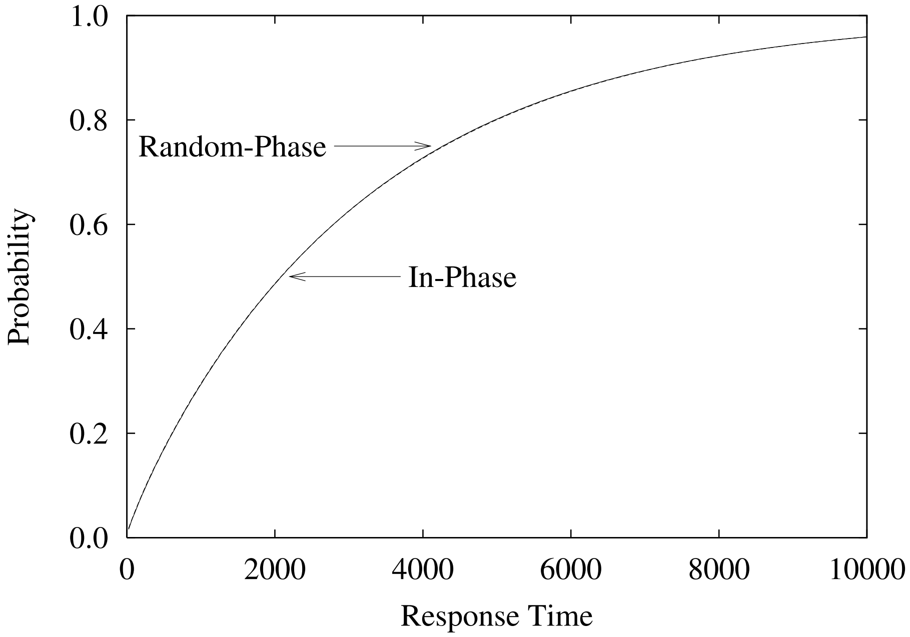


(b) Distribution

Figure 4.9: Average response times of T_2: Exponential, 35% Utilization.



(a) Density



(b) Distribution

Figure 4.10: Average response times of T_2: Exponential, 95% Utilization.

Table 4.3: Tightness of STDA bound for tasks of Table 4.2.

System	T_i	STDA	Simulation In-phase	Ratio	Random-phase	Ratio
1	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
2	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	39.2	80.8 ± 0.1	0.485	81.3 ± 0.1	0.482
3	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	2.6	18.3 ± 0.1	0.142	18.4 ± 0.1	0.141

4.1.5 Comparing STDA to Simulation Results

We now compare the lower bound on the probability of meeting deadlines computed via STDA for the systems in Table 4.2 with the percentage of the jobs in each task meeting their deadlines obtained by simulating the the systems. The minimum number of jobs in a task that were released during the simulation ranged from 1,000 for System 1 to 32,000 for System 3. The simulation results are summarized in Table 4.3. In all cases, the lower bounds computed by STDA are below the percentage of deadlines met obtained by simulation. The table also shows that increasing the average load of the system decreases the tightness of the lower bound. Part of the difference between the lower bound and the simulation results occurs because STDA computes the worst-case probability that jobs released in an in-phase busy interval meet their deadlines rather than the average.

The tightness of the bound computed by STDA is also affected by the variance of the execution time distributions. Table 4.4 gives the parameters for three systems with the same mean utilization but different maximum utilizations due to changing the variance of the uniform distributions. (Once again, the parameters of Table 4.1 are duplicated as system #6 for ease of comparison.) Table 4.5 compares the lower bounds on the percentage of jobs in a task meeting their deadlines, as computed by STDA, to the average obtained by simulation. As expected, decreased variance causes the difference between the STDA bound and the average deadlines met to decrease.

We note that system #4 is a good example of a system which is not schedulable even though the maximum utilization of the system is less than unity. Deterministic analysis does not indicate how close the system is to being schedulable. On the other hand, not only does STDA indicate that T_2 is unschedulable, but also that at least 85.9% of the deadlines will be met. We note that the bound for system

Table 4.4: Parameters of the tasks with different variances.

System	T_i	P_i^-	D_i	E_i^-	$\bar{E}_i$	E_i^+	$\bar{U}$	U^+
4	T_1	300	300	72	100	128	0.708	0.997
	T_2	400	400	72	150	228		
5	T_1	300	300	50	100	150	0.708	1.125
	T_2	400	400	50	150	250		
6	T_1	300	300	1	100	199	0.708	1.411
	T_2	400	400	1	150	299		

Table 4.5: Tightness of STDA bound for tasks of Table 4.4.

			Simulation			
System	T_i	STDA	In-phase	Ratio	Random-phase	Ratio
4	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	85.9	95.3 ± 0.1	0.901	97.6 ± 0.3	0.880
5	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	51.1	92.6 ± 0.2	0.552	94.1 ± 0.2	0.543
6	T_1	100.0	100.0 ± 0.0	1.000	100.0 ± 0.0	1.000
	T_2	39.2	80.8 ± 0.1	0.485	81.3 ± 0.1	0.482

#4 is within 12% of the simulation results. Instead of taking over 10 minutes to simulate the system to the accuracy shown, STDA took less than 1 second.

In the examples shown here, simulating the behavior of the two tasks is reasonable. However, even for the very simple systems considered here, simulation required significantly greater effort than STDA. Hence STDA provides a faster way to determine if the probability of a missed deadlines is acceptable.

4.2 Extending STDA to Handle Mutual Exclusion

So far, we have assumed that jobs do not share resources. Realistically, however, jobs in a system share resources in order to perform meaningful computations. Furthermore, access to the resources may require synchronization to ensure the integrity of the system. In this section, we extend the Stochastic Time Demand Analysis method so that it can deal with mutual exclusion and focus on systems where all resource accesses are made according to the Non-Preemptable Section (NPS) protocol [13].

According to the NPS protocol, a job in the system cannot be preempted while accessing shared resources, i.e., while it is in a critical section. Jobs with a higher priority released after the currently executing job has entered a critical section are blocked until the currently executing job exits the critical section. A *priority inversion* is said to occur whenever a job waits while a lower priority job executes [14]. Uncontrolled priority inversion cannot occur under the NPS protocol if the duration of every critical section is bounded. This fact is formally stated by the following lemma, the proof of which can be found in [13].

Lemma 4.2.1 *The maximum blocking time of a job is the duration of the longest critical section of all lower priority jobs that can block it.*

We now state and prove another lemma which allows blocking to be account for in the formulation of both deterministic and probabilistic time demand analysis techniques.

Lemma 4.2.2 *Only the first job in a busy interval can be blocked.*

Proof: Because of priority scheduling, only jobs with priority equal to or higher than ϕ_i can execute in a level-ϕ_i busy interval in the absence of blocking. Thus a job with priority lower than ϕ_i will be unable to enter its critical section unless it does so before the interval starts. Therefore only the first job in a busy interval can be blocked. □

We note that the duration of the critical sections in a job are random variables. Let $B^k_{i,j}$ denote the duration of the kth critical section of job $J_{i,j}$ when the job executes alone. We let b^{k+}_i be the maximum duration of $B^k_{i,j}$ and let b^+_i be the maximum duration of all critical sections of T_i. Blocking time is accounted for in Time Demand Analysis and Generalized Time Demand Analysis by increasing the time demand of task T_i by the maximum duration of a non-preemptable section of a lower priority task. The time demand according to TDA with non-preemptable sections is

$$w_i(t) = \sum_{1 \le k \le i} \left\lceil \frac{t}{P^-_j} \right\rceil E^+_j + \max_{i<j \le n} b^+_j$$

As before, if the time demand is met by the deadline of the jobs in the task, the task is schedulable. A straight forward way to extend STDA to accommodate blocking increases the time demand in a like manner. Because of Lemma 4.2.2, the time demand of job $J_{i,1}$, i.e., $w_{i,1}(t)$, is increased at the time of its release by the maximum duration of critical sections of lower priority tasks. Thereafter, the probability of timely completion of each job is computed as in Section 4.1.

To maintain responsiveness and maximize performance, most critical sections will be of short duration and will not vary too widely. Thus increasing the time demand by $\max_{i<j\leq n} b_j^+$ will not be overly pessimistic for most systems. Nevertheless, for completeness we outline an approach which takes into account the variable duration of critical sections.

Consider a system of n tasks, each of which has a critical section with a known (independent) duration distribution. The effect of blocking is accounted for in STDA by convolving the time demand function, $w_{i,1}(t)$ with the density function for the maximum blocking times. The distribution of blocking time suffered by a job from T_i due to a job with a single critical section from task T_j is $\mathcal{P}\left[B_j^1(x) \leq x\right]$. Therefore, the distribution of the delay experienced by a job from T_i due to critical sections of lower priority jobs from T_l is the weighted sum of the probability distributions of the blocking time of the lower priority tasks. For a system of n tasks, the distribution of the blocking time for $0 \leq x \leq \infty$ is

$$\mathcal{P}\left[B_{\{i+1,\ldots,n\}}^1(x) \leq x\right] = \sum_{l=i+1}^{n} \alpha_{i,l} \mathcal{P}\left[B_l^1(x) \leq x\right]$$

where $\alpha_{i,l}$ is the probability that a job from T_i is blocked by the critical section of a job in T_l. In general, the value of $\alpha_{i,l}$ must be estimated by measurement or simulation because it depends on the execution histories of the tasks involved. However, assuming the probability that T_i is blocked by a critical section in T_l is proportional to the frequency of job releases, as it would be if tasks consist of straight-line code without conditionals or loops and the first job in each task is released with an arbitrary phase, $\alpha_{i,l}$ can be approximated by

$$\alpha_{i,l} = \lim_{t\to\infty} \frac{\frac{t}{P_l^-}}{\sum_{i<l\leq n} \frac{t}{P_l^-}}$$

Note that tighter bounds can be obtained by accounting for the blocking of individual critical sections of lower priority tasks. Given that task T_l has m_l critical sections,

$$\mathcal{P}\left[B_{\{i+1,\dots,n\}}(x) \le x\right] = \sum_{l=i+1}^{n} \sum_{k=1}^{m_l} \alpha_{i,l}^{k} \mathcal{P}\left[B_l^k(x) \le x\right]$$

where $\alpha_{i,l}^k$ is the probability that a job from T_i is blocked by the kth critical section of a job from T_l. Once again, $\alpha_{i,j}^k$ must be determined by measurement or simulation. Alternatively, it can be estimated by static execution path analysis.

Because of the difficulty in obtaining the weight factors, $\alpha_{i,l}$ or $\alpha_{i,l}^k$, we expect that the maximum blocking delay experienced by jobs of T_i due to critical sections in lower priority tasks will be used instead of the more detailed analysis, except when the duration of some critical sections are very long. As noted before, such critical sections should be very rare as non-preemptable sections with long durations generally lead to performance problems.

4.3 Applying STDA to Distributed Systems

Real-time systems are often designed to make use of multiple processors. In this section, we apply the Stochastic Time Demand Analysis method to systems of distributed tasks with end-to-end deadlines.

By a distributed task, we mean a periodic task each of whose jobs is a chain of *subjobs* which execute sequentially on a set of processors according to a fixed assignment of subjobs to processors. We denote the jth job of task T_i by $J_{i,j}$ and the kth subjob of $J_{i,j}$ by $J_{i,j,k}$. (We extend the periodic task notation in the obvious manner, e.g., the release time of $J_{i,j,k}$ is $r_{i,j,k}$ and its completion time is $c_{i,j,k}$.) Subjobs are constrained to execute in order, i.e., a subjob becomes ready to execute only when its successor has completed. The first subjob in each job has no successor. The end-to-end response time of a distributed job is the length of time from the release of the first subjob in a task until the completion of the last subjob. Each subjob has a fixed priority which can be assigned according to one of the existing distributed fixed priority assignment algorithms [54–56]. (We only require that the period of subjobs be equal to the periods of their end-to-end jobs.) Because the Release Guard Protocol (RGP) has been shown to have better average performance than other methods for synchronizing the releases of subjobs [25], we focus on systems where the RGP is used.

The key idea behind the RGP is to ensure that the inter-release time between

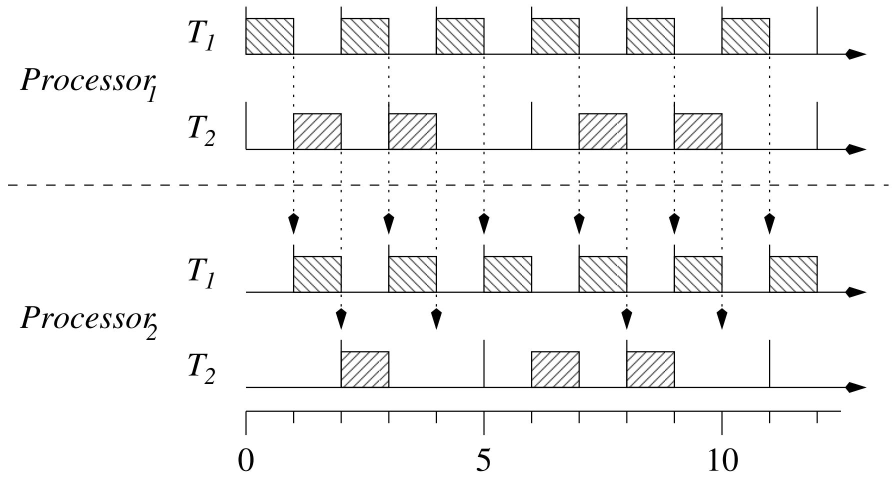


Figure 4.11: Schedule of Distributed Tasks using Release Guard Protocol.

any two consecutive kth subjobs of a task on a processor is no shorter than the period of the task. Associated with the kth subjob of a task is a variable called its *release guard*, $G_{i,j,k}$, which represents the earliest allowable release time of the subjob. The release of a subjob whose predecessor completes before time $G_{i,j,k}$ is delayed until at least $G_{i,j,k}$. Initially, the release guards of all subjobs are zero. Whenever a subjob $J_{i,j,k}$ is released, the release guard of $J_{i,j+1,k}$ is set equal to the sum of the current time and the guaranteed inter-release time of the task containing the subjob. Figure 4.11 shows a distributed system of two tasks with guaranteed inter-release times of 2 and 3, respectively, each containing two subjobs. The first subjob of each task executes for 1 time unit on Processor 1 and the second for 1 time unit on Processor 2. The dotted arrows indicate when the second subjobs can be released because their successors have completed.

Since its release guard is zero, $J_{1,1,2}$ is released on Processor 2 when its predecessor $J_{1,1,1}$ completes at time 1. The release guard $G_{1,2,2}$ for $J_{1,2,2}$ is set to time 3. At time 2, $J_{2,1,1}$ completes, and its successor $J_{2,1,2}$ is released on Processor 2. The release guard $G_{2,2,2}$ is set to time 5. The end-to-end response times of distributed jobs $J_{1,1}$ and $J_{2,1}$ are the sum of the response times of their subjobs, i.e., their response times are 2 and 3, respectively. Following the same process yields the end-to-end response times of other jobs in the tasks.

Note that while the predecessor of subjob $J_{2,2,2}$ completed by time 4, the release guard prevented its release until time 5. Likewise, job $J_{2,4,2}$ is not released until time 11 because of its release guard. Since the inter-release times of subjobs on a processor are no shorter than their guaranteed inter-release times, (Generalized) Time Demand Analysis can be used to compute the maximum response times of subjobs on each processor independent of subjobs on other processors. An upper bound on the end-to-end response time of jobs in a task is the sum of the maximum response times of its subjobs [25]. The maximum end-to-end response time obtained by this method allows us to determine the schedulability of the system. However, we wish to determine a lower bound on the probability that jobs in a task meet their end-to-end deadlines. We must therefore take into account the variations in execution times. Because the release guards of subjobs $J_{i,j,k}$ $j, k > 1$ depend upon the completion time of $J_{i,1,k-1}$, we can no longer consider subjobs on a processor independent of the subjobs on other processors.

We now describe how to apply Stochastic Time Demand Analysis to compute distributions for the end-to-end response times of jobs in a distributed system. The end-to-end response time of a distributed job whose subjobs are released according to the RGP is the sum of the response times of the subjobs. Therefore, the response time distribution of an end-to-end job is determined by convolving the probability density functions of the response time distributions of the individual subjobs in the chain. The response time density functions of the first subjobs of the first jobs in each task are readily computed using STDA. We now show how to compute the response time density functions of the subsequent subjobs.

We begin by noting that the once subjob $J_{2,1,2}$ of Figure 4.11 is released, the RGP will ensure that the subjobs $\{J_{2,2,2}, J_{2,3,2}, \dots\}$ are released at least $P_2^* = 3$ time units after the release of subjobs $\{J_{2,1,2}, J_{2,2,2}, \dots\}$. Thus, the release time of any subjob $J_{i,j,k}$ depends not only upon the completion time of its predecessor, $J_{i,j,k-1}$, but also upon the release time of $J_{i,1,k}$. The release time of $J_{i,1,k}$ is a random variable and hence the release guards of $\{J_{i,2,k}, J_{i,3,k}, \dots\}$ are also random variables. The release guard $G_{i,j,k}$ has probability density function equal to the probability density function of the release time of $J_{i,1,k}$ plus $(j-1)P_i^*$.

Applying the approach in Section 4.1.1, the response time of a subjob depends upon the time demand of jobs which execute on its processor between when it is ready for execution and when it completes, as well as upon the backlog of work that exists on the processor at the time it is ready for execution. Again, subjob $J_{i,j,k}$ is ready for execution no earlier than its release guard, $G_{i,j,k}$. Therefore, the backlog of work that exists at $G_{i,j,k}$ is computed by conditioning on the event that $J_{i,j,k-1}$ completes after $G_{i,j,k}$. Since the completion time of $J_{i,j,k-1}$ is also a random variable $C_{i,j,k}$, we are required to condition upon an event whose time of occurrence is also random variable.[1] The distribution of the backlog at $G_{i,j,k}$ is therefore

$$\begin{aligned}\mathcal{P}[C_{i,j,k-1} \geq G_{i,j,k}] &= \int \mathcal{P}[C_{i,j,k-1} \geq G_{i,j,k},\ G_{i,j,k} = t]\, dt \\ &= \int \mathcal{P}[C_{i,j,k-1} \geq G_{i,j,k} \mid G_{i,j,k} = t]\, dt \\ &= \int \mathcal{P}[C_{i,j,k-1} \geq t]\, g_{i,j,k}(t) dt\end{aligned}$$

where $g_{i,j,k}(t)$ is the probability density function of $G_{i,j,k}$. A closed form solution to the distribution of backlog is unlikely except for very simple systems. In general, the distribution of the backlog must be solved for numerically. Once the distribution of the backlog has been computed, the response time of $J_{i,j,k}$ can be determined by the STDA technique of Section 4.1.

We can simplify the process of computing a distribution for the backlog at the cost of a less tight bound by assuming the worst-case response time of the kth subjob of the first job in each task, $J_{i,1,k}$, so that the release guards are no longer random variables.

4.4 Summary

Using deterministic real-time analysis techniques to design critical soft real-time systems can lead to low resource utilization, increased cost, and poor average performance. An expensive alternative is to simulate critical soft real-time systems to determine their performance. The Stochastic Time Demand Analysis method is a less expensive way to determine if the performance of a system is acceptable. With the Stochastic Time Demand Analysis method, a lower bound on the percentage of jobs in a task that meet their deadlines under a fixed priority scheduling policy can be computed thereby enabling missed deadlines to be balanced against other design goals such as processor utilization or cost.

We have shown how STDA can be used to compute the percentage of deadlines met for systems with non-preemptable critical sections. Because critical sections must be short or the performance of the system will suffer, the response time distribution of the first job in a busy interval need only be offset by the max-

[1]Incidently, computing the response time distributions of jobs whose inter-release times are random variables also requires conditioning on an event whose time of occurrence is a random variable.

imum delay due to blocking, after which the other response time distributions are computed as if no critical sections exist. For systems with long duration critical sections, we have also outlined a complex but more accurate technique for computing the distribution of blocking delay that a job may suffer. The distribution of blocking delay is convolved with the execution time distribution of the first job in a busy interval before computing the response time distributions according to STDA.

Finally, we have shown how STDA can be used in conjunction with the Release Guard Protocol to compute response time distributions for end-to-end jobs in a distributed system and hence obtain a lower bound on the percentage of jobs that meet their deadlines. While the general formulation requires the numerical evaluation of the probability that the predecessor of a subjob completes after the release guard, i.e., conditioning on an event whose time of occurrence is also a random variable, a conservative bound can be obtained by assuming the worst-case release times of the subjobs of the first job in each task when establishing the release guards for the subjobs in jobs after the first.

Chapter 5

Scheduling Overruns

In this chapter, we address the problem of scheduling jobs that may overrun their processor allocations, potentially causing the system to be overloaded. Specifically, we compare the performance of three classes of scheduling algorithms on workloads with and without execution time dependencies. The first class, which contains classical priority scheduling algorithms, as exemplified by DM and EDF, provides a baseline. The second class is the Overrun Server Method which interrupts the execution of a job when it has used its allocated processor time and schedules the remaining portion as a request to an aperiodic server. The final class is the Isolation Server Method which executes each job as a request to an aperiodic server to which it has been assigned.

5.1 Algorithms for Scheduling Overruns

Any algorithm for scheduling jobs with a potential for overrun must meet two criteria if it is to perform well. First, it must guarantee that jobs which do not overrun meet their deadlines. Second, it should maximize the number of overrunning jobs that meet their deadlines or minimize the response times of overrunning jobs. In this section, we discuss two new classes of scheduling algorithms, the first of which achieves the former while striving for the latter. The second relaxes the guarantee that non-overrunning jobs will meet their deadlines in order to perform better on dependent workloads. As a basis for comparison, we also consider the performance of classical fixed and dynamic priority hard real-time scheduling algorithms.

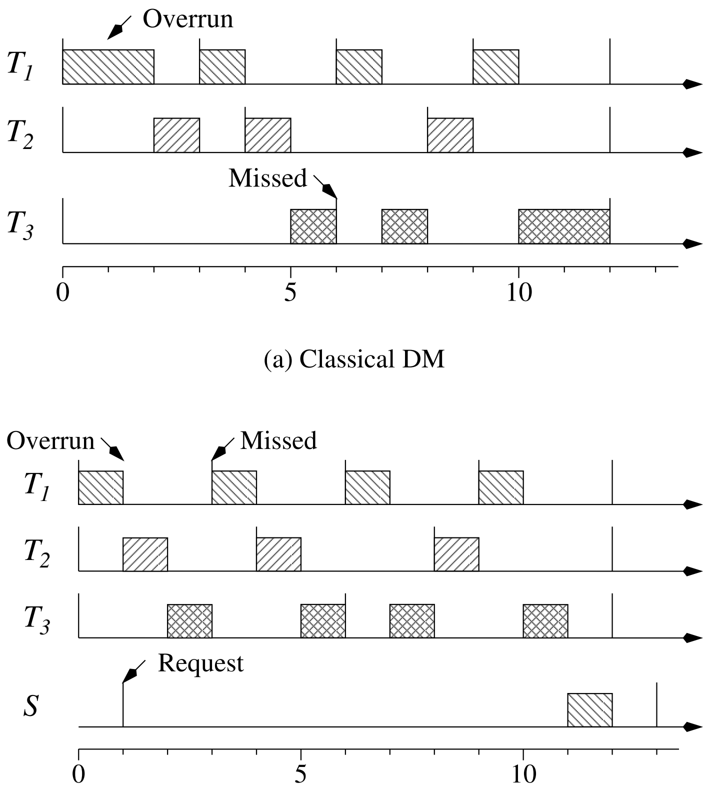
The algorithms used as the baseline are the Deadline Monotonic (DM) [30] and Earliest Deadline First (EDF) [1] scheduling algorithms which where discussed in Chapter 2. These algorithms are optimal in that the DM and EDF algorithms ensure that no deadlines are missed if all the tasks can be scheduled without missing a deadline by any fixed or dynamic priority algorithm, respectively. The DM algorithm also has the desirable property that an overrun will not cause a job of a higher priority task to miss its deadline. In contrast, it is well known that the EDF algorithm behaves unpredictably upon overrun. However, EDF has a higher schedulable utilization than DM and hence makes better use of resources as long as the maximum utilization is no greater than one. The desire to combine the predictability of DM with the optimal schedulable utilization of EDF motivates the development of the two new classes of algorithms.

5.1.1 Overrun Server Method

The Overrun Server Method (OSM), is both a simplification and an extension of the Task Transform Method proposed by Tia *et al.* [32]. Under the OSM, a job is released for execution and scheduled according to the algorithms in the baseline class. At the time of overrun, the execution of the overrunning job is interrupted and the remaining part of the job is released as an aperiodic request to a server. A Sporadic Server (SS) [20] is used to execute the requests in fixed priority systems while either a Constant Utilization Server (CUS) [35] or Total Bandwidth Server (TBS) [21] is used to execute requests in a dynamic priority system. We denote the former as OSM-DM and the latter as OSM-EDF.

Sporadic Servers are specified by a replenishment period and an execution budget. A SS demands no more time within any time interval than a corresponding periodic task with the same period and maximum execution time. Thus, the schedulability of a system containing Sporadic Servers can be determined by the methods applicable to systems of periodic tasks. We use a particularly aggressive implementation of the Sporadic Server which reduces the average response time of requests [57]. A Sporadic Server has a separate ready queue so it makes sense to consider various queueing disciplines. We consider the First-Come-First-Served (FCFS), Deadline Monotonic (DM), and *Shortest time Remaining at Overrun* (SRO) queue disciplines. (In the latter, the priority of a request is inversely proportional to the amount of work remaining at the time of overrun.)

As an example of OSM-DM, consider a system of three tasks. Task T_1 has a period of 3 and a guaranteed execution time of 1. Task T_2 has a period of 4 and a guaranteed execution time of 1. Task T_3 has a period of 6 and a guaranteed



(a) Classical DM

(b) OSM-DM

Figure 5.1: Behavior of DM upon Overrun.

execution time of 2. In Figure 5.1(a), job $J_{1,1}$ overruns. As a result, T_3 misses a deadline at time 6 when the tasks are scheduled according to the DM algorithm. Figure 5.1(b) shows the same workload but with a Sporadic Server having a replenishment period of 12 and an execution budget of 1. According to deterministic schedulability theory, the three tasks and the server are schedulable according to DM on the basis of guaranteed execution times. Note that now the jobs of T_3 meet their deadlines in spite of the overrun by a job in T_1.

The schedulability of an OSM-EDF system using either a CUS or a TBS can also be determined by the methods applicable to systems of period tasks because they demand no more time than a corresponding task with the same utilization. The difference between the two algorithms is that a TBS uses background time whereas a CUS does not. Typically, a CUS is preferred when there are several servers and it is undesirable for the servers to compete for background time. However, one would expect the average response time of requests executed by a CUS to be greater than if executed by a TBS. We later show results that support this conclusion.

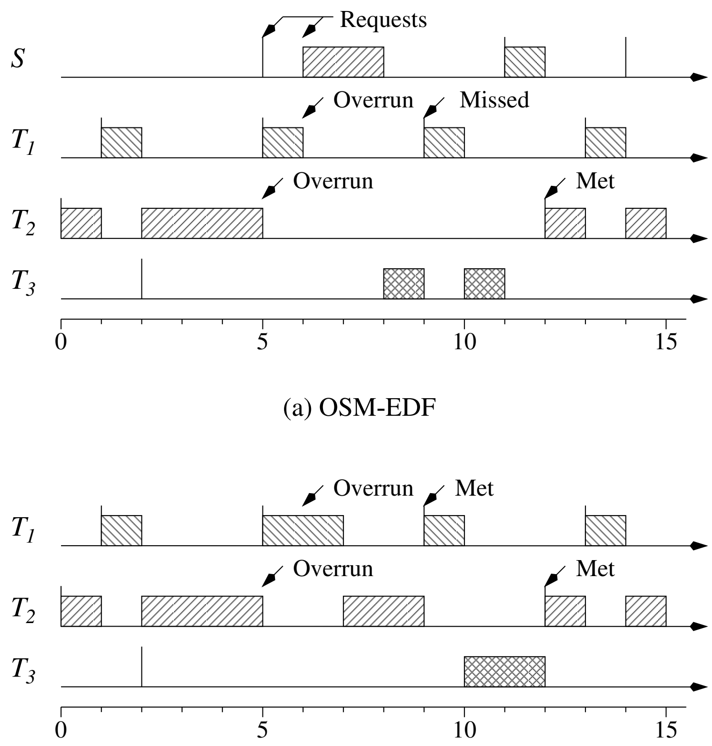
5.1.2 Isolation Server Method

The Isolation Server Method (ISM) is named for its use of a SS, CUS, or TBS to isolate other parts of the system from the effects of an overrunning job. Jobs are submitted as aperiodic requests to the server assigned to their task at the time of their release and execute completely under server control. A server may be assigned to execute jobs from multiple tasks. Whereas OSM requires a portion of the processor separate from the tasks to be allocated to servers, ISM allocates portions of the processor to servers and does not allocate time to the tasks. Because jobs of a task are released as requests to a server, an overrunning job under ISM can only delay the completion of jobs from tasks assigned to that server. The execution of jobs in tasks not assigned to that server are isolated from the overrunning job. Unlike OSM, ISM cannot guarantee all jobs with execution times which do not exceed their guaranteed execution times will complete by their deadlines.[1] We denote fixed priority ISM by ISM-DM and dynamic priority ISM by ISM-EDF.

As an example, consider another system of three tasks scheduled according to the EDF algorithm together with a TBS. Task T_1 has a period of 4 and a guaranteed execution time of 1. Task T_2 has a period of 12 and a guaranteed execution time of 4. Task T_3 has a period of 24 and a guaranteed execution time of 2. Under OSM-EDF, the overrun server has a replenishment period of 3 and an execution budget of 1 while ISM-EDF has a separate server per task with utilizations equal to the utilizations of the respective tasks. According to deterministic schedulability theory, the system is schedulable under both OSM-EDF and ISM-EDF.

Suppose that a job of T_2 overruns by 2 at time 6 and a job of T_1 overruns by 1 at time 7, as shown in Figure 5.2. The overrunning job of T_1 misses its deadlines under OSM-EDF but completes in time under ISM-EDF. Also, both overrunning jobs complete earlier under ISM-EDF than under OSM-EDF because ISM servers execute jobs at their original priority, while OSM servers execute the overrun portion of jobs at the priority of the corresponding server. In this case, the isolation server has a lower priority than T_1 while it executes the overrunning job

[1]The ISM can be considered a special case of the OSM in which the guaranteed execution time of each task is zero and hence all jobs immediately "overrun" and are released directly to the overrun servers.



(a) OSM-EDF

(b) ISM-EDF

Figure 5.2: Behavior of Server upon Overrun.

of T_2.

5.2 Comparison Methodology

In this section, we describe the methodology used to compare the performance of the three classes of algorithms. The average performance of the system is obtained by discrete event simulation. We first discuss the criteria used to compare the performance and then describe the workload used in the comparison.

5.2.1 Performance Criteria

In Section 5.1 we stated what the ideal behavior of an algorithm for scheduling jobs in the presence overruns should be. First and foremost is the requirement that jobs which do not exceed their guaranteed execution times should never miss a deadline. OSM meets this condition by design. ISM relaxes this condition slightly in that an overrunning job may delay the completion of subsequent jobs assigned to the same server. The next condition is that the algorithm maximizes the number of deadlines met and minimizes the response times of overrunning jobs. Thus, the fraction of deadlines met by jobs in each task and the average response time of jobs in a task are the metrics use in our comparison study.

It is clear that the above metrics cannot be directly compared for different workloads because, for example, the percentage of deadlines missed depends upon the execution time and deadline of the tasks, both of which vary across workloads. For this reason, we use the ratio of a metric measured for one algorithm against the same metric measured for another algorithm on the same workload, averaged over all the workloads. As an example, if the average ratio of deadlines met is 1.0, then the two algorithms have equivalent performance on the average. Likewise, an average ratio of deadlines met of 1.25 indicates that the first algorithm performs 25% better than the second on the average.

Finally, there are two perspectives from which to compare performance. The first is from the perspective of system performance and the second is from the perspective of task performance. The average ratio of a metric from the perspective of system performance is computed as a weighted sum of the metric for each task. The weight for a task is the fraction of all jobs that belong to the task. The average ratio of a metric from the perspective of task performance is just the sum of the metric for each task. As we will see, the system perspective of performance biases the comparison in favor of the fixed priority baseline algorithm since the task with the highest priority also releases the most jobs in a given length of time. Because our objective is to schedule overloaded critical real-time systems to maximize the deadlines met and minimize the average response time of each task, we are primarily interested in the performance of tasks rather than overall system performance. For completeness, however, we present the performance from both perspectives in the following sections.

5.2.2 Workload Generation

The performance of a scheduling algorithm depends upon the average utilization of the system. At low utilizations, sufficient time exists for nearly all overrunning jobs to complete in time. As the utilization increases, overrunning jobs become more likely to miss their deadlines. At some point, overruns will cause the system to be overloaded. As long as the average utilization of the system is less than one, the system will continue to function, albeit with increasingly reduced per-

Table 5.1: Overrun Simulation Parameters.

Parameter	Values
Distribution Types	Uniform, Exponential
Average Utilizations	0.50, 0.75, 0.90, 0.95
Dependency Patterns	$\{1\}$, $\{1, 1\}$

formance. For the workloads used in this study, an average utilization of 0.50 is sufficient for nearly all jobs to met their deadlines. The average utilizations we consider are 0.50, 0.75, 0.90 and 0.95.

Given the average utilization of the system, the average utilization of a task is obtained by multiplying the average system utilization by a utilization factor for the task. The utilization factors of the tasks in a system are uniformly distributed in the range $(0, 1]$ and normalized such that the sum of the factors equals 1.0. The execution time of jobs in a task are random variables with a common distribution, either uniform or exponential. The mean execution time of a job is equal to the mean utilization of its task multiplied by the (constant) period of the task. The minimum execution time is 1 time unit. The periods of the tasks are constant and are uniformly distributed in the range $[1000, 10000]$.

Overruns are often caused by common factors and hence the execution times of jobs are likely to be correlated. For dependent execution times, we model dependencies as being exclusively between jobs in a task and following a fixed pattern. The pattern is represented by a list of execution time factors, with a mean value of 1.0, representing the correlations between the execution times of consecutive jobs. For example, suppose that the pattern is $\{2, 0.5, 1, 0.5\}$. If the mean execution time of a set of dependent jobs is 100, the actual execution times of the jobs are $\{200, 50, 100, 50\}$. (The mean execution time of a set of dependent jobs is computed by the previous procedure.) Dependence patterns are varied and clearly application dependent. In our current study, we examined the performance of the algorithms for the (independent) pattern $\{1\}$ and for the (dependent) pattern $\{1, 1\}$.

For each average system utilization, distribution type and dependency pattern, we generated 100 systems, each consisting of 8 tasks. Each of the tasks in a system had a minimum of 2000 jobs. A summary of the parameters which we varied during the performance study are given in Table 5.1.

5.3 Baseline

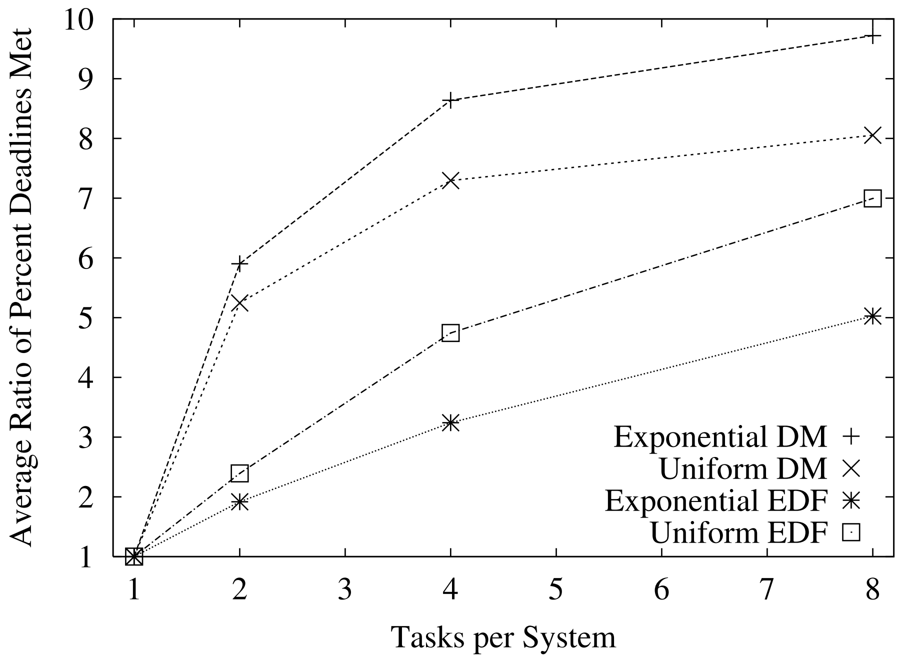
We compare the performance metrics for equal release times (in-phase release) and random release times. For the random release time case, 30 runs with the initial release time of each task uniformly distributed in $[0, P_i^-]$ were performed. The results clearly indicate that the average performance of the system does not depend on the initial phase of the tasks.

Another factor which may influence performance is the number of tasks in the system. Figure 5.3 shows a comparison of the performance of the DM and EDF schedulers on systems with between 1 and 8 tasks. The results are presented as the average of the ratios of the performance of systems with n tasks to the performance of a system with 1 task. The average system utilization is 95%. The figure shows that the average performance of the system improves with increased numbers of tasks. This occurs because distributing a given system load across more tasks decreases the average overrun per task. The average response time decreases as the average overrun decreases making it more likely for jobs to meet their deadlines. We use 8 tasks per system for the remainder of the study.

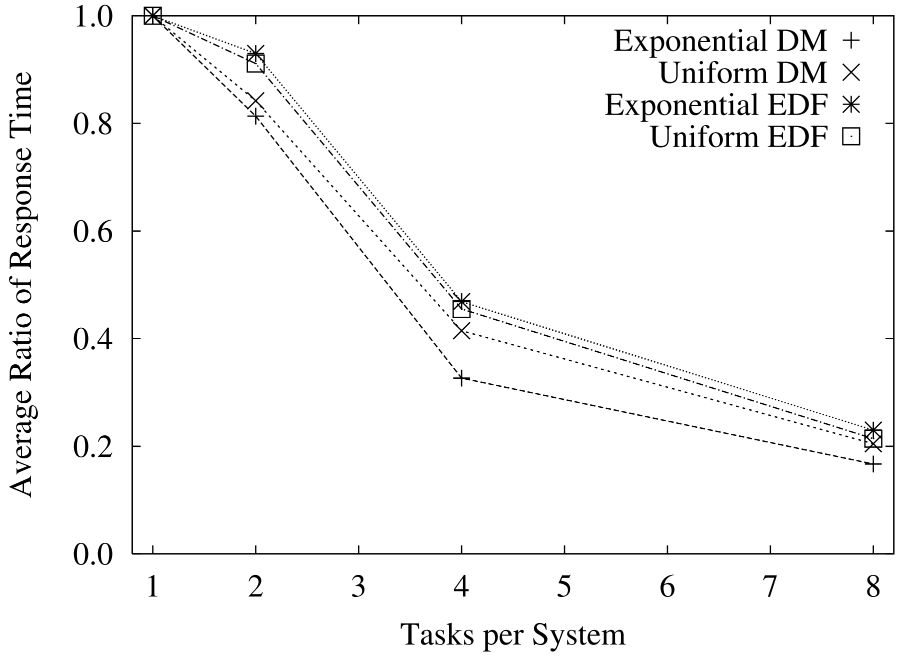
Finally, we compare the performance of EDF and DM on independent and dependent workloads. As can be seen in Figure 5.4, the average performance of the EDF policy is clearly worse than the average performance of DM from a system perspective. Considering performance from the perspective of a task, the percentage of deadlines met for EDF slightly exceeds that of DM for the independent uniform distribution at high average utilizations, as Figure 5.5 shows. In general, DM performs better than EDF, however, particularly with respect to average response time.

5.4 Overrun Server Method

As discussed above, one way to schedule jobs with the potential for overrun is to suspended a job when it has executed for its guaranteed execution time and release the remainder of the job as a request to an overrun server. Only jobs with execution times in excess of their guaranteed values may miss their deadlines. The main issues in using overrun servers to schedule jobs are the number of servers to use, the assignment of jobs to servers, the parameters of each server, and the queueing discipline employed by the server.

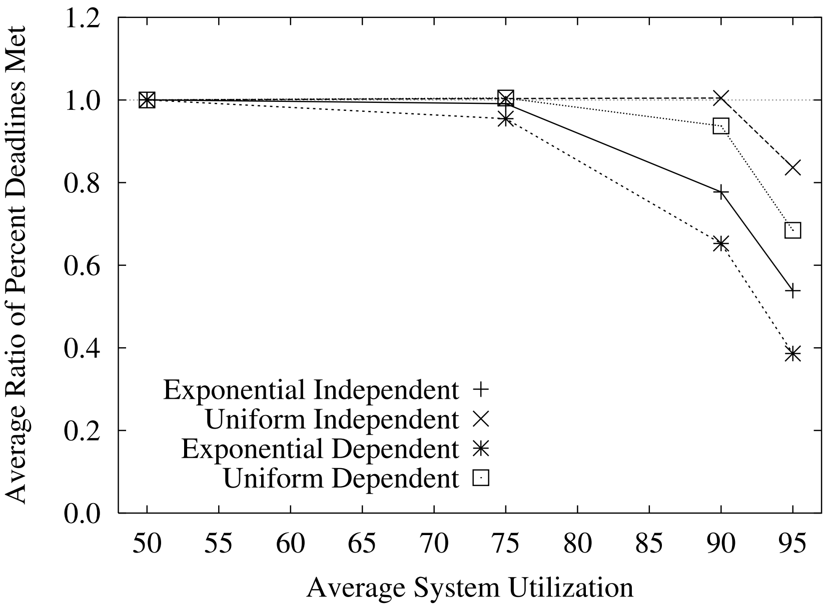


(a) Deadlines Met

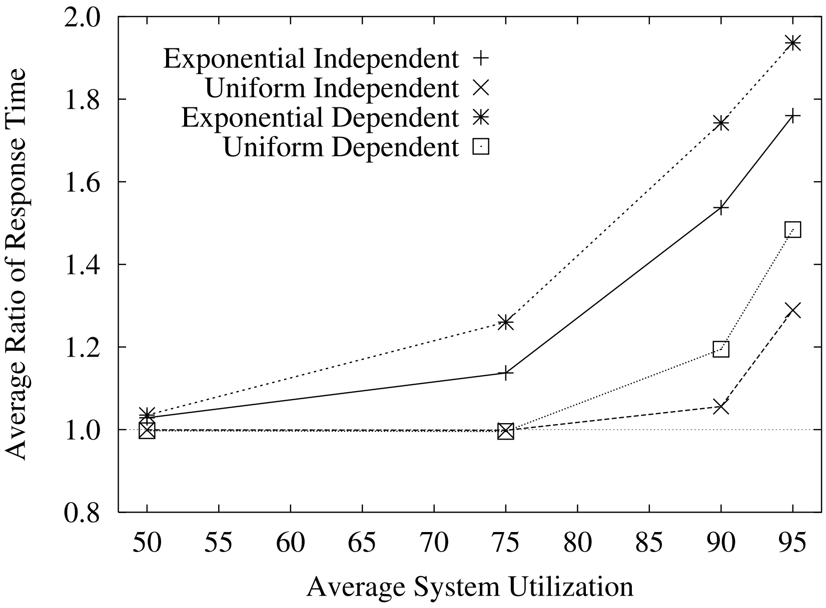


(b) Response Time

Figure 5.3: Performance vs. Number of Tasks: the average of the ratios of the metrics for systems with n tasks to the metrics of a system with 1 task.

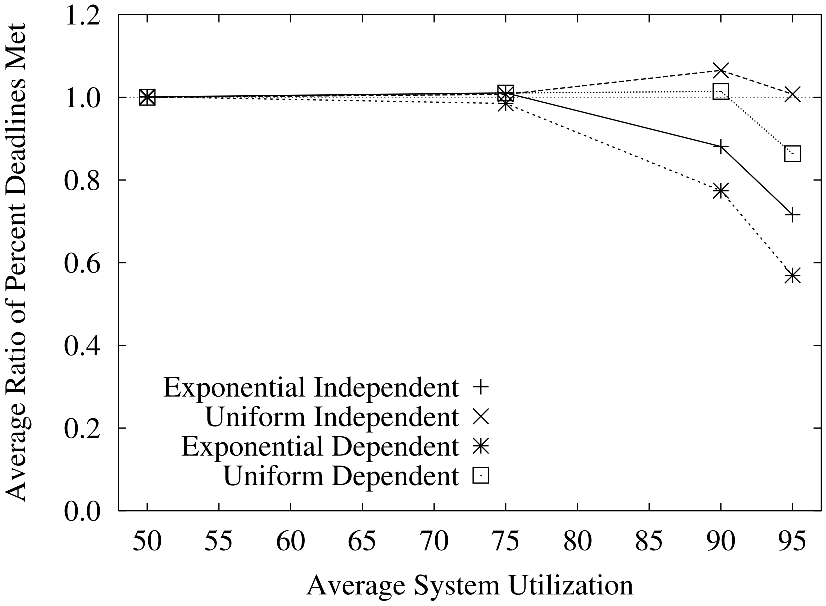


(a) Deadlines Met

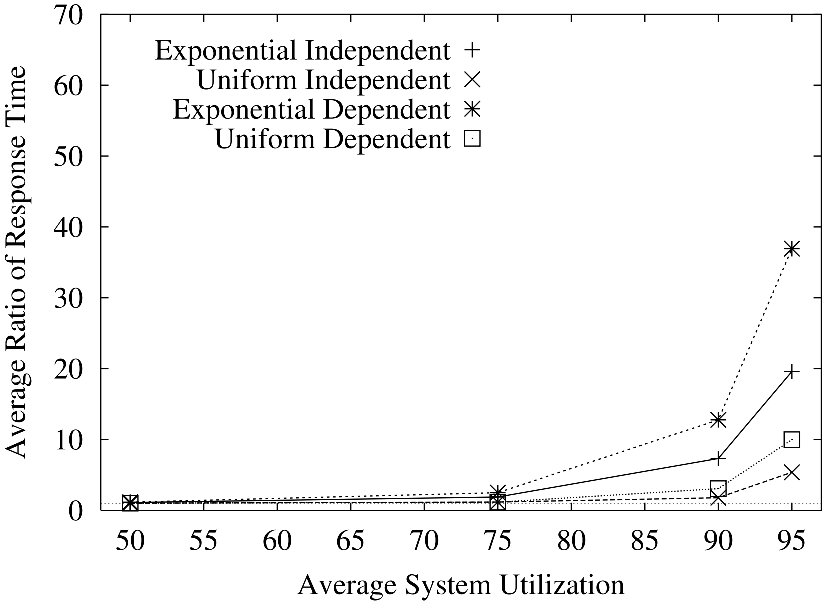


(b) Response Time

Figure 5.4: EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics of systems with a DM scheduler.



(a) Deadlines Met



(b) Response Time

Figure 5.5: EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics for systems with a DM scheduler.

5.4.1 Number of Servers

The number of servers can range from a single server for all tasks to a server for each overrunning task. Clearly the total average utilization of the servers can be at most $1 - \sum_i \bar{U}_i$, without the potential for causing non-overrunning jobs to miss their deadlines. (It may be less than this under fixed priority scheduling policies due to a lower schedulable utilization.) The question is how to distribute the uncommitted utilization of the processor. If the remaining utilization is divided without regard for the average overrun assigned to each server, e.g., dividing the remaining utilization evenly, some servers may have a proportionally greater average load than others. This utilization distribution will likely cause jobs to have worse response times and be less likely to complete by their deadlines. On the other hand, dividing the remaining utilization proportionally according to the average overrun assigned to each server ensures that servers will receive processor time proportional to their load.

The results indicate that the performance of a single server for all tasks and a server per task bound the performance of systems with intermediate numbers of servers. In addition, there is no significant difference in performance between different assignments of tasks to servers when the remaining utilization is assigned in proportion to the average overrun seen be each server since the load on the servers are equal. For brevity, we present only the results for systems with 1 or 8 servers having groups of consecutive tasks assigned to servers.

5.4.2 Deadline Monotonic

Given the server utilization, an overrun server in a fixed priority system requires the specification of a replenishment period in order to establish a budget for the server. It also requires a scheduling discipline for its job queue. For simplicity, we consider the choice of replenishment period and queue discipline for a system with a single overrun server.

The literature pertaining to the service of aperiodic requests almost universally suggests that the priority of a server should be greater than that of the tasks. To determine the best server period, we simulated the behavior of a simplified set of 100 systems consisting of 3 tasks with 1000 jobs per task and 9 priorities chosen to be slightly higher than, equal to, and slightly lower than the priority of each task. The execution time of jobs in the highest priority task were taken from a uniform or exponential distribution. The execution times of jobs in the other tasks were constant. The task periods were in the range $[100, 1000]$. The average utilizations of the systems were 75%, chosen to be less than the Liu and Layland bound of 75.7% thereby guaranteeing schedulability on the basis of mean execution times.

Initial results presented in [58] indicated that the choice of Sporadic Server priority does not significantly affect the average response time or percentage of deadlines met. Those results were based on simulation results with 30 systems. Increasing the number of systems to 100 allowed statistically significant differences to appear. As Figure 5.6 shows, selecting a server priority slightly higher than the task with variable execution time increases deadlines met for exponentially distributed execution times, where as selecting a server priority slightly lower than that the task with variable execution time increases deadlines met for uniformly distributed execution times.[2] The same conclusions follow when response times are considered. For convenience, we let the period of all servers be mean of the range of periods in the experiments that follow.

Next we consider the choice of queue discipline for the Sporadic Server ready queue. We use FCFS as a baseline (as in [32]) and compare its performance from the system perspective with DM and SRO on the workloads discussed in Section 5.2.2. Figure 5.7 shows that the percentage of deadlines met when DM is used as the queue discipline is higher than when FCFS is used. Likewise, more deadlines are met when SRO is used. The average response times with DM and SRO are better than FCFS.[3]

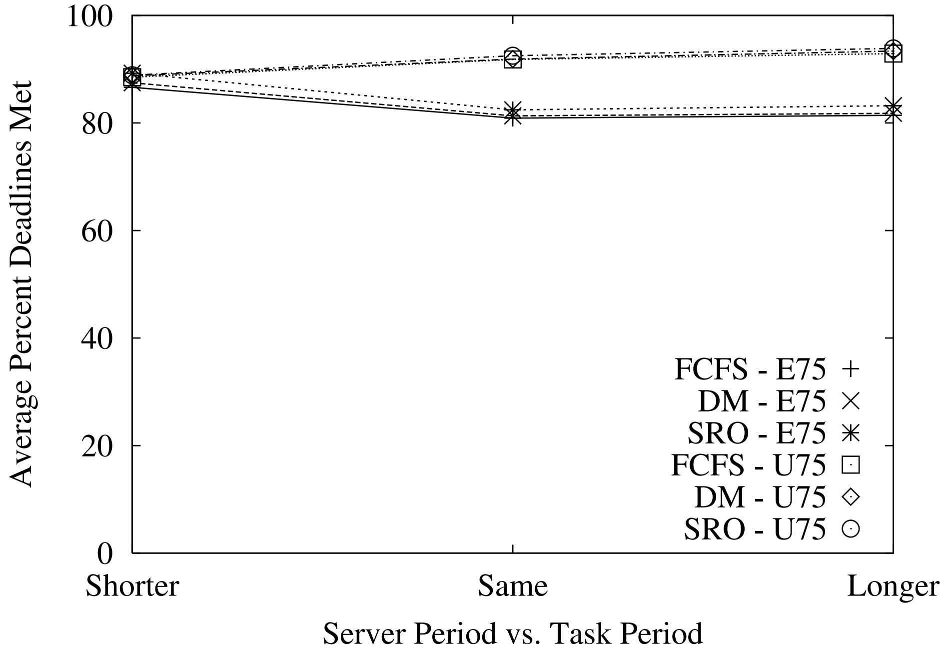
The latter result may be surprising because SRO performs poorly as a real-time scheduling algorithm since it prioritizes jobs on the basis of remaining work at release rather than on task deadlines or periods. However, it minimizes response time and thus it allows more overrunning jobs to complete by their deadlines when used as a SS queue discipline. We use SRO as the queue discipline in the experiments that follow.

5.4.3 Earliest Deadline First

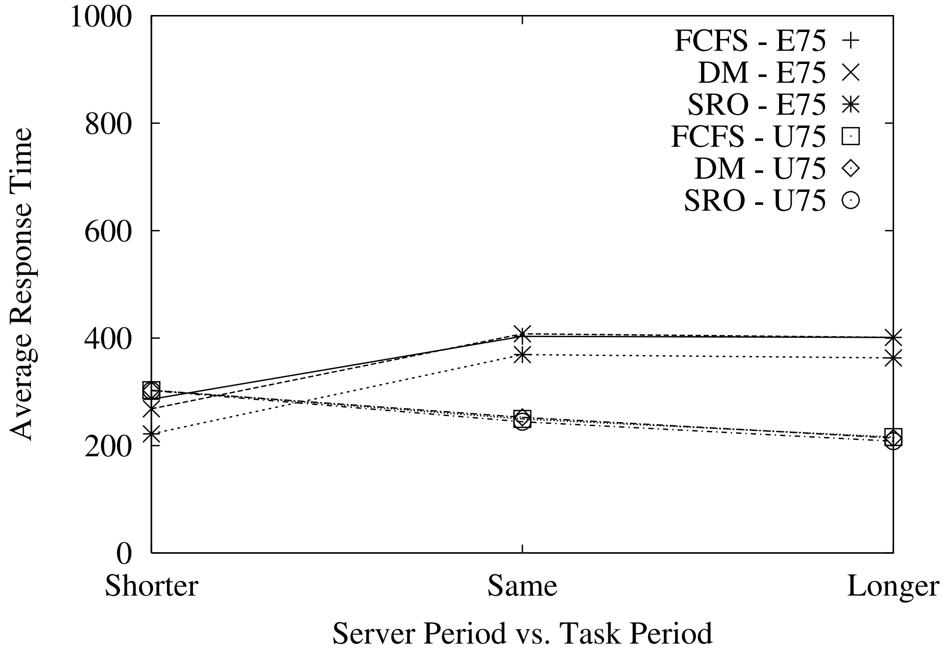
We now consider the OSM in a deadline-driven system. As stated earlier, we use either a CUS or a TBS in connection with an EDF scheduling policy. The

[2] The differences between FCFS, DM and SRO queue disciplines are not statistically significant except for exponentially distributed execution times where SRO performs better than FCFS and DM.

[3] The average response time ratios of DM with respect to FCFS appears to exceed 1.0 but is in fact within $\pm 5\%$ at 95% confidence.

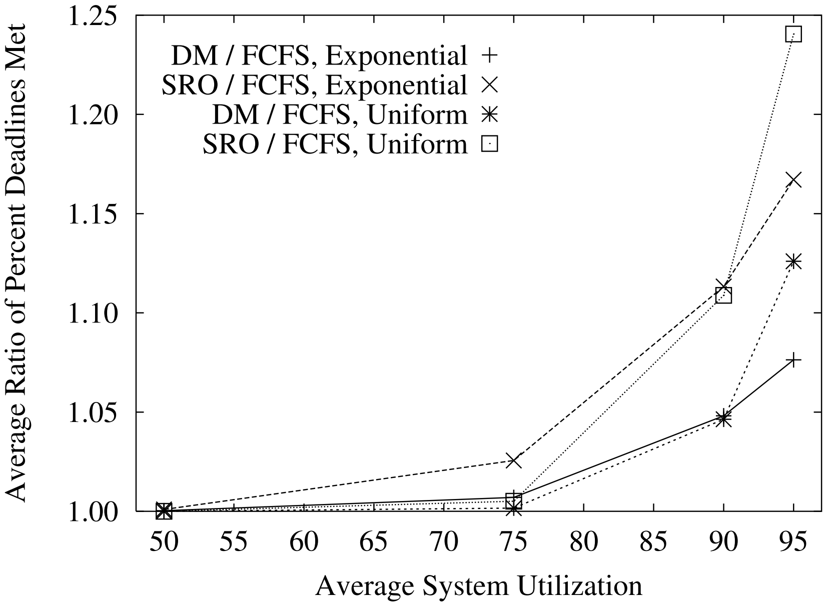


(a) Deadlines Met

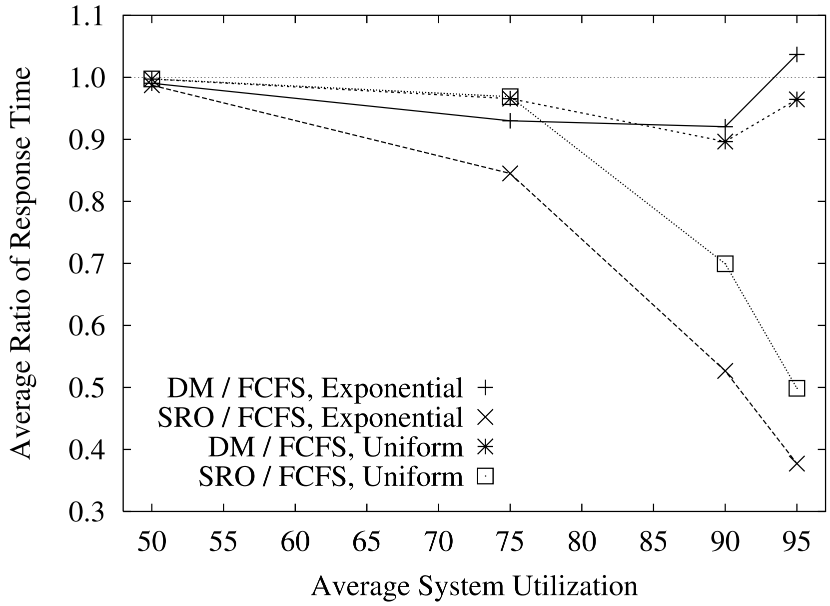


(b) Response Time

Figure 5.6: Effect of Sporadic Server Period: the average performance as a function of the relationship between the server and the task periods.



(a) Deadlines Met



(b) Response Time

Figure 5.7: Performance of Sporadic Server Queue Disciplines: the average of the ratios of the metrics for Sporadic Servers with DM or SRO queue disciplines to the metrics with Sporadic Servers with FCFS queue disciplines.

CUS and TBS overrun servers are specified by establishing server utilizations and assigning tasks to servers in the manner discussed earlier.

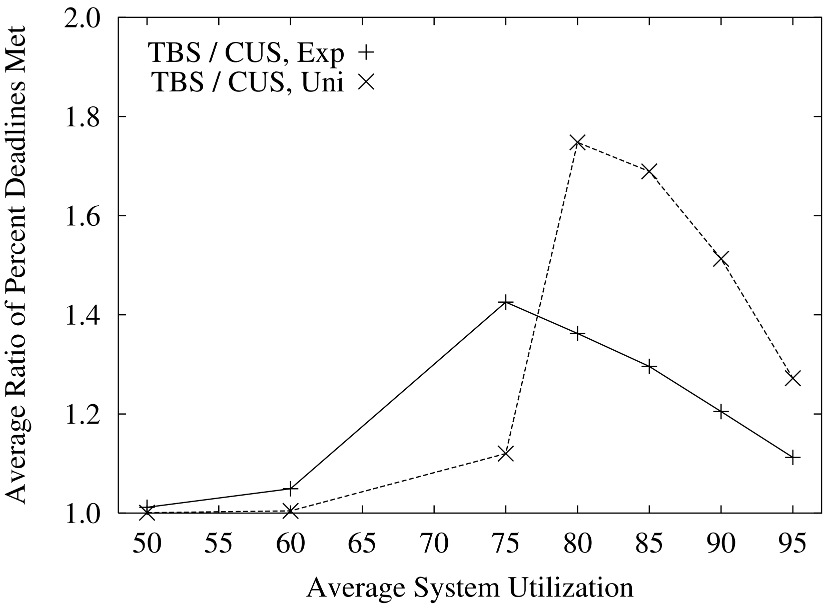
Again, a CUS server and a TBS server behave identically except that the latter is allowed to compete for background processing time. Because of this, jobs executed by a TBS server may complete earlier if background time is available, possibly meeting their deadlines. Thus we would expect a TBS to perform better than a CUS as an overrun server. Indeed, the average response times using a TBS are much lower than those using a CUS, as Figure 5.8 shows, particularly at moderate to high average system utilization. The percentage of deadlines met is also greater for a TBS. The shape of the curves for deadlines met comes from two opposing effects. At low load, the average overrun is small so a TBS and a CUS perform similarly. Increasing the load causes an increase in the average overrun which a TBS executes earlier. At the same time, increasing the load decreases the amount of background time that a TBS can exploit. As the average load approaches 100%, the average deadline met once again becomes similar because the amount of background time approaches zero.

There is another subtle reason for preferring a TBS over a CUS in overloaded systems. Both a TBS and a CUS compute the server deadline as the greater of the current time or the previous deadline of the server plus the execution time of the job being released divided by the guaranteed utilization of the server, i.e., for the server per task configuration

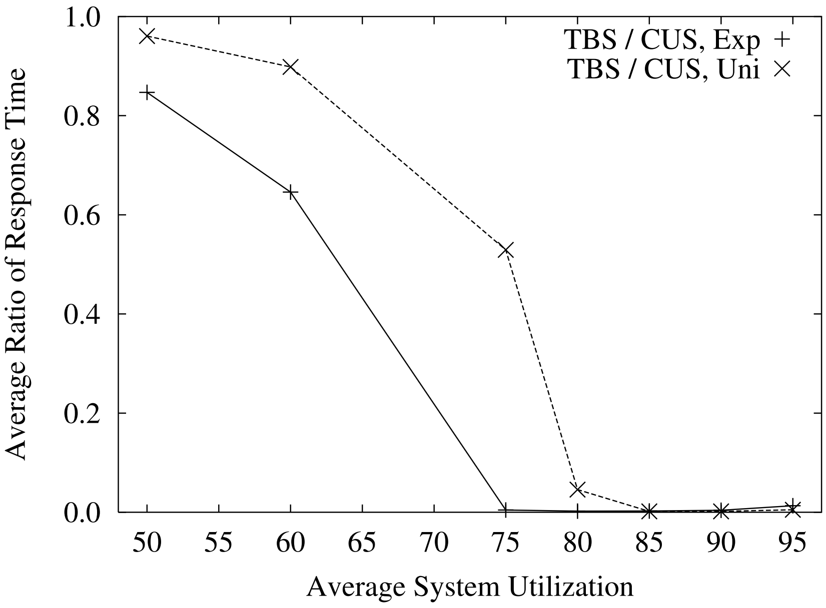
$$d'_{i,j} = \max(d'i, j-1, t) + \frac{e_{i,j}}{U_i^*}$$

This ensures that servers in a fully loaded system do not use more than their allocated bandwidth. A job being released is given the deadline of its server for scheduling according to EDF. (The deadline by which a job should complete, its *completion deadline* $d_{i,j}$, is distinct from the *scheduling deadline* of the server $d'_{i,j}$. The completion deadline is a constraint that the system tries to maintain whereas the scheduling deadline is an artifact of scheduling.) Jobs executed by a TBS are allowed to compete for background time by being entered into the OS ready queue as soon as they are released. Jobs executed by a CUS must wait to enter the ready queue until time $\max(d'_{i,j-1}, t)$ and thus cannot compete for background time.[4]

[4]As an optimization, the scheduling deadline of the CUS can be reset to the current time when the processor is idle. During a busy interval, however, a CUS does not compete for background time while a TBS does.



(a) Deadlines Met



(b) Response Time

Figure 5.8: Performance of TBS vs. CUS: the average of the ratios of the metrics for systems with Total Bandwidth Servers to the metrics for systems with Constant Utilization Servers.

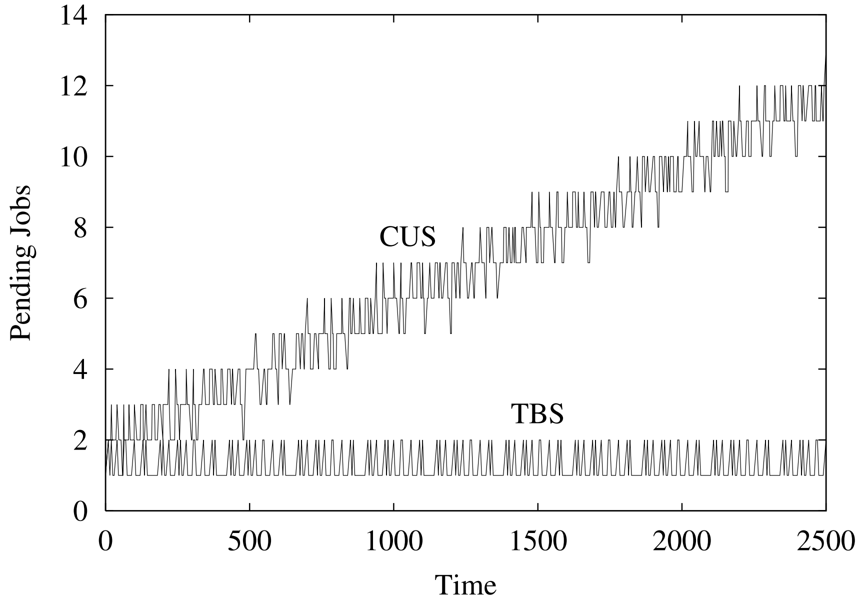


Figure 5.9: Queue Length of TBS vs. CUS: the number of released but not yet completed jobs as a function of time at 95% average utilization showing that the overhead of queue management grows for Constant Utilization Servers.

As a result, a CUS must maintain a separate wait queue for the jobs which are released but not yet eligible to compete for the processor.

Figure 5.9 shows the minimum number of released but not yet completed ("pending") jobs as a function of time for a system of three tasks with an average utilization of 95%. All of the pending jobs are in the ready queue of the TBS, while all but at most one of the pending jobs are in the wait queue of the CUS. Clearly, the length of the wait queue of the CUS is growing. The number of jobs in the wait queue of the CUS will continue to grow as more jobs are released. Meanwhile the overhead of maintaining the wait queue increases until it overwhelms the scheduler. We call the utilization at which the overhead of maintaining the wait queue overwhelms the scheduling of jobs the *saturation utilization* of the server. The saturation utilization of a TBS is greater than a CUS because a TBS completes jobs more quickly through its use of background time. As the average utilization approaches 1.0, however, all servers will saturate. Because a CUS has a lower saturation utilization and worse performance than a TBS, we conclude that a CUS should not be used for scheduling highly overloaded systems.

5.4.4 Performance of OSM vs. Baseline

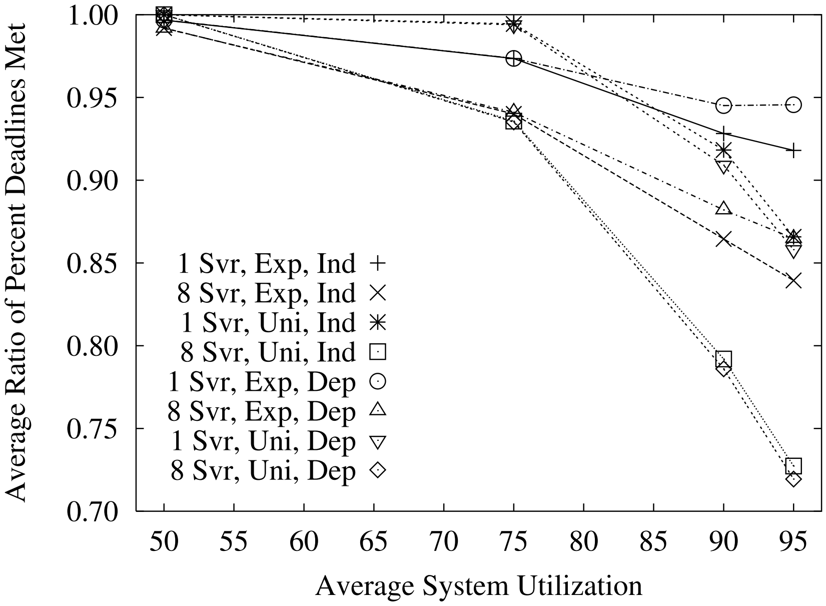
Figures 5.10, 5.11, 5.12, and 5.12 compare the performance of the OSM with the performance of the baseline algorithms. As Figure 5.10(a) shows, significantly more deadlines were missed by OSM-DM from the system's perspective. The Liu and Layland bound for the baseline algorithm is 72.4% and hence the systems are not likely to be schedulable at high average system utilizations. In spite this, we present results for higher utilizations as an indication of critical soft real-time behavior at high system loads.

Figure 5.10(b) indicates that OSM-DM yields better response times than classical DM for a single overrun server and dependent exponential workloads. In all cases, a single overrun server for all tasks performs better than having an overrun server per task. This occurs because interactions between servers due to their fixed priorities prevents them from optimally reclaiming time allocated to idle servers even though the aggressive implementation of the SS employed does make some use of background time.

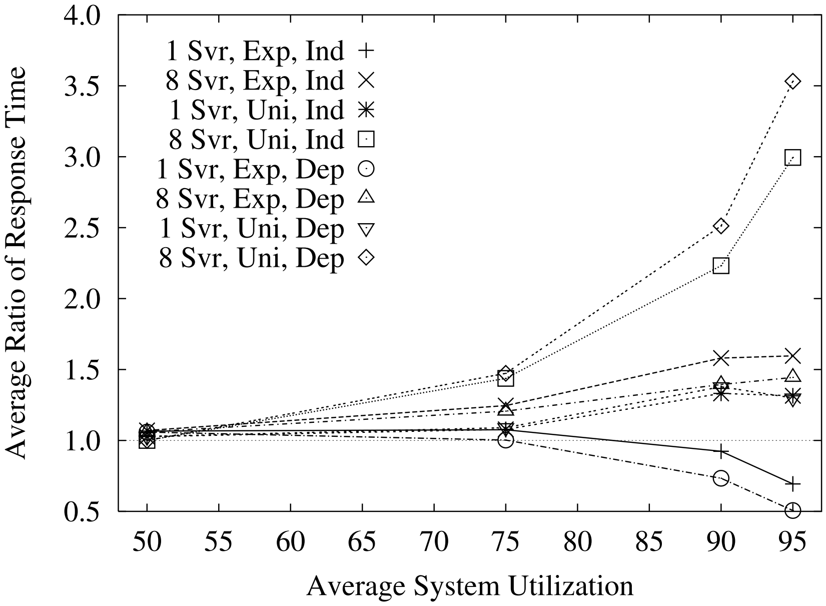
From the tasks' perspective, OSM-DM meets more of the deadlines than DM, as shown in Figure 5.11, particularly for exponential execution time distributions, high average utilizations and a single server for all tasks. However, it does so at a dramatic increase in response times above 75% average utilization. The higher response times of OSM-DM, from both the system and task perspectives, are caused by overruns being executed at a fixed priority which is likely to be different from their natural priority. The effect of servers having fixed priorities is not as apparent from the system perspective because high priority tasks have both better average response times and greater influence on the final results due to more frequent releases of jobs.

In Figure 5.12 we see that OSM-EDF performs better than classical EDF for exponential workloads, particularly if the workload has dependencies. Also it performs better on exponential workloads than on uniform ones because the probability of overrun is less for an exponential distribution than for a uniform one when the guaranteed execution time equals the mean, as it does here. Finally, we observe that OSM-EDF performs better with a server per task than with a single server because a TBS makes use of background time unused by other servers and does so at the priority of the overrunning job.

Curiously, the results from the task perspective (Figure 5.13) are nearly identical to those from the perspective of the system. This is not as strange as it first

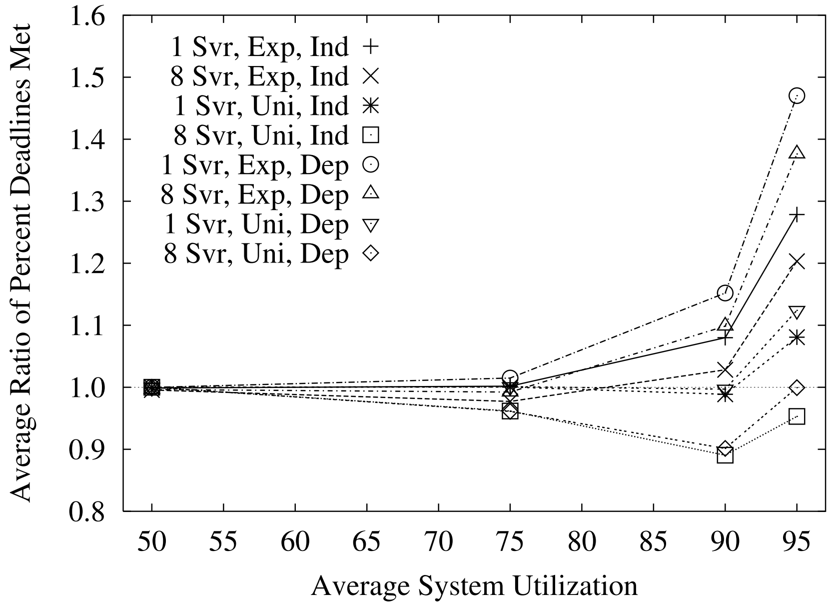


(a) Deadlines Met

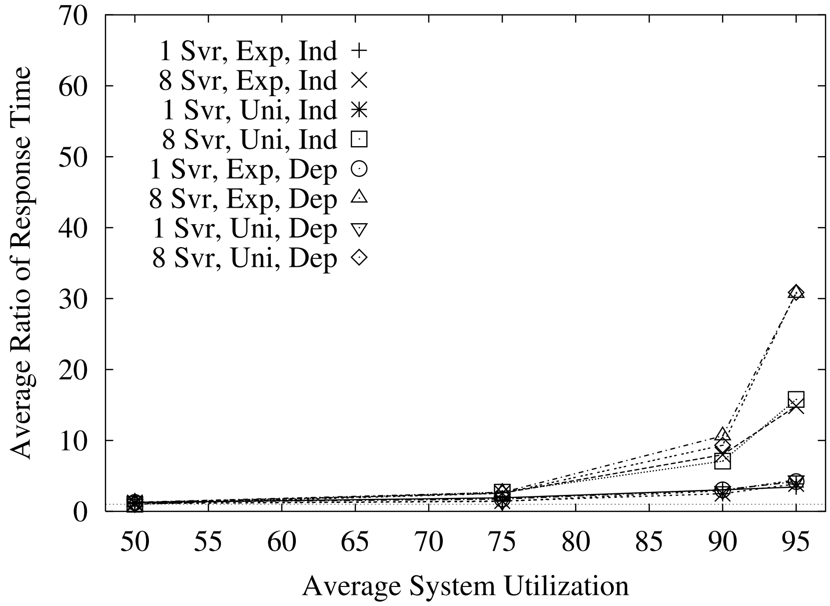


(b) Response Time

Figure 5.10: OSM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to DM.

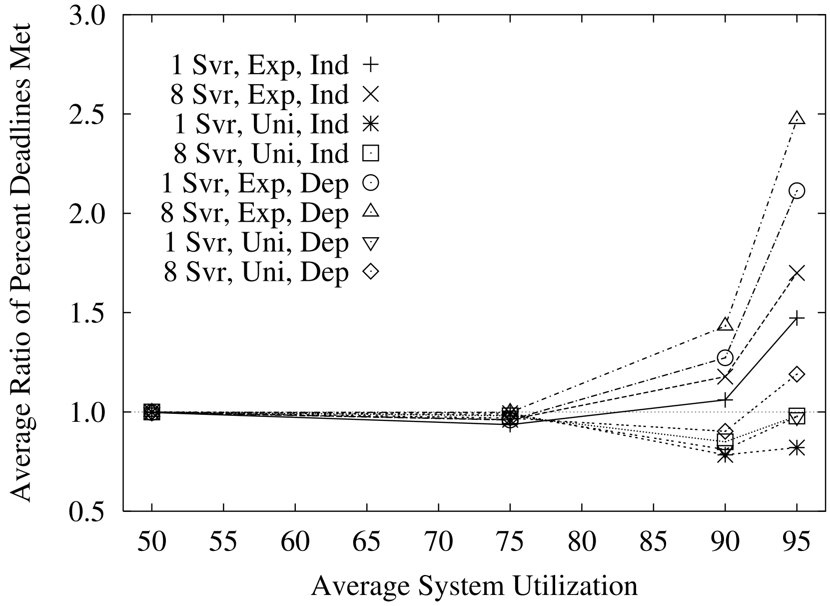


(a) Deadlines Met

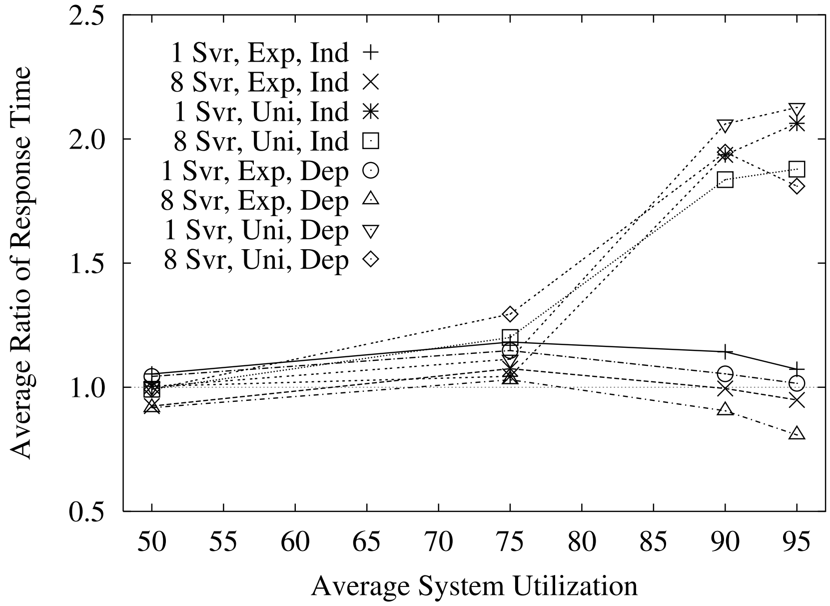


(b) Response Time

Figure 5.11: OSM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 5.12: OSM-EDF vs. EDF from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to EDF.

appears. When a system is scheduled EDF, there is no way to predict which jobs will be affected by an overrun. If the system is overloaded and many jobs overrun, each task is just as likely to miss jobs as any other and in about the same proportion. For example, the EDF results for exponentially distributed execution times at 95% average utilization indicate that 95% of the deadlines met lie within $\pm 4\%$ of the mean from both the system and the task perspectives. The spread is even less for uniformly distributed execution times or for systems with lower average utilizations. Since an overrun server behaves like a periodic task, the performance of the tasks in the system, including the servers, are similar. Therefore, because the response times of the tasks with the greatest release rates do not differ appreciably from the mean, the performance from the system and task perspectives are similar.

5.4.5 Performance of OSM-EDF vs. OSM-DM and DM

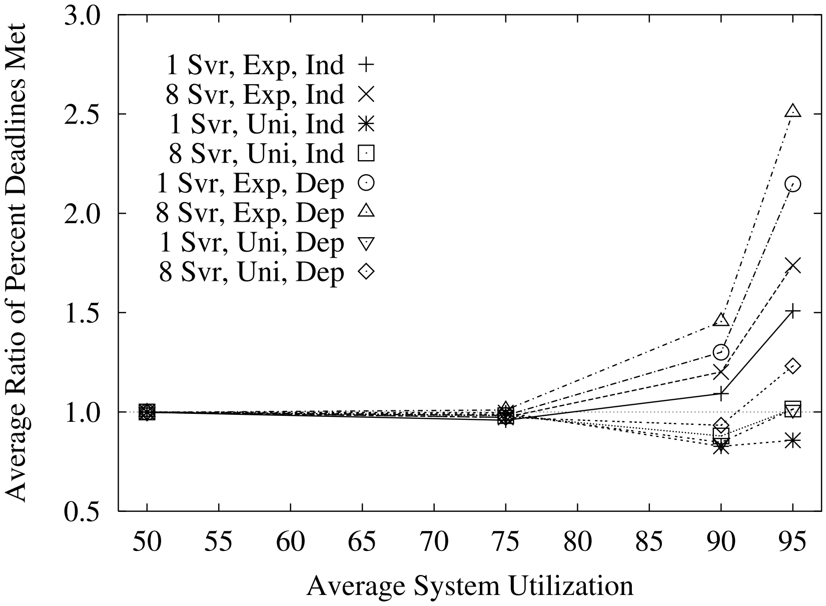
Under classical scheduling, DM performs better than EDF when jobs overrun. However, as Figures 5.14 and 5.15 show, OSM-EDF performs better than OSM-DM in the server per task configuration. Thus, in general, OSM should employ a server per task and be scheduled according to the EDF algorithm.

Finally, Figures 5.16 and 5.17 compare OSM-EDF to classical DM. From the system's perspective, the performance of DM is clearly superior while from the tasks' perspective, OSM-EDF yields significantly more deadlines met for the exponential workloads. However, the response time ratios of DM are still better than OSM-EDF. From this we conclude that OSM-EDF may be preferable only for highly loaded systems containing overrunning jobs that have long execution time tails where percent deadlines met is the primary metric.

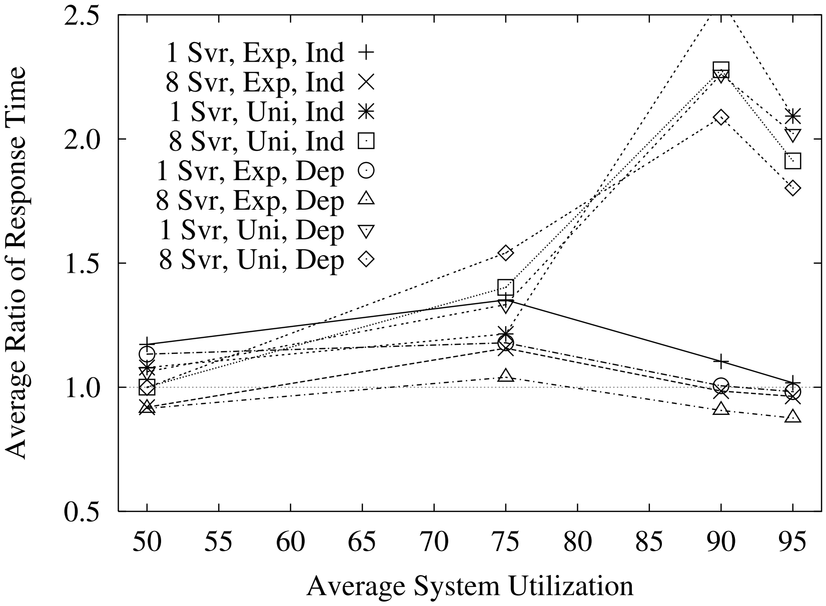
In spite of its generally lack-luster performance from the system perspective when compared with DM, OSM is ideal for systems in which hard, soft and non-real-time workloads execute on the same processor because of the guarantee that non-overrunning jobs will complete in time if the guaranteed portion of the jobs, along with the overrun servers, are schedulable.

5.5 Isolation Server Method

Another way to schedule jobs with the potential for overrun is to release the jobs as requests to isolation servers for execution. The same factors need to be considered

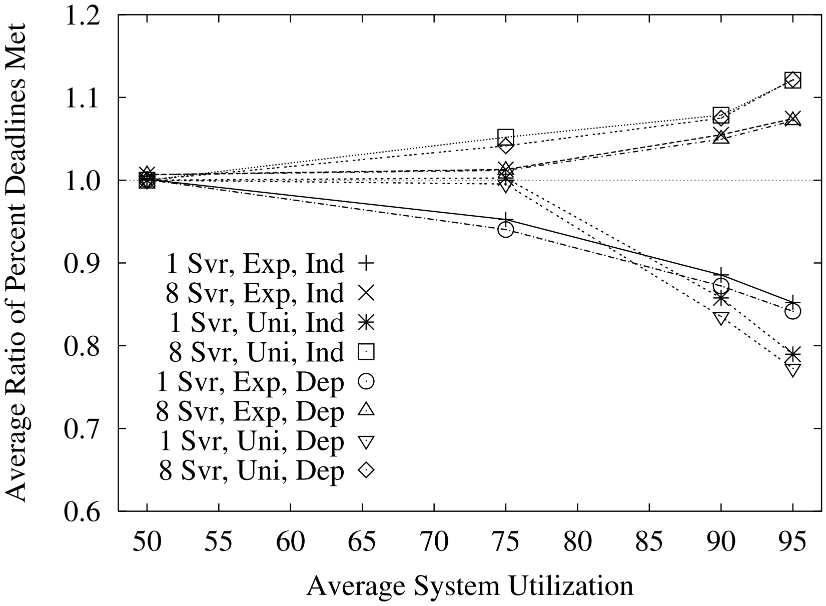


(a) Deadlines Met

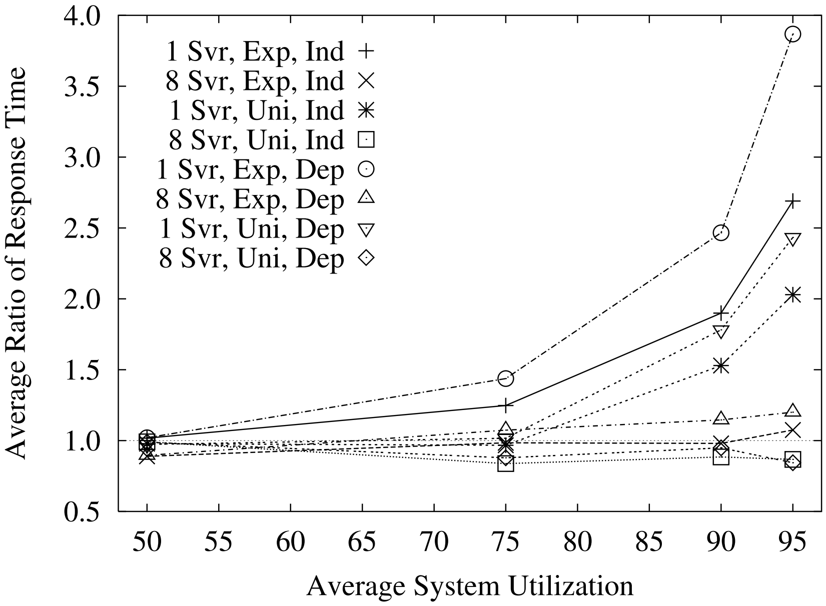


(b) Response Time

Figure 5.13: OSM-EDF vs. EDF from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to EDF.

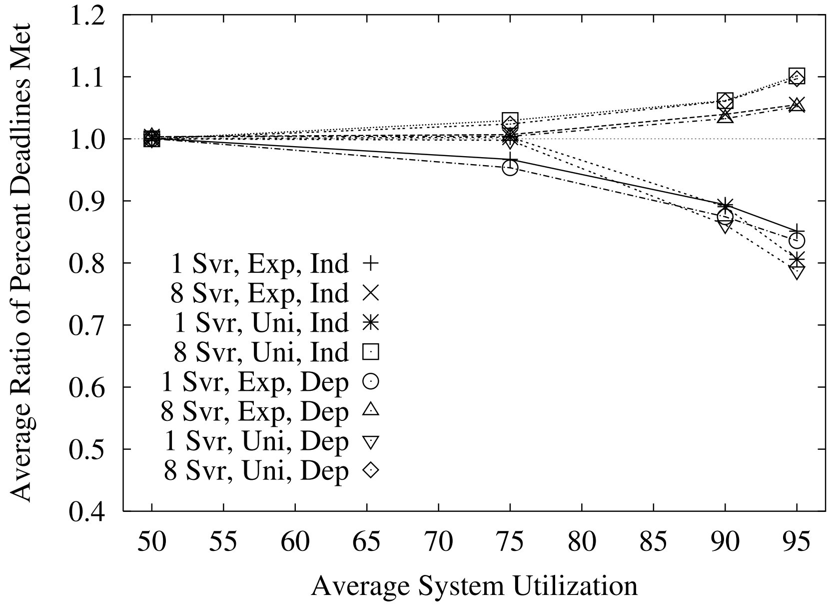


(a) Deadlines Met

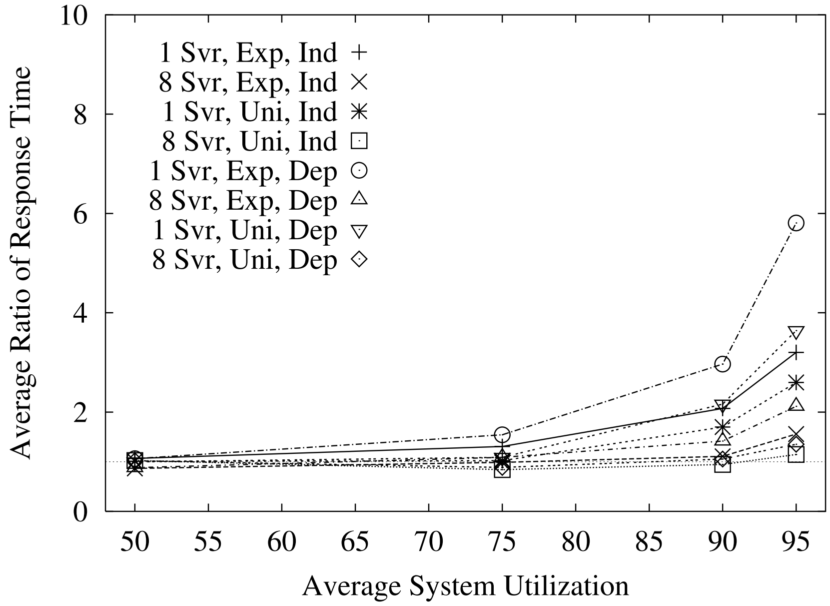


(b) Response Time

Figure 5.14: OSM-EDF vs. OSM-DM from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to OSM-DM.

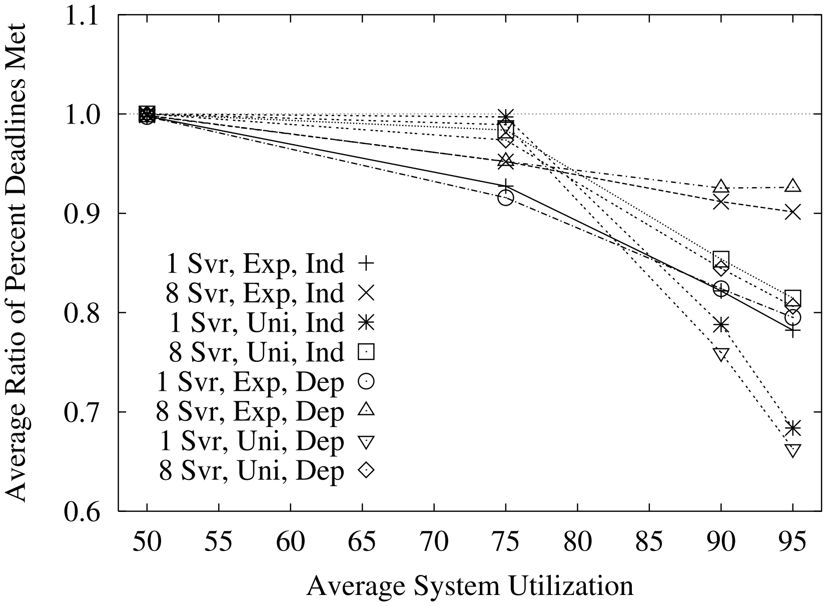


(a) Deadlines Met

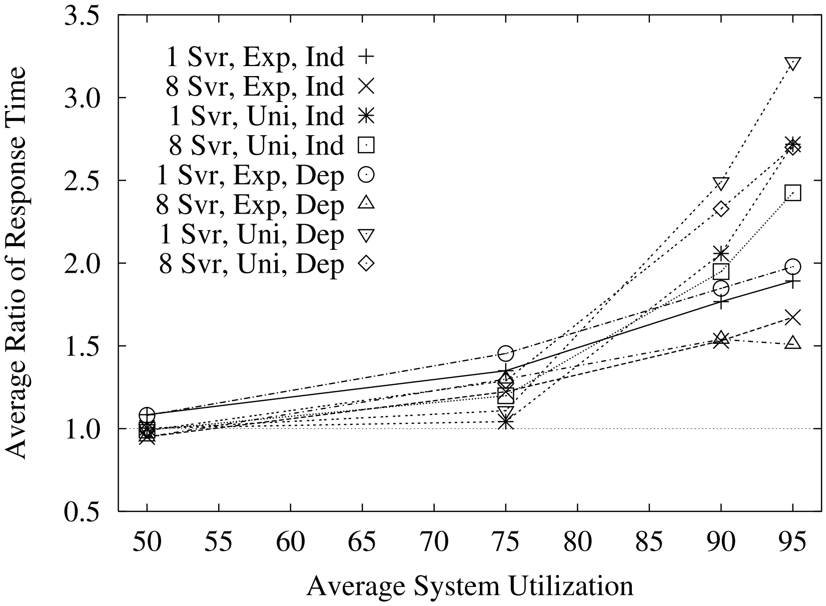


(b) Response Time

Figure 5.15: OSM-EDF vs. OSM-DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to OSM-DM.

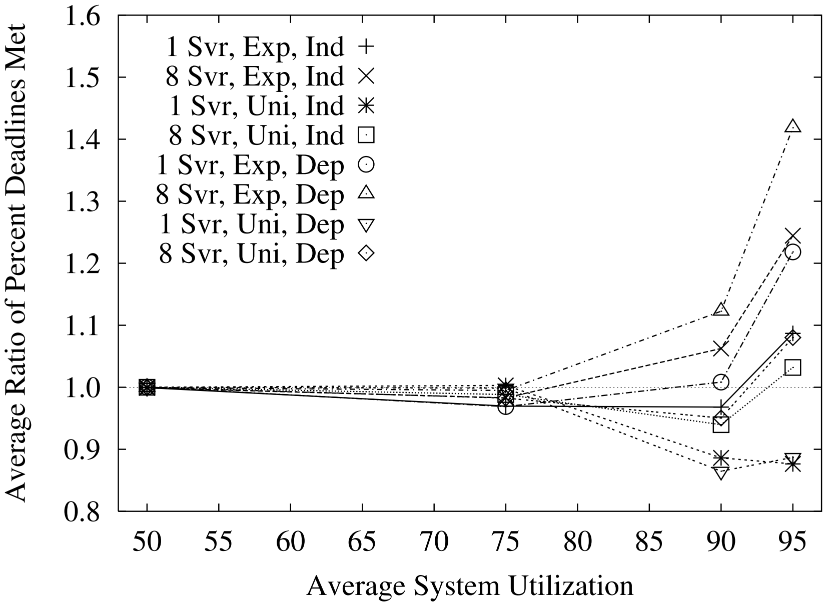


(a) Deadlines Met

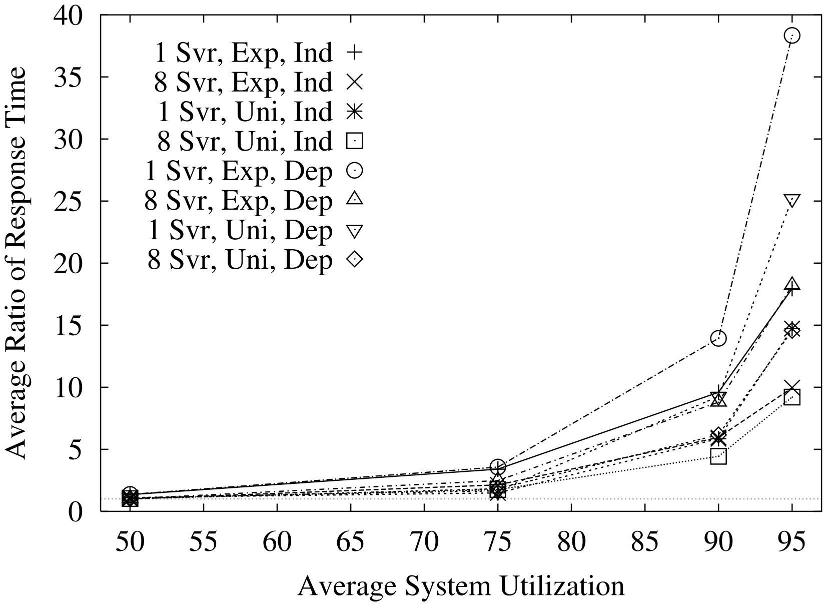


(b) Response Time

Figure 5.16: OSM-EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 5.17: OSM-EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to DM.

when scheduling jobs using the ISM as when scheduling overruns using the OSM: the number of servers, the assignment of jobs to servers, the budget of each server, and the queueing discipline employed by each server.

As in Section 5.4, we consider 1–8 servers. Unlike the OSM, each server is allocated a portion of the processor bandwidth proportional to the fraction of the total utilization the tasks it executes contribute. As is the case for OSM-DM, the SRO queue discipline gives the best results. A TBS is also clearly superior to a CUS for ISM-EDF. Here also, the performance of a single server for all tasks and a server per task bounds the performance of intermediate numbers of servers and hence only the results for 1 and 8 servers are presented.

5.5.1 Performance of ISM vs. Baseline

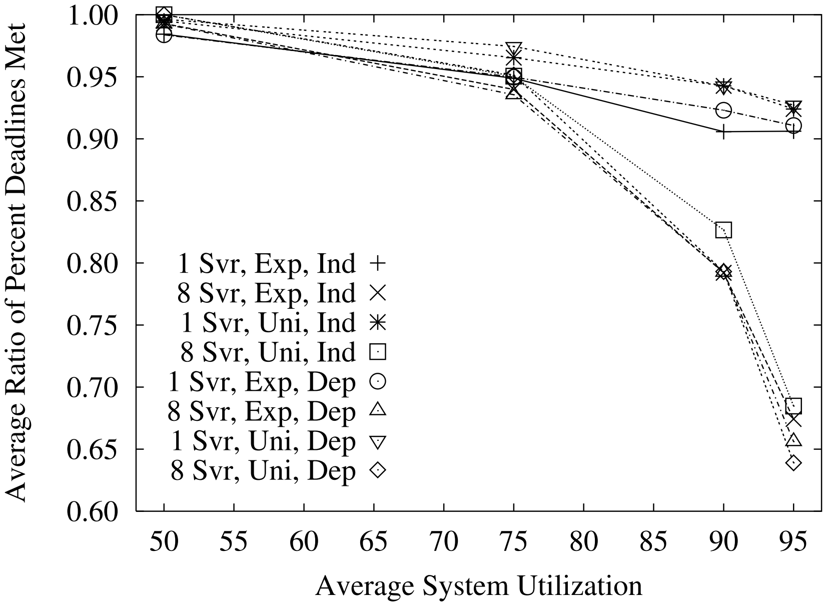
Figures 5.18, 5.19, 5.20 and 5.21 compare the performance of the ISM with the performance of the baseline algorithms. As Figure 5.18 shows, the performance of ISM-DM from the system perspective is worse than classical DM; the systems are less likely to be schedulable as the average system utilization increases. From the tasks' perspective, ISM-DM with a single server meets more deadlines than DM at the expense of an increase in the average response time.

We note that ISM-DM, in Figure 5.18(b) and 5.19(b), has a higher average response ratio than OSM-DM (in Figure 5.10) at 50% utilization because overrunning jobs may delay subsequent jobs under ISM where they cannot under OSM. ISM-DM has a better response ratio than OSM-DM at high utilizations, however. Once again, ISM-DM gives the best results with one server for all tasks for the same reason that OSM-DM does.

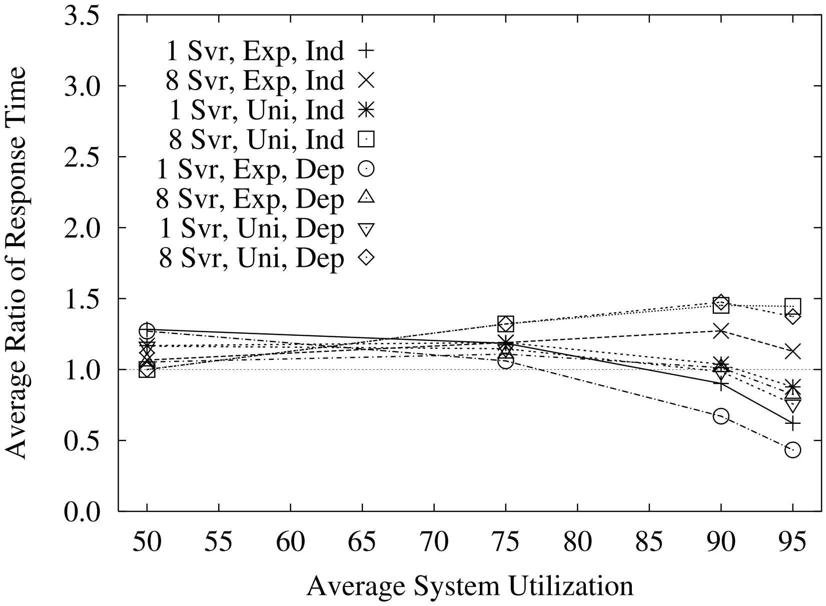
In Figure 5.20(a) we see that ISM-EDF with one server per task meets more of its deadlines on the average than classical EDF and less with a single server for all tasks. Also, the average response time ratio is better for a server per task, as Figure 5.20(b) shows. This behavior is evident for both dependent and independent workloads with exponential or uniform distributions. The conclusions also hold from a task perspective, as shown in Figure 5.21.

5.5.2 Performance of ISM-EDF vs. ISM-DM and DM

As Figure 5.22 shows, the performance of ISM-EDF from the system perspective is better than the performance of ISM-DM in the server per task configuration and

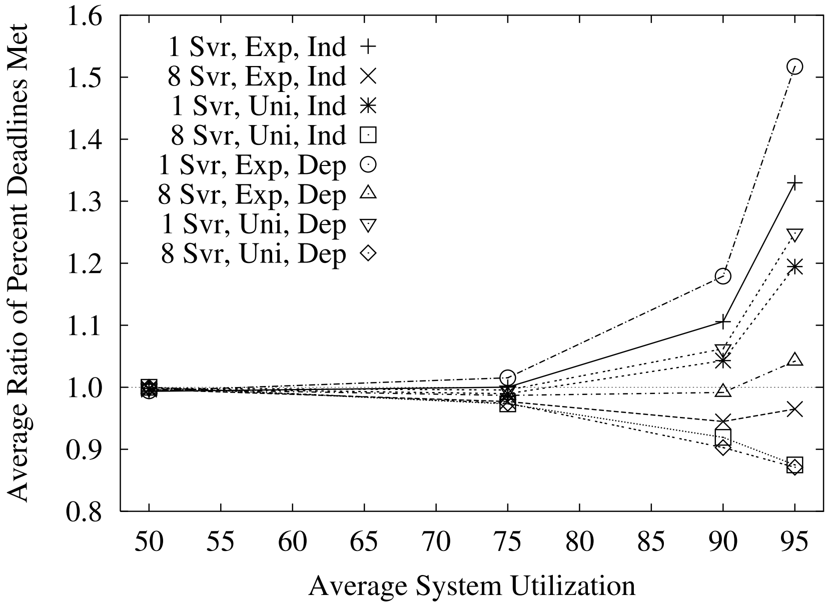


(a) Deadlines Met

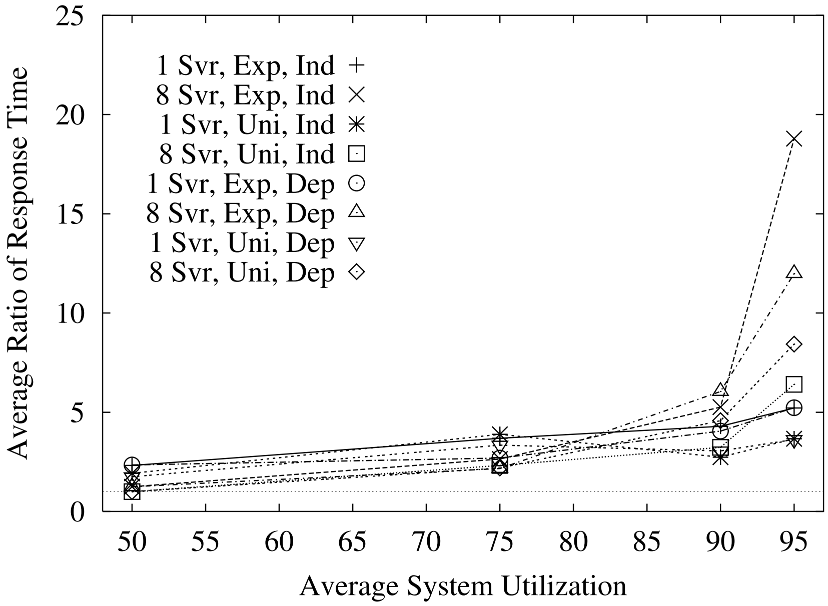


(b) Response Time

Figure 5.18: ISM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

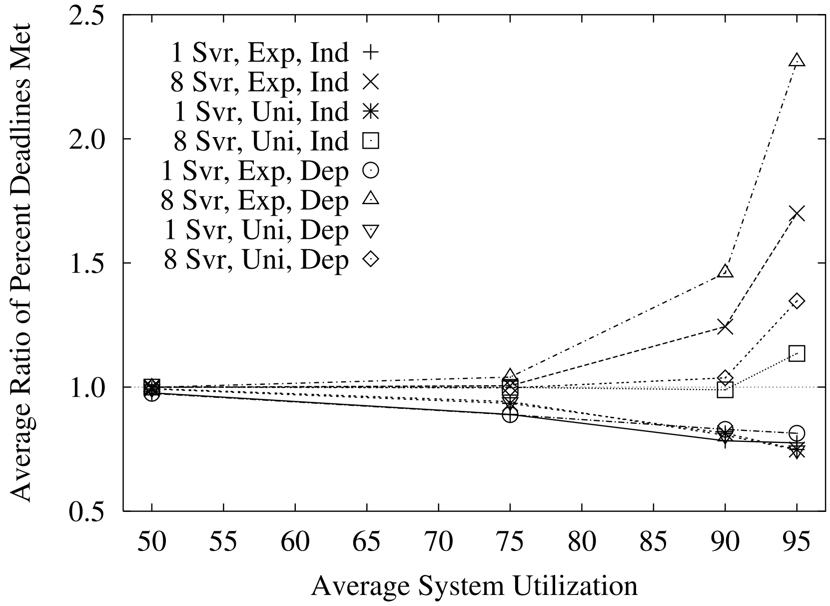


(a) Deadlines Met

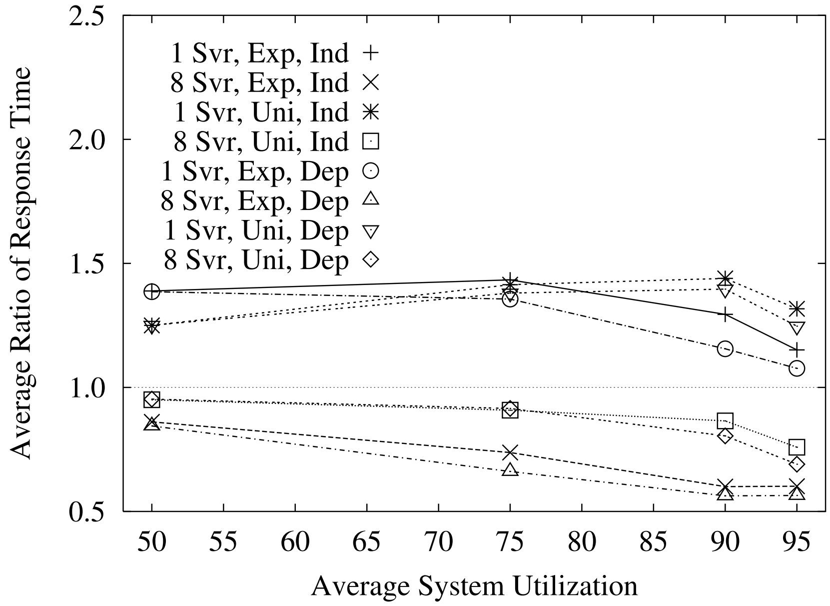


(b) Response Time

Figure 5.19: ISM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

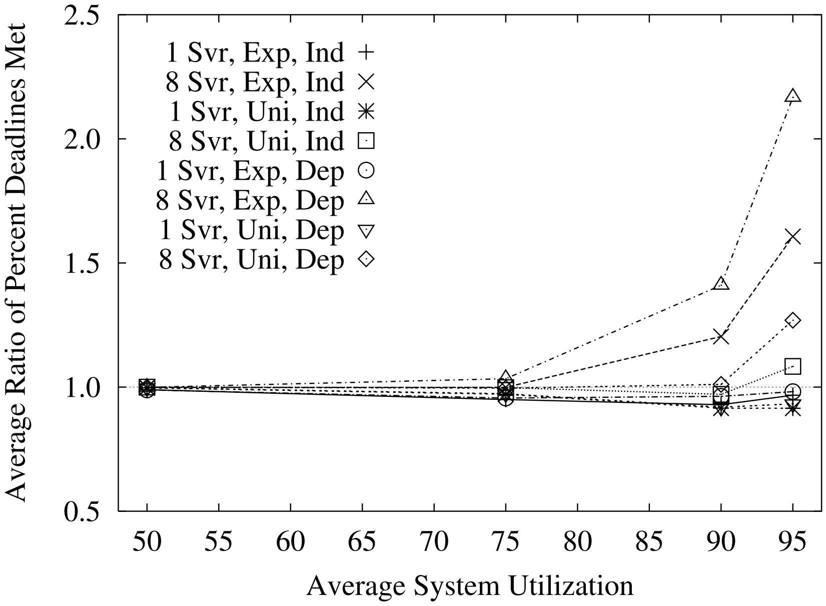


(a) Deadlines Met

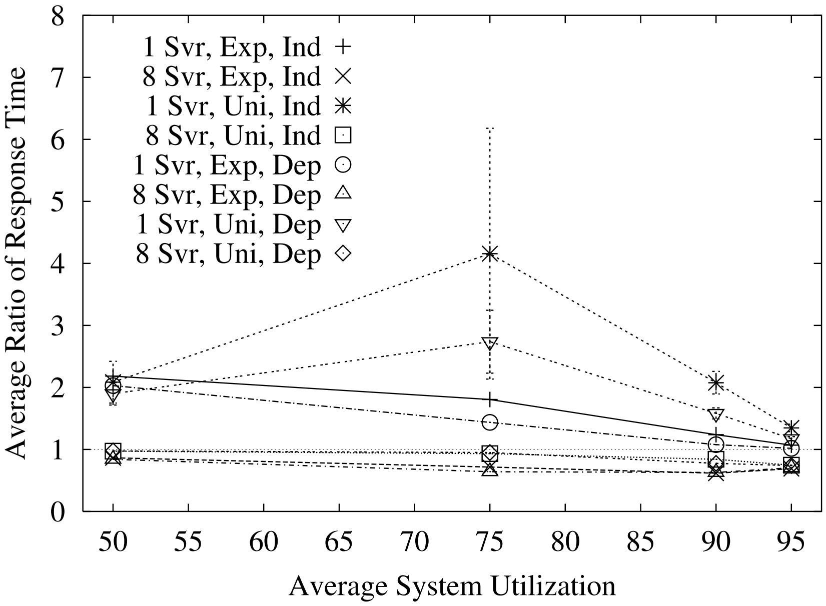


(b) Response Time

Figure 5.20: ISM-EDF vs. EDF from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.



(a) Deadlines Met



(b) Response Time

Figure 5.21: ISM-EDF vs. EDF from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.

worse in the single server configuration. There was surprisingly little variation between independent and dependent workloads. For the best performance, the ISM should employ a server per task and be scheduled according to the EDF algorithm.

Finally, Figures 5.24 and 5.25 compare ISM-EDF to classical DM. From the perspective of the system, the performance of DM is clearly superior than ISM-EDF with a single server, while ISM-EDF and DM are very similar for a server per task. At 95% average utilization, the average ratio of percent deadlines met is lower for ISM-EDF with a server per task than for the DM algorithm by between 4% and 7% for uniform and exponential execution time distributions, respectively. On the other hand, at 50% average utilization the average response time ratio is of ISM-EDF with a server per task is better than that of DM by between 5% and 10%, with exponential execution time distributions yielding the greatest difference.

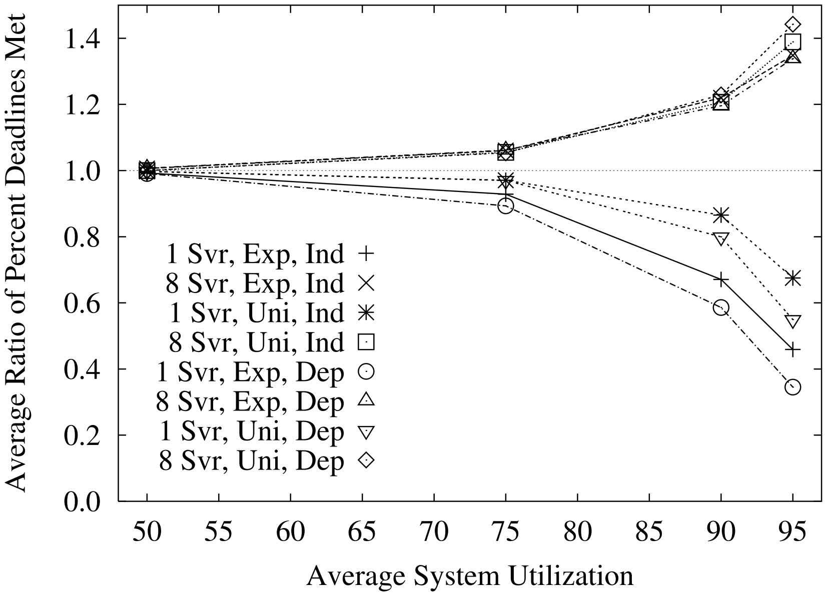
From the task perspective, ISM-EDF with a server per task yields more deadlines met than the DM algorithm for all workloads. However, the average response times of the DM algorithm are significantly better than ISM-EDF. Once again, this is because of the highest priority task has the highest job release rate.

In conclusion, ISM-EDF is generally preferred for most systems, especially those with execution time dependencies or distributions with long tails. The only exceptions are those critical soft real-time systems for which short response times are more critical than deadlines met.

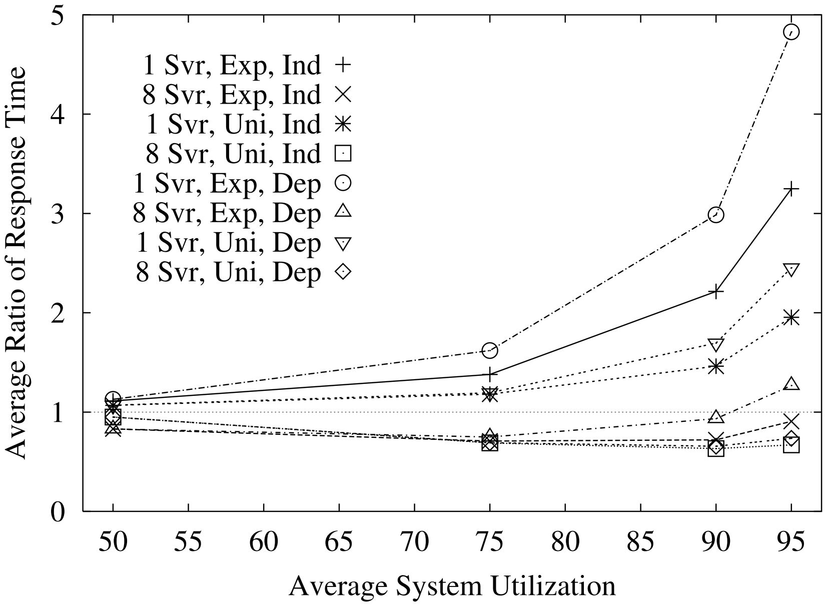
5.6 Realistic Dependencies

The results in the previous section suggest that the ISM-EDF algorithm with a server per task may perform better than the DM algorithm on workloads with dependencies. In this section, we further compare the performance of the two algorithms on workloads with dependencies taken from the execution of jobs of an application. We also compare the performance of classical EDF and DM scheduling to highlight improvements brought about by using the ISM.

The motivating example is a critical soft real-time system for video telephony. In addition to supporting voice communications, the system transmits a video stream of the remote participant for viewing locally. Figure 5.26 shows a block diagram of one of the two end-to-end tasks involved in the video portion of the phone call. There is another end-to-end task transmitting video in the opposite

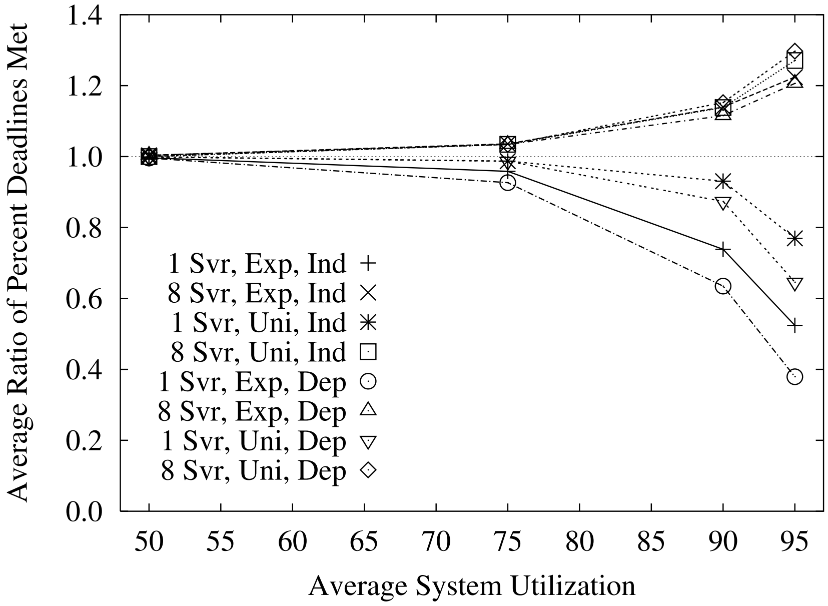


(a) Deadlines Met

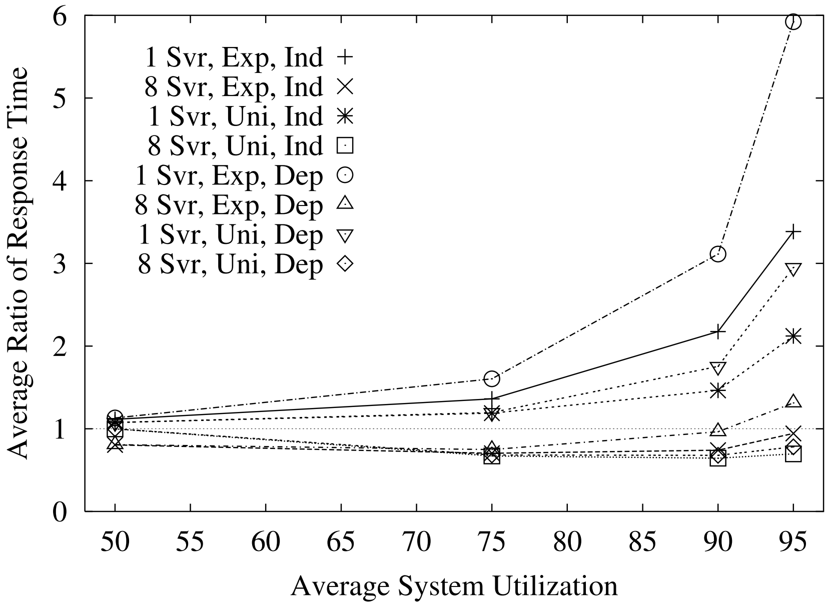


(b) Response Time

Figure 5.22: ISM-EDF vs. ISM-DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

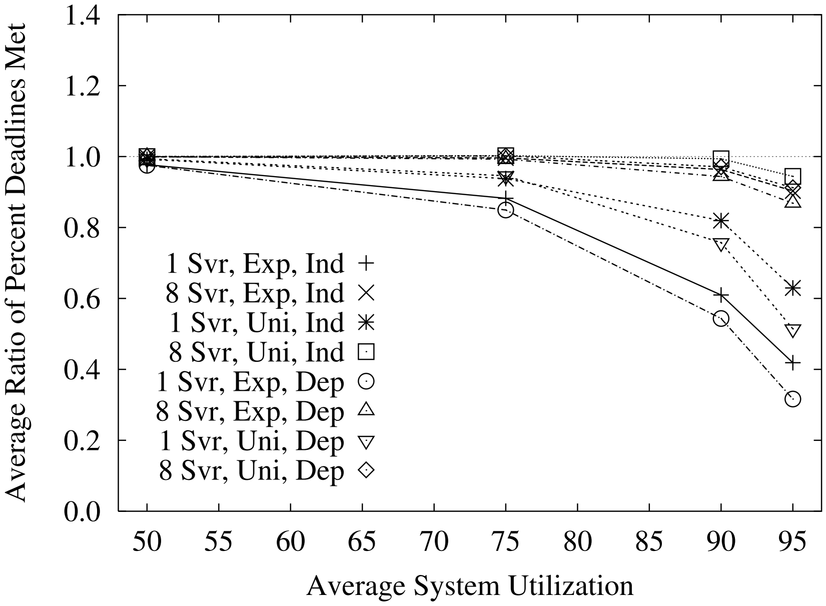


(a) Deadlines Met

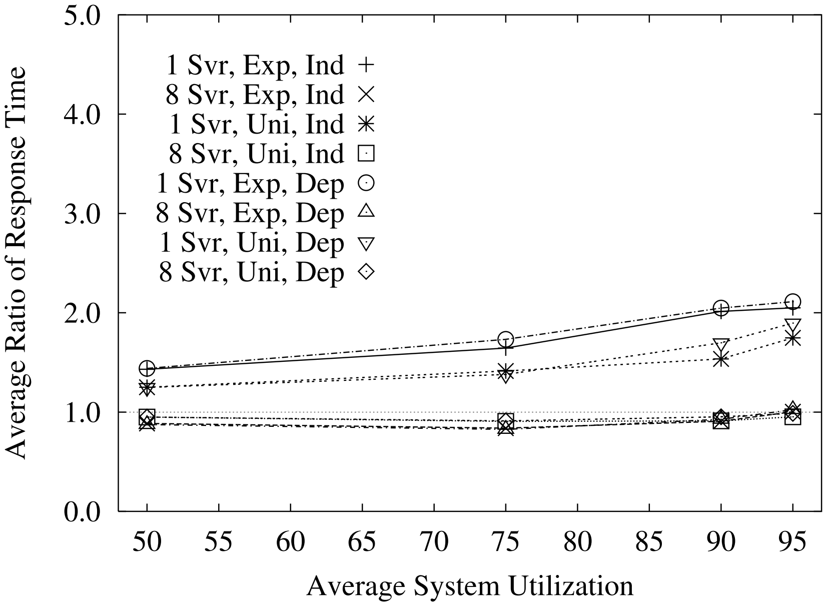


(b) Response Time

Figure 5.23: ISM-EDF vs. ISM-DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

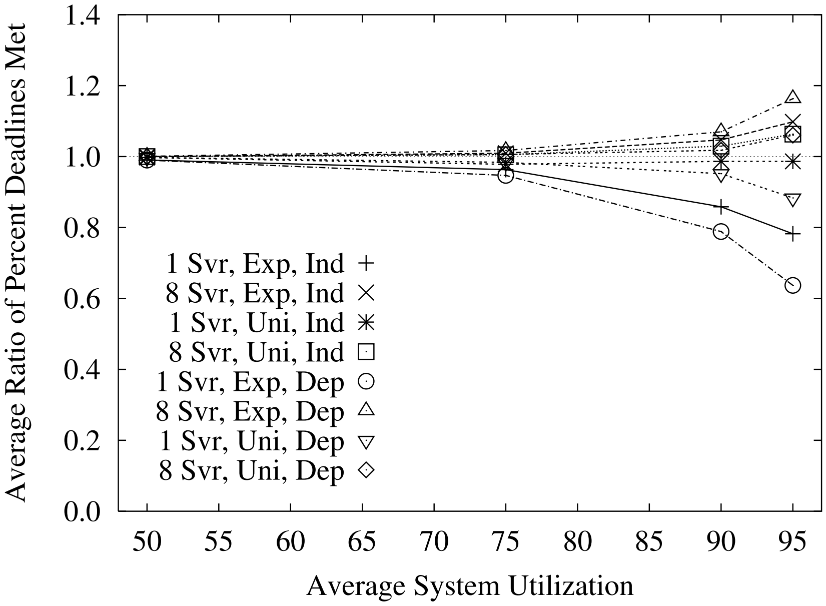


(a) Deadlines Met

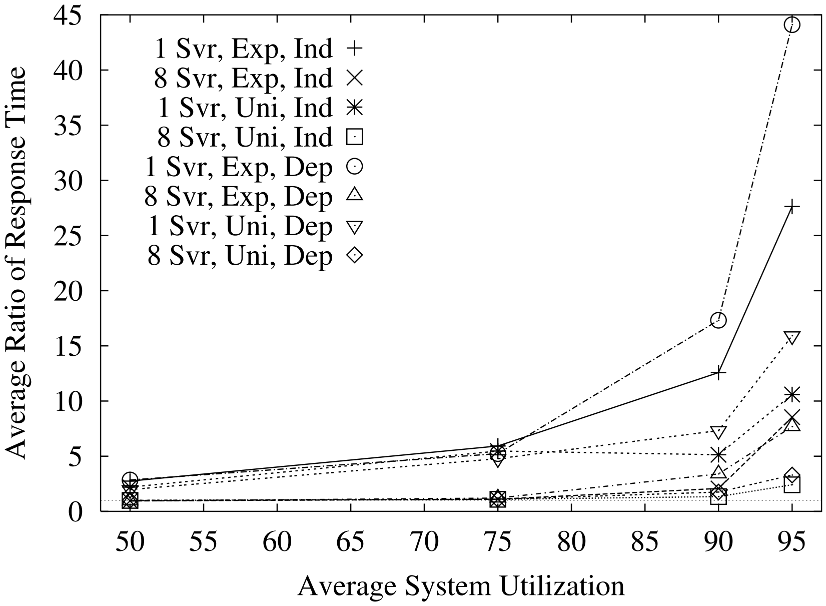


(b) Response Time

Figure 5.24: ISM-EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 5.25: ISM-EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.

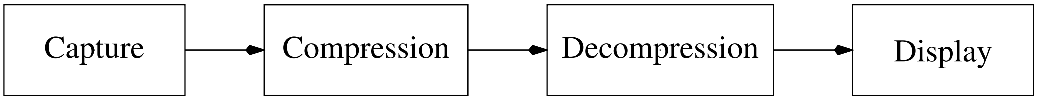


Figure 5.26: Video Telephony End-to-End Task for One Side of a Call.

direction. Each end-to-end task is divided into sub-tasks which capture, compress, transmit, decompress and display frames of the video stream. In order for participants to communicate effectively, the majority of the deadlines needs to be met. We focus on the video decoding (decompression) task which transforms a MPEG-1 encoded video stream into a sequence of video frames to be displayed.

Because of the sophisticated compression algorithm used to produce a MPEG video stream, decode times of frames vary dramatically. Furthermore, successive decode times are correlated due to the way in which a video stream is encoded into I, P, and B frames. Similarities in the content of successive frames of the original video stream also introduce dependencies in the decode times of successive frames in the encoded video stream.

Ideally, we would like to have used video streams from actual teleconferences in comparing the performance of the ISM-EDF and DM algorithms. However, such streams are unavailable, probably because of a lack of demand but also for privacy reasons. Instead, we used readily available MPEG-1 video streams. The first is a segment from an animation entitled “A Close Shave” by Nick Park. The second is a video about the dreams of a unicycle entitled “Red’s Nightmare”. The final stream is a simulated walk through of an imaginary art museum entitled “The Incredible Museum”. Traces of the decode times of frames in the videos were obtained by using a high precision timer to measure the time required to decode each frame. Only decode times were measured. The time consumed reading data from the disk or displaying the decoded frames was not measured.

One reason we chose these streams was their length; they all contain over 1000 frames. We also chose the streams for their wide variations in decode times. (A more detailed description of the characteristics of the streams can be found in Appendix B along with more information on the process used to obtain the traces.) We used the traces of decode times to generate the workloads used in comparing the performance of the two algorithms.

5.6.1 Workload Generation

As in Section 5.2.2, the average utilizations of the workloads are 0.50, 0.75, 0.90 and 0.95. The average utilization of each task was chosen randomly as before. The execution times of jobs in each task were taken from the traces of decode times. Assuming 30 frames per second (fps), the average utilizations of the traces ranged from 8% to 13%. To obtain the desired average system utilization, either the frame rate of the trace can be increased or the decode times scaled. Since frame rates higher than 30 fps generally do not yield noticeable improvements in quality, we chose to scale the decode times of the traces.

Workloads consisted of 100 systems containing 4 tasks. The execution times of all tasks in a system were taken from the same trace. Execution times for the "control", or independent group, were selected uniformly from the trace. Execution times for the dependent workloads were selected sequentially from the traces with the start of a sequence of decode times chosen uniformly from the range of frame numbers. Selections of decode times were wrapped around to the beginning upon reaching the end of the trace. The minimum number of jobs in a task varied from 1000 at 50% utilization to 4000 at 95% utilization in order to reduce the width of the confidence intervals at high average utilizations. (In general, the width of the 95% confidence intervals were less than $\pm 5\%$.)

We note that as an artifact of selecting values from traces rather than from uniform or exponential distributions, the average utilization of a workload was occasionally higher or lower than desired because the coefficient of variation of the traces was less than 1.0. The frequency of this occurring was quite low. It occurred three times for the "Incredible Museum" video: twice at 50% average utilization with dependencies and once for 90% utilization without dependencies. It did not occur for the other two video streams. To avoid the possibility of introducing a bias, the three configurations where this occurred were disregarded in the final results (even though the results would likely not have been significantly affected).

5.6.2 Performance

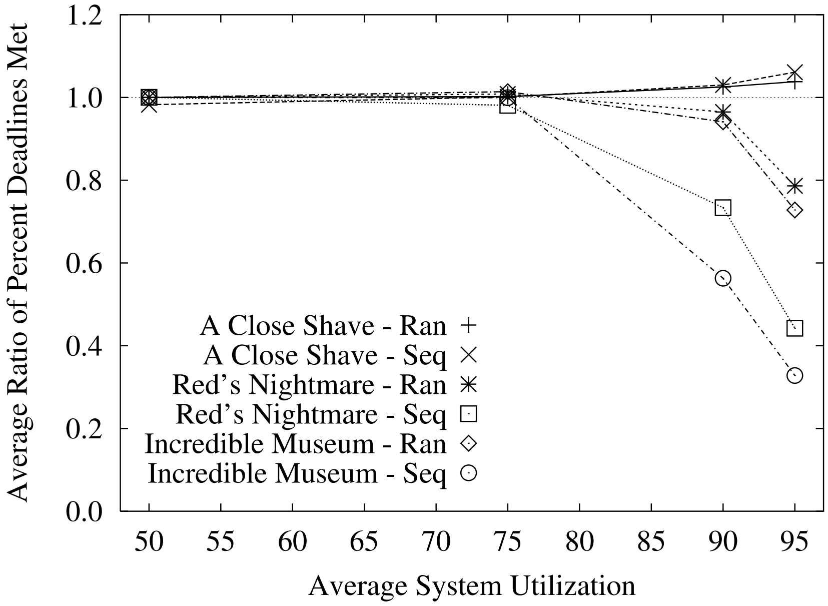
A comparison between the classical EDF and DM algorithms yields no surprises. From the system perspective, the DM algorithm generally performs better than the EDF algorithm, as Figure 5.27 shows. The only exception was the random and sequentially selected workloads from the video "A Close Shave" where the EDF algorithm performed better from the system perspective than the DM algorithm at 90% or higher average utilizations. From the task perspective, Figure 5.28, the EDF algorithm met more deadlines than the DM algorithm for all streams except “Red’s Nightmare” and “Incredible Museum” with dependencies. The response times were generally worse, however.

Figures 5.29 and 5.30 compare the performance of ISM-EDF with a server per task and classical DM. From the system perspective, DM met more deadlines than ISM-EDF. However, ISM-EDF met more deadlines than DM from the task perspective. The response times for the algorithms were exactly opposite. ISM-EDF generally had better response times from a system perspective and worse response times from a task perspective.

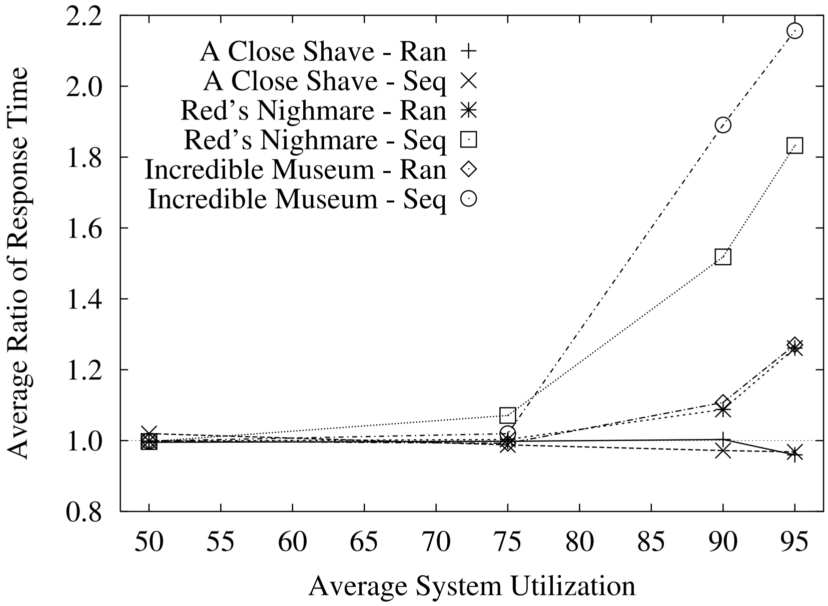
5.7 Summary

In summary, the primary metrics by which to compare algorithms for scheduling critical soft real-time systems in the presence of overrun are the percentage of deadlines met and the average response time. The metrics can be computed from the perspective of the system or of the tasks in the system. The former is useful for considering overall system performance, while the latter is important for designers of critical soft real-time systems.

In this chapter, we proposed two classes of scheduling algorithms, the Overrun Server Method and the Isolation Server Method. We also compared their performance through simulation with classical real-time scheduling algorithms exemplified by the Deadline Monotonic and Earliest Deadline First algorithms. We observed that the performance of the Overrun Server Method was the highest when there was an overrun server per task scheduled by a deadline-driven scheduler. In spite of this, the performance of the Overrun Server Method is generally inferior to classical DM from both the system and task perspective with the exception that OSM-EDF met more deadlines than the Deadline Monotonic scheduling algorithm from the task perspective. However, unlike the classical scheduling algorithms, the Overrun Server Method guarantees that jobs which do not overrun will meet their deadlines if the tasks of the system and the overrun servers are schedulable on the basis of the guaranteed execution times and periods. Thus it is ideal for systems in which hard, soft and non-real-time workloads execute on the same processor.

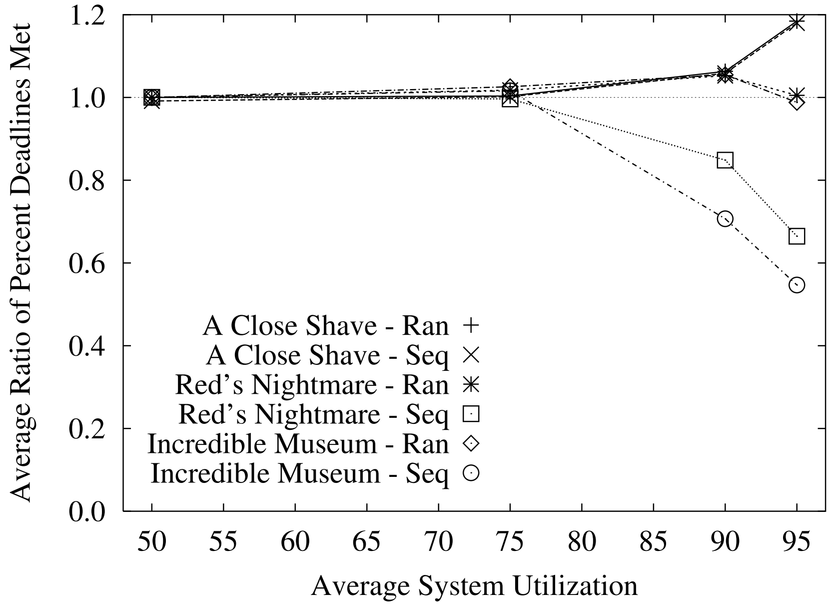


(a) Deadlines Met

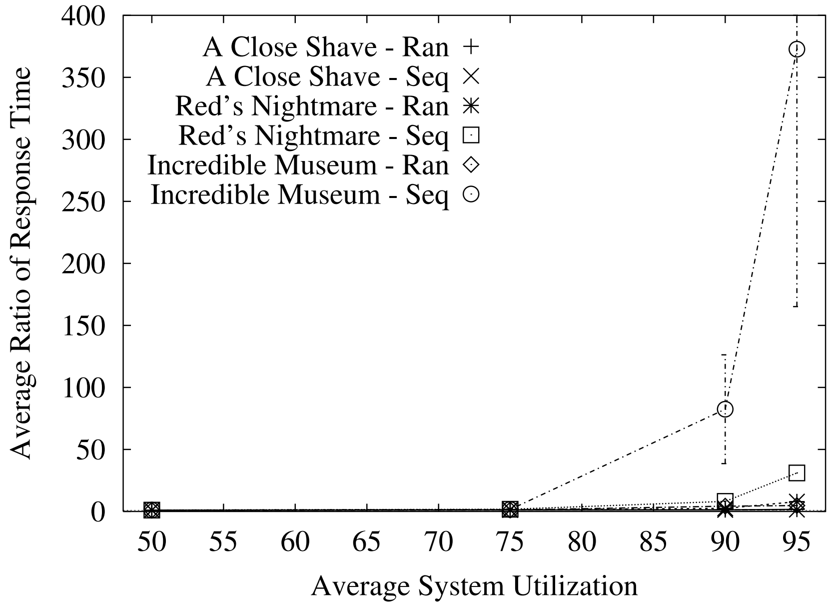


(b) Response Time

Figure 5.27: EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according EDF to the metrics for systems scheduled according to DM on the MPEG workloads.

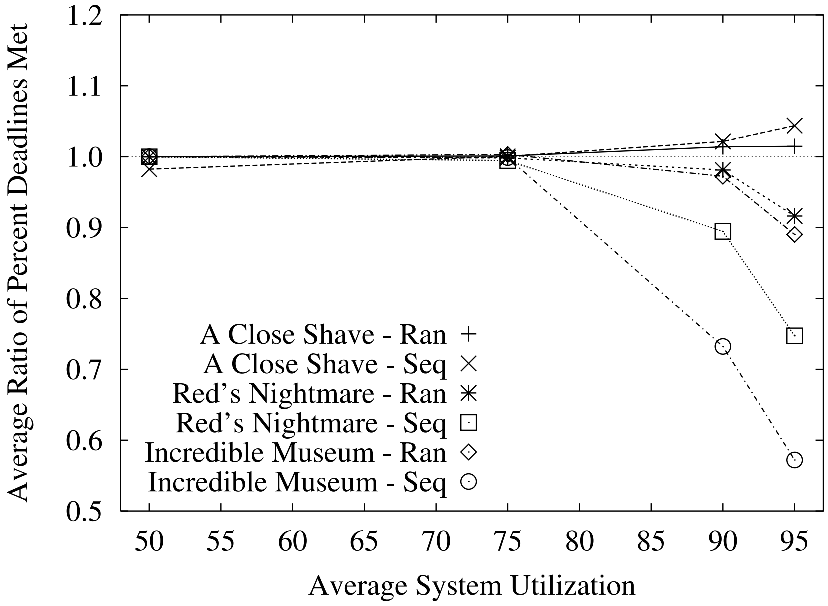


(a) Deadlines Met

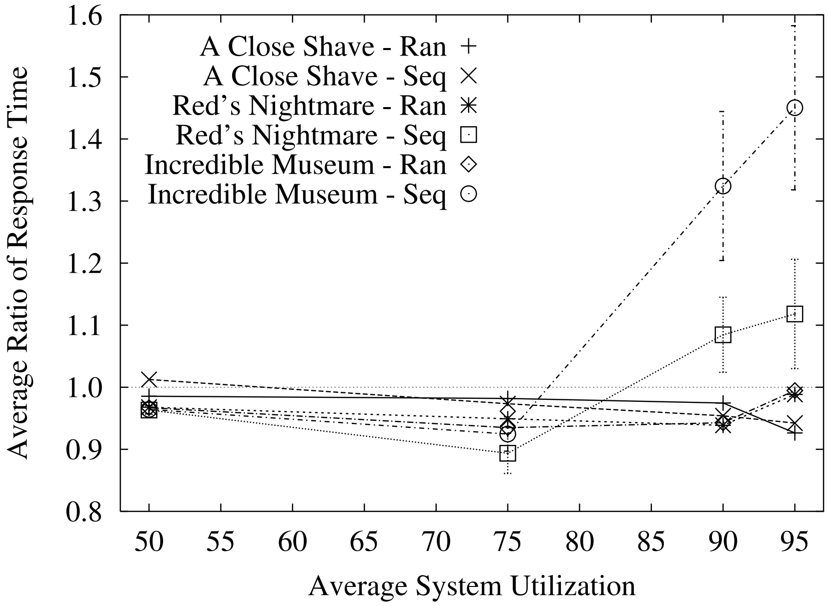


(b) Response Time

Figure 5.28: EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according EDF to the metrics for systems scheduled according to DM on the MPEG workloads.

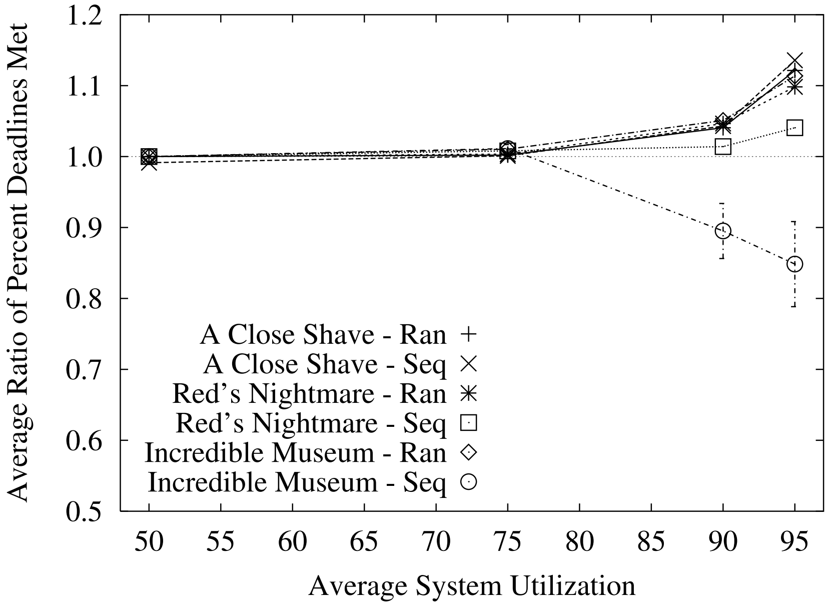


(a) Deadlines Met

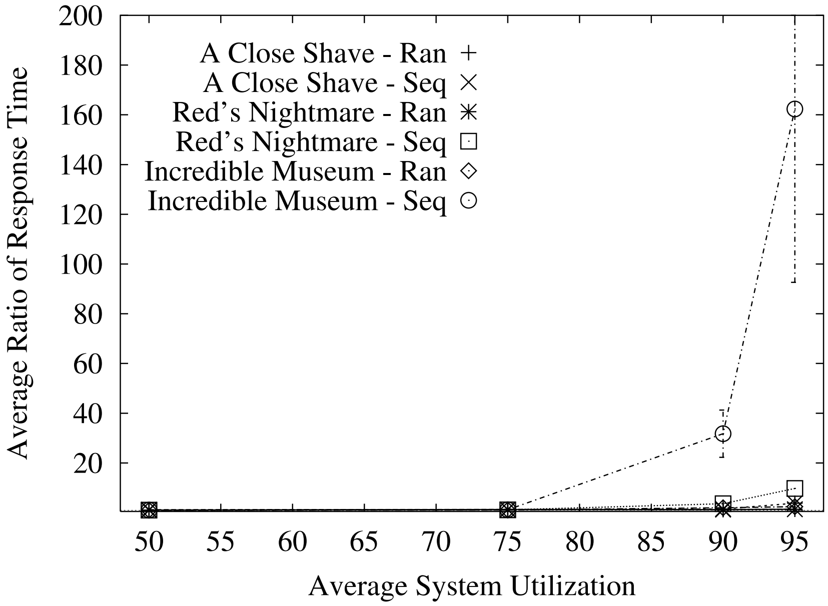


(b) Response Time

Figure 5.29: ISM-EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM on the MPEG workloads.



(a) Deadlines Met



(b) Response Time

Figure 5.30: ISM-EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM on the MPEG workloads.

The performance of the Isolation Server Method is also best when there is a server per task scheduled by a deadline-driven scheduler. From both the system and task perspectives, the performance was generally comparable to the best classical algorithm on workloads with independent execution times; the classical algorithms having the advantage on one metric from each perspective and the Isolation Server Method having the advantage on the remainder. Because the causes of overrun are often correlated, dependencies arise between the execution times of jobs in a task. For the dependent workloads considered, ISM-EDF with a server per task performed better than the classical algorithms, especially from the perspective of the tasks in the system, at the cost of increased average response time.

The results indicate that all the methods perform similarly for average system utilizations below about 75% regardless of the distribution of execution times and the presence or lack of dependencies. (A 75% average system utilization corresponds to a maximum system utilization of at least 150% for the distributions considered.) Thus, the choice of scheduling algorithm is probably not crucial for all but the most highly overloaded systems.

Although we have explicitly considered the performance of static systems in this study, the results are also applicable to systems in which tasks arrive and leave. Through the use of admission control to manage the average system utilization, non-overrunning jobs are assured of meeting their deadlines when scheduled according to the Overrun Server Method. This allows hard, soft and non-real-time workloads to be executed on a single processor while making real-time guarantees appropriate to each class of traffic and is done without dividing bandwidth into fixed partitions assigned to the various types of workloads. The distinction between hard, soft and non-real-time workloads is made by the choice of guaranteed parameters and enforced by the scheduling algorithm. Any allocated bandwidth not needed by a server is automatically given to a server which can use it. Under OSM-EDF and ISM-EDF, the spare bandwidth is given to the server which will most benefit from receiving additional time.

Chapter 6

Scheduling Jittered Releases

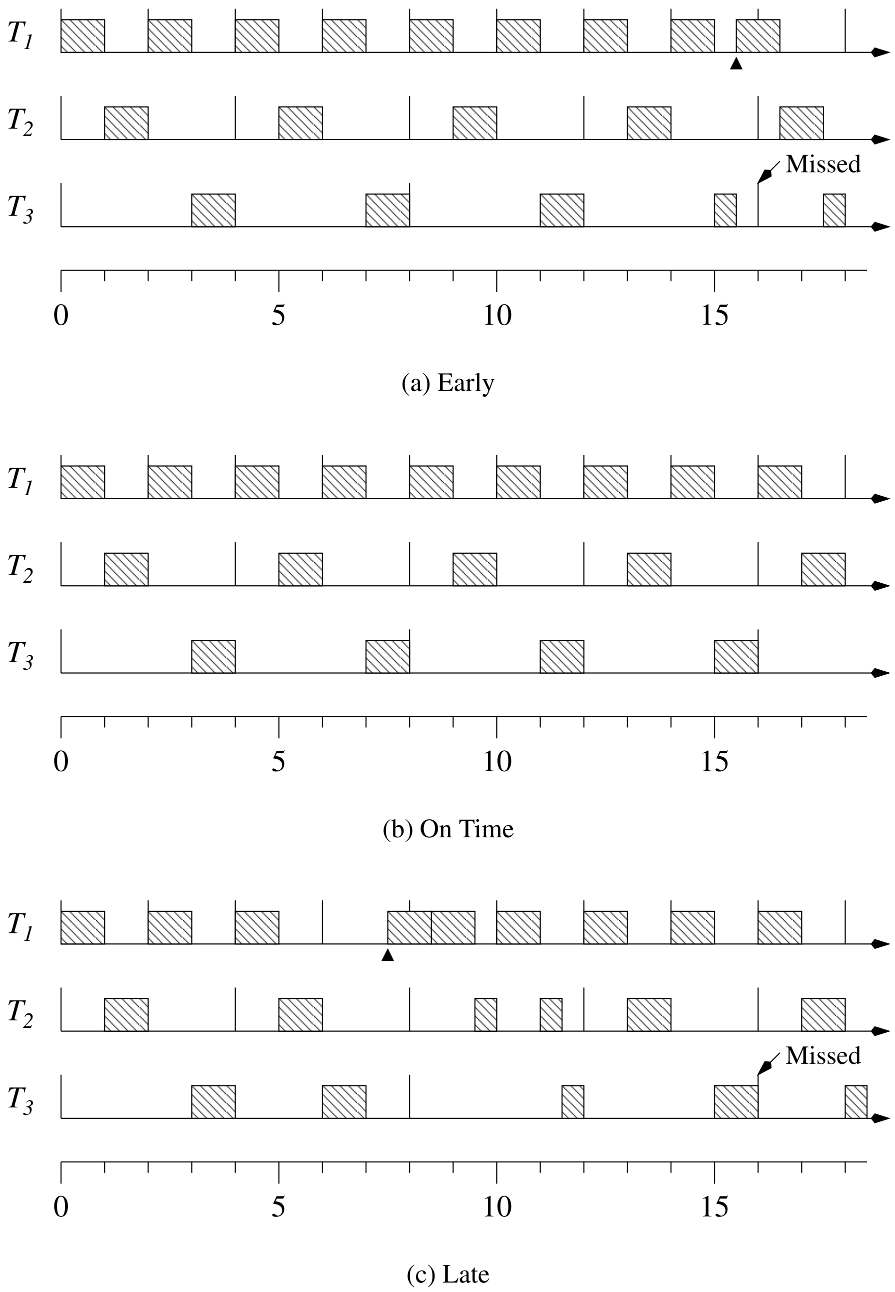
In Chapter 5, we considered the scheduling of systems in which the execution times of the tasks varied. We now turn our attention to systems in which the inter-release times of jobs also vary. We initially consider the case where the execution times of all jobs are equal to the guaranteed execution times of their tasks. We then consider the more general case where both the execution times and inter-release times of jobs are variable.

6.1 Effect of Release Time Jitter

A system in which jobs may be released earlier or later than their guaranteed inter-release times would require is said to experience *release time jitter*. Release time jitter can cause jobs to miss deadlines or have increase response times because it can cause short-term overload in a system which would not otherwise be overloaded.

As an example, consider a system of three tasks with guaranteed inter-release times of 2, 4, and 8 (and with guaranteed execution times of 1, 1 and 2 respectively). The maximum utilization of the system is 1.0. In the schedules shown in Figures 6.1(b) and 6.2(b), all jobs meet their deadlines when the jobs are released "on time" and the system is scheduled according to the DM or EDF algorithms.

Suppose, however, that job $J_{1,9}$, which is nominally released at time 16, is released 0.5 time units early and scheduled according to the DM algorithm, as shown in Figure 6.1(a). (The release time of the job is indicated by the filled triangle.) Because the priority of T_1 is higher than the priority of T_3, $J_{1,9}$ preempts $J_{3,2}$ causing it to miss its deadline. Similarly, the early release of jobs in a deadline-driven system can cause deadlines to be missed (see Figure 6.2(a)). Now

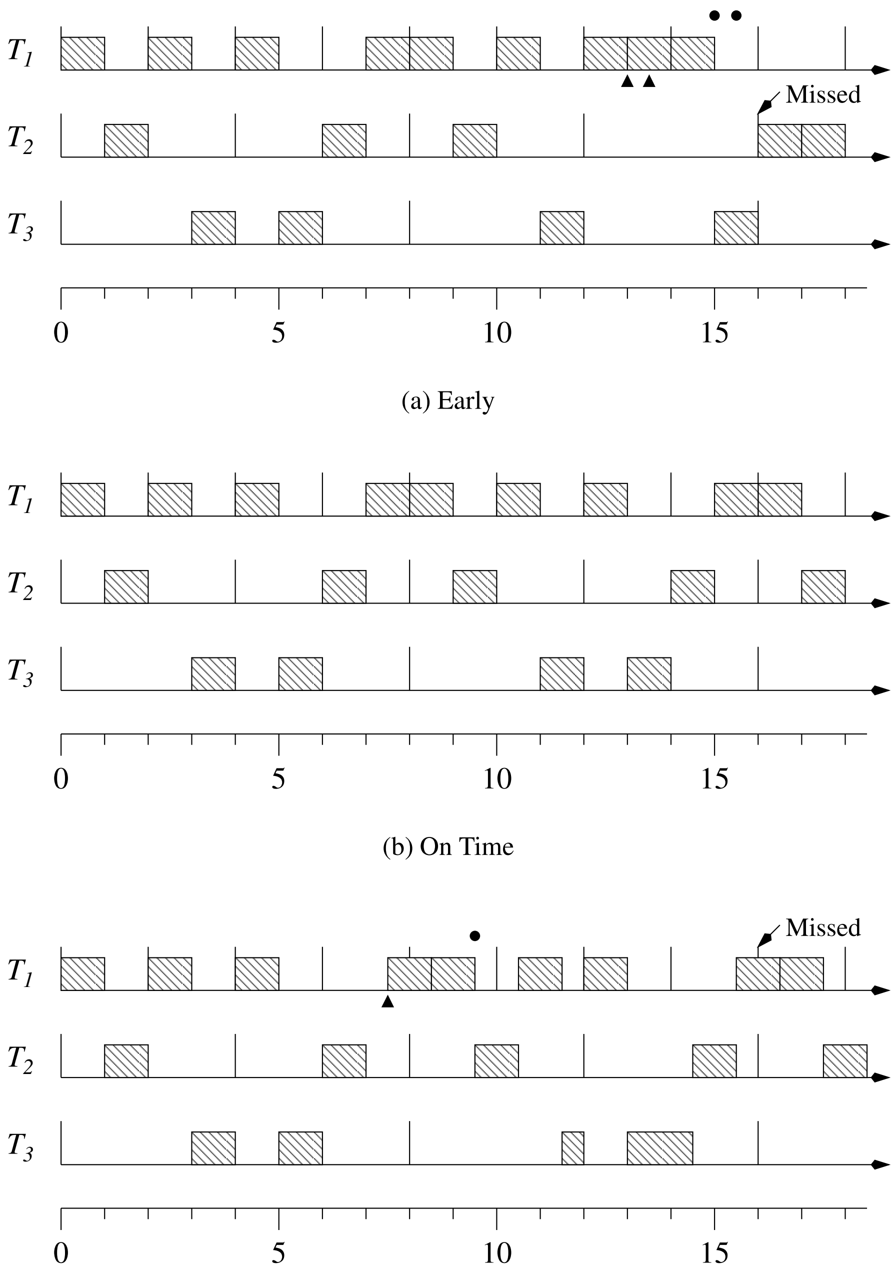


(a) Early

(b) On Time

(c) Late

Figure 6.1: Effect of release time on a fixed-priority system.



(a) Early

(b) On Time

(c) Late

Figure 6.2: Effect of release time on a deadline-driven system.

suppose that job $J_{1,4}$ of T_1, which is nominally released at time 6, is released 1.5 time units late and the subsequent job, $J_{1,5}$ is released on time.[1] (The scheduling deadline of the job is indicated by a filled circle.) The late release of $J_{1,4}$, coupled with the on-time release of its successor, causes job $J_{3,2}$ to miss its deadlines when scheduled according to the DM algorithm, as shown in Figure 6.1(c). Likewise, the delay due to $J_{1,4}$ being released late causes job $J_{1,8}$ to miss its deadline when scheduled according to the EDF algorithm (Figure 6.2(c)).

As the examples show, jittered release times can cause the amount of work which has been released but not yet completed to exceed the capacity of the processor causing a short-term overload to occur even though the system is schedulable on the basis of the guaranteed inter-release times.

6.2 Fairness in Scheduling

One result of variations in release times within a task is the clustering of jobs into bursts of work followed by intervals in which the task is idle. We allow the scheduler to give additional time to backlogged tasks whenever some other task is idle, as long as the idle task is once again given its guaranteed allocation when it releases another job. Ideally, all tasks should receive processing time proportional to their guaranteed utilizations when they have work to do. In order to limit scheduling overhead, we allocate the processor to one job at a time, hence the guaranteed utilization is only realized on the average over some interval of time. The degree to which the actual utilization over a time interval approximates the guaranteed utilization indicates the fairness of the scheduler [40, 59]. We are interested in fairness because a fair scheduler is likely to lower response times and meet more deadlines on the average than an unfair scheduler.

A scheduler is said to be *fair* when the *normalized service* received by each task which is backlogged throughout an interval differs by no more than a *fairness threshold* [60], $\mathcal{F} \geq 0$. (A task is backlogged throughout an interval if at any time in the interval at least one job from the task is eligible for execution.) Let $s_i(t_1, t_2)$ denote the service (i.e., amount of processor time) received by a task T_i during the interval (t_1, t_2). The normalized service received by T_i in the interval is the ratio

[1] Typically, the timer which releases $J_{1,5}$ is not set until $J_{1,4}$ is released, preventing the inter-release time from being less than the guaranteed inter-release time. However in a system of distributed tasks whose end-to-end jobs consist of chains of subjobs as discussed in Section 4.3, subjob $J_{i,j,k}$ can be released "on time" even though subjob $J_{i,j-1,k}$ is released late unless the Release Guard Protocol [25] is used.

$s_i(t_1, t_2)/U_i^*$, where the fraction of processor time allocated to T_i is equal to the guaranteed utilization U_i^*. Hence an algorithm is fair to within $\mathcal{F}$ over the interval (t_1, t_2) if

$$\left| \frac{s_i(t_1, t_2)}{U_i^*} - \frac{s_j(t_1, t_2)}{U_j^*} \right| \leq \mathcal{F} \tag{6.1}$$

for each pair of tasks T_i and T_j which are backlogged throughout the interval. In the ideal case, $\mathcal{F} = 0$ and

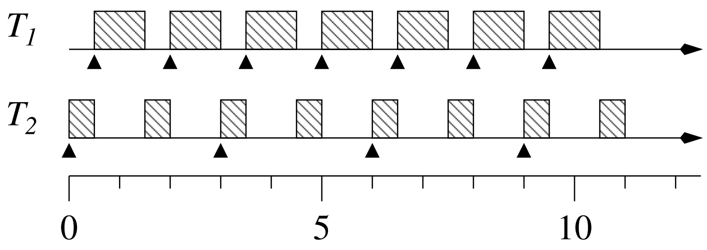
$$\frac{s_i(t_1, t_2)}{s_j(t_1, t_2)} = \frac{U_i^*}{U_j^*} \tag{6.2}$$

which implies $s_i(t_1, t_2) = U_i^*(t_2 - t_1)$ for each backlogged task T_i. The minimum value of $\mathcal{F}$ which satisfies Equation 6.1 over any nonzero interval indicates the fairness of an algorithm. While the fairness parameter $\mathcal{F}$ is theoretically useful for comparing the maximum unfairness of scheduling algorithms, the connection between fairness and either percent deadlines met or average response time is unclear.

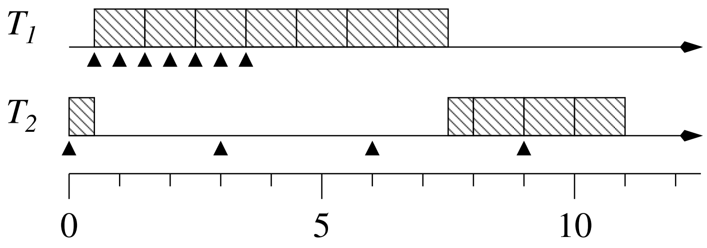
As long as the release times of jobs are periodic, fairness is not an issue. This is shown in Figures 6.3 and 6.4 for a system of two tasks with execution times of 1 time unit each and guaranteed inter-release times of 1.5 and 3 time units, respectively. (Again, we indicate release times by filled triangles.) The DM scheduler is unfair because clusters of releases from a high priority task can delay the completion of low priority tasks causing them to be starved, as shown in Figure 6.3(b). Now consider the same system scheduled according to EDF. (The scheduling deadlines of jobs are shown by filled circles.) Clustered releases of jobs can also cause starvation, as Figure 6.4(b) shows.

In contrast, the SS and CUS algorithms are fair because they strictly emulate the behavior of periodic tasks and hence ensure that clustered releases in one task cannot effect other tasks. As shown in the previous chapter, however, a CUS does not perform as well as a TBS because it does not use background time to improve response times and increase met deadlines.

When jobs are release periodically, a TBS is also fair Figure 6.5(a) shows an example scheduled according to ISM-EDF with a TBS server per task. The guaranteed utilizations of the servers are $U_1^* = 1/3$ and $U_2^* = 2/3$. Job releases (and hence server deadlines) are periodic so the difference in normalized service

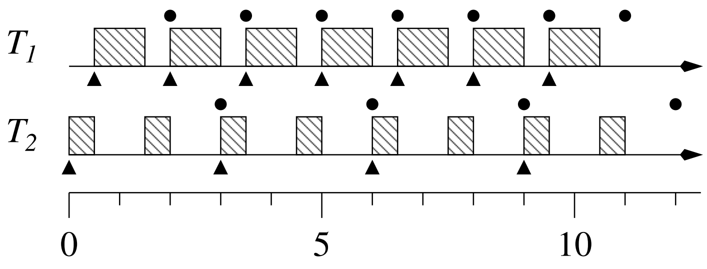


(a) Periodic

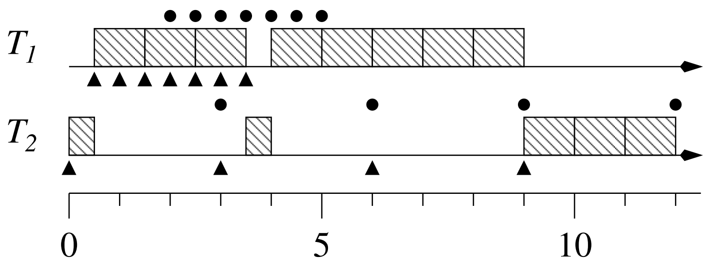


(b) Clustered

Figure 6.3: Effect of clustered releases on DM.

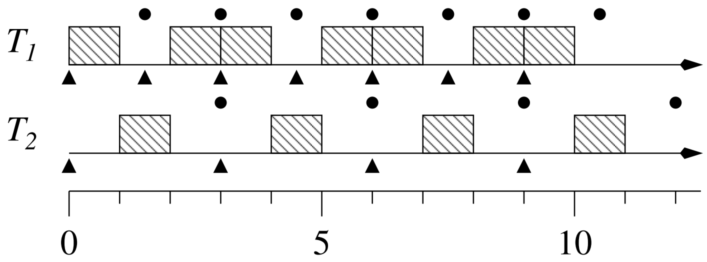


(a) Periodic

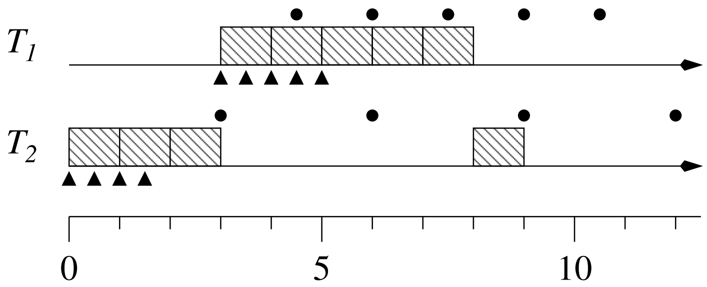


(b) Clustered

Figure 6.4: Effect of clustered releases on EDF.



(a) Periodic



(b) Clustered

Figure 6.5: Effect of clustered releases on ISM-EDF using TBS.

given to each task over any interval is bounded. On the other hand, if the releases of jobs in T_2 are clustered while T_1 is idle after which the releases of jobs in T_1 are clustered, as shown in Figure 6.5(b), the server deadlines for T_2 are sufficiently far into the future with respect to the server deadlines of T_1 that only jobs in T_1 are executed in the interval $(3, 9]$. In effect, the TBS server is penalizing T_2 for having used background time during $(0, 3]$ while T_1 was idle. An arbitrary amount of work released by a task while other tasks are idle results in the potential for starvation by the same arbitrary amount. This result is not surprising since TBS is a preemptable variant of the Virtual Clock (VC) network packet scheduling algorithm [61–63] which is known to be unfair [59, 64].

Note that fairness is a more stringent requirement on servers than ensuring that tasks receive their guaranteed utilizations on the average. It implies that tasks are allocated an amount of time at least equal to their guaranteed execution times over intervals with a length equal to their guaranteed inter-release times.

If we modify the TBS algorithm so that servers do not penalize tasks for additional service received while other tasks are idle, fairness is greatly improved. Conceptually, the improved algorithm would allocate processor time to back-

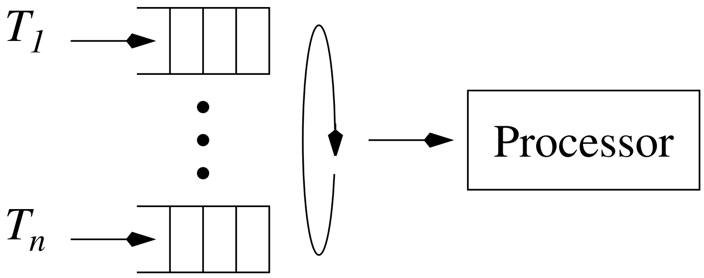


Figure 6.6: Generalized Processor Sharing Model.

logged tasks in proportion to their guaranteed utilizations. If the jobs in a task do not overrun and are not released too soon, each job is guaranteed to receive sufficient processor bandwidth between its release and its deadline to complete in time. (This assumes that the system is schedulable on the basis of the guaranteed parameters.) The improved algorithm is a preemptable variant of the Weighted Fair Queueing (WFQ) or Packet-by-Packet Generalized Processor Sharing (PGPS) algorithm used to schedule network packets [39,40]. Because of improved fairness, Weighted Fair Queueing Servers (WFQS) are expected to perform better than TBS on workloads with significant release time jitter. (See [60,64,65] for further discussions of the properties of the latency-rate and rate-proportional classes of servers to which WFQ and VC belong. See also [66] for a comparison of other network scheduling algorithms.) We now describe WFQ servers by first describing the Generalized Processor Sharing (GPS) algorithm [59,67,68] that they approximate.

The GPS algorithm, as shown in Figure 6.6, is an idealized weighted round robin scheduling technique in which each backlogged server is given an infinitesimally small time slice proportional to its utilization. A *round*, R, is the length of time necessary to give each backlogged server one time slice. Because the set of backlogged servers (and their total utilization) varies, the length of a round does also. Clearly, a literal implementation a GPS server is not practical due to the exorbitant overhead of small time slices. The WFQ algorithm approximates the fairness of GPS (over sufficiently long time intervals) but with much lower overhead.

A WFQS emulates a GPS system by scheduling jobs in order of their *finish numbers*, f, where the finish number of a job is the round in which the job would have finished had it been executed by a GPS server, i.e., a WFQS prioritizes jobs by their completion time under GPS. A job executed by a WFQS will finish no later than the round designated by its finish number.

We now turn our attention to computing the round number. The rate at which the round number changes is inversely proportional to the total utilization of the backlogged servers and hence the rate is constant except when servers become backlogged or idle, i.e., when jobs are released or when they complete. Hence, we only need to update the round number at these times.

We observe that the round number can be updated by either "looking forward or "looking backward". In a forward-looking algorithm, such as the one from [69] shown in Figure 6.7, the round number is brought up to date with every release and completion. We also see that the finish number is updated in the same way as the scheduling deadline of a TBS except that the scheduling deadline of a TBS is computed using physical time, which progresses at a constant rate, while the finish number of a WFQS is computed using virtual time (round number), which progresses at a rate inversely proportional to the total utilization of the backlogged servers. The increased fairness of a WFQS comes from the fact that virtual time, in the form of the round number, increases more rapidly when some servers are idle so that the finish number of a job released to an idle server does not lag too far behind the finish numbers of jobs released to backlogged servers.

In a backward-looking WFQ algorithm, the round number is updated to account for changes in the total backlogged utilization only when a job is released. As the algorithm in Figure 6.8 (adapted from the Fair Queueing algorithm of [39]) shows, we look backward to the time when the round number was last updated, then move forward one completion event at a time accounting for the changes in total backlogged utilization caused by job completions. The finish number of the job being released is computed after the round number has been updated.

Figure 6.9(a) shows a schedule in which jobs are released periodically to the WFQ servers. The schedule is identical to the one obtained under TB servers (Figure 6.5(a). Figure 6.9(b) shows a schedule of the clustered release example under a WFQS. We see that a WFQS does not penalize T_1 for having received additional bandwidth while T_2 is idle. It is expected that the performance of WFQ algorithm will be superior to the performance of a TBS or classical EDF scheduling.

(Assuming $J_{i,j}$ is assigned to server $S_k \ldots$)
If all servers are idle,
  Reset round number R to current time t,
  Set last update time $t_{update} = t$,
  Set finish number $f_k = e_{i,j}/U_k^*$,
  Release $J_{i,j}$ to ready queue with a scheduling deadline of f_k.
Else if server S_k is idle,
  Update round number $R = (t - t_{update})/U_{S_{total}}$,
  Set last update time $t_{update} = t$,
  Increment $U_{S_{total}}$ by U_{S_k},
  Set finish number $f_k = R + e_{i,j}/U_k^*$,
  Release $J_{i,j}$ to ready queue with a scheduling deadline of f_k.
Else,
  Add $J_{i,j}$ to ready queue of server S_k for later release.

(a) On release of $J_{i,j}$

(Assuming $J_{i,j}$ is assigned to server $S_k \ldots$)
If server S_k is no longer backlogged,
  Increment round number R by $(t - t_{update})/U_{S_{total}}$,
  Set last update time $t_{update} = t$,
  Decrement $U_{S_{total}}$ by U_{S_k}.
Else,
  Remove next job J from ready queue of server S_k,
  Increment finish number f_k by $e_{i,j}/U_k^*$,
  Release $J_{i,j}$ to ready queue with a scheduling deadline of f_k.

(b) On completion of $J_{i,j}$

Figure 6.7: Forward-Looking Weighted Fair Queueing Algorithm.

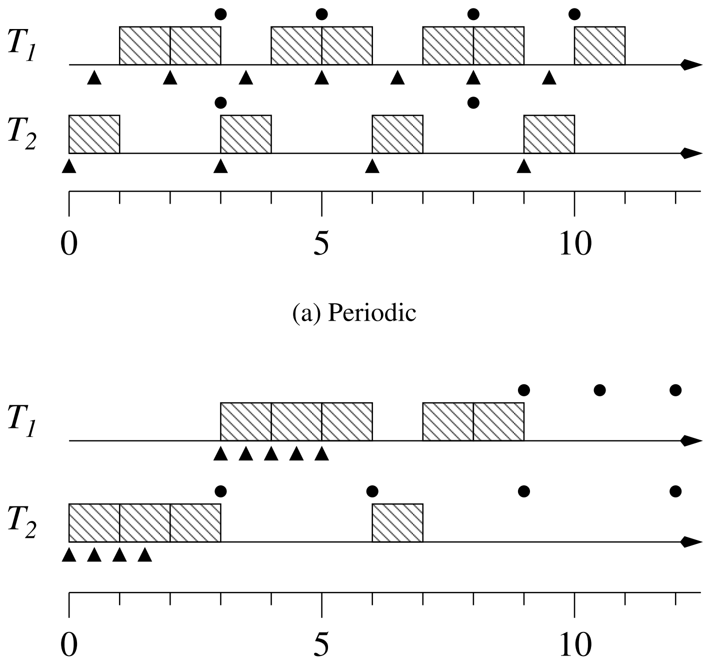
Update round number R following algorithm in Figure 6.8(b),
Set finish number f_k of server S_k (to which $J_{i,j}$ is assigned), to
    $\max(f_k, R) + e_{i,j}/U_{S_k}$, where U_{S_k} is the processor
    allocation of server S_k,
If server S_k was idle, increase total utilization
    of backlogged servers $U_{S_{total}}$ by U_{S_k},
Release job $J_{i,j}$ to ready queue with a scheduling deadline of f_k,
Add tuple $(J_{i,j}, f_k)$ to finish queue $\mathcal{Q}$ ordered by finish number.

(a) On release of $J_{i,j}$

If processor was idle prior to release of $J_{i,j}$,
    Reset round number R to current time t,
    Reset round number of last update R_{update} to t,
    Reset time of last update t_{update} to t,
    Reset finish number f_k of each server S_k to t.
Otherwise,
    While last update time t_{update} is less than t,
        Get tuple (J, f) from $\mathcal{Q}$, where f has the
            smallest finish number of tuples in $\mathcal{Q}$,
        Compute elapsed time since last update, $\Delta t = t - t_{update}$,
        If job would have finished by t under GPS
          (i.e., if $f \leq R_{update} + \Delta t/U_{S_{total}}$),
            Increment t_{update} by $(f - R_{update}) * U_{S_{total}}$,
            Update R_{update} to f,
            If server S_k became idle with completion of J,
                Decrement $U_{S_{total}}$ by U_{S_k},
            Remove (J, f) from $\mathcal{Q}$.
        Otherwise,
            Update round number R to $R_{update} + \Delta t/U_{S_{total}}$,
            Set round number of last update R_{update} to R,
            Set time of last update t_{update} to t.

(b) Update round number

Figure 6.8: Backward-Looking Weighted Fair Queueing Algorithm.



(a) Periodic

(b) Clustered

Figure 6.9: Effect of clustered releases on WFQS.

6.3 Comparison Methodology

The general approach to comparing the performance of the classes of scheduling algorithms is the same as in Section 5.2. Again, the metrics are percent deadlines met and average response time, which can be considered from either the perspective of the system or the perspective of the tasks in the system. We report results from both perspectives.

The time between releases of consecutive jobs in a task are random variables with a common distribution, either uniform or exponential. The mean inter-release times of the tasks were chosen uniformly from the range $[1000, 10000]$, with minimum inter-release times of 1.0 to maximize the inter-release time variations of jobs within a task. The average utilization of a task was obtained by multiplying the average system utilization by a utilization factor for the task. The utilization factors of the tasks in a system were uniformly distributed in the range $(0, 1]$ and normalized such that the sum of the factors equals 1.0. The mean execution times of jobs in a task are equal to the mean utilization of the task multiplied by the mean inter-release time.

Table 6.1: Jitter Simulation Parameters.

Parameter	Values
Distribution Types	Uniform, Exponential
Average Utilizations	0.50, 0.75, 0.90, 0.95
Variations	Period-only
Dependency Patterns	$\{1\}$, $\{1, 1\}$

For the initial performance comparison, we let the execution times of jobs in a task be equal to the guaranteed execution time of the task, which we set equal to the mean execution time of the task. (Since no jobs overrun, the performance of the OSM is identical with the baseline scheduling algorithm and hence need not be considered.)

When both the inter-release times and execution times vary (Section 6.6), the parameters of the inter-release time distributions were chosen as above. The execution times of jobs in a task were taken from a common distribution with the mean computed as above and the minimum chosen to be 1.0 to maximize the effect of execution time variations. Because variations are often correlated, we considered both dependent and independent distributions with dependencies modeled as being exclusively between jobs in a task and following a fixed pattern, as was done in Chapter 5. A summary of the parameters are given in Table 6.1.

As in Section 5.2, the workloads used in the simulations had average system utilizations of 50, 75, 90 and 95%. For each average system utilization, distribution type and dependency pattern, we generated 100 systems, each consisting of 8 tasks. Each of the tasks in a system had at least 1000 jobs. (To decrease confidence intervals at high utilizations without causing simulation times to become excessive, the minimum number of jobs in a task varied from 1,000 at 50% average utilization to 4,000 at 95% average utilization.)

6.4 Baseline

Because the results of Chapter 5 showed that the average performance of the system does not depend on the initial phase of the tasks, we restrict our attention to in-phase releases. Also, we only consider systems with 8 tasks. Therefore, we start our comparison of the performance of the various algorithms by comparing classical EDF and DM scheduling on workloads with release time jitter and no execution time variations.

Figure 6.10 shows that from the system perspective the average ratio of percent deadlines met by the EDF algorithm with respect to the DM algorithm deteriorates rapidly from nearly 1.0 at 75% average system utilization to less than 50% at 95% average system utilization for exponential workloads with or without dependencies. Correspondingly, the average ratio of the response times increases to around 1.8 at 95% average system utilization. While the results are somewhat better for uniform distributions, the DM algorithm still performs better than the EDF algorithm. From the task perspective, the number of deadlines met by the EDF algorithm is inferior to the DM algorithm except for uniform workloads with no dependencies (see Figure 6.11). However, the average response times of jobs scheduled EDF is significantly worse than when they are scheduled DM. For the exponential workload with dependencies, the average ratio of response times at 95% average utilization was over 60.0! Clearly, heavily loaded systems with release time jitter should not be scheduled according to classical EDF.

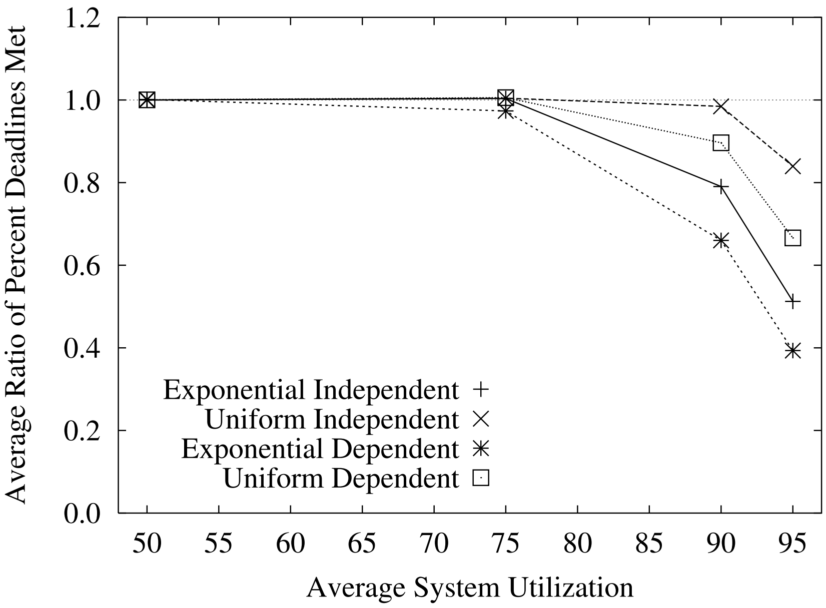
6.5 Isolation Server Method

We now consider the performance of the Isolation Server Method for both fixed-priority and deadline-driven systems from both the system and task perspectives. Previous results showed that the performance of a single server for all tasks and a server for each task bound the results of intermediate configurations. Therefore, we consider systems with either 1 or 8 servers. Since a CUS did not perform as well as a TBS under overload, we will not consider it further. We therefore start with a comparison of the performance of ISM-EDF using a WFQS to the performance of ISM-EDF using a TBS.

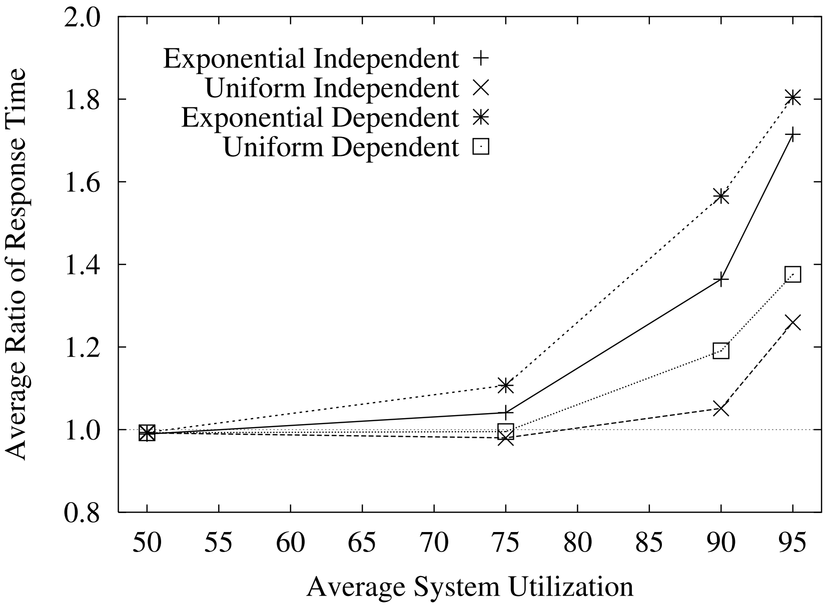
6.5.1 Comparison of WFQS and TBS

As discussed in Section 6.2, it was expected that ISM-EDF using a WFQS will perform better than ISM-EDF using a TBS. We note that a single WFQS is equivalent to a single TBS server and hence only consider the case of a server per task.

Contrary to intuition, Figures 6.12 and 6.13 indicate that a configuration with a WFQS per task meets fewer deadlines than a similar configuration with a TBS per task. The average response times also increase. The difference becomes more pronounced as the average utilization approaches 1.0.

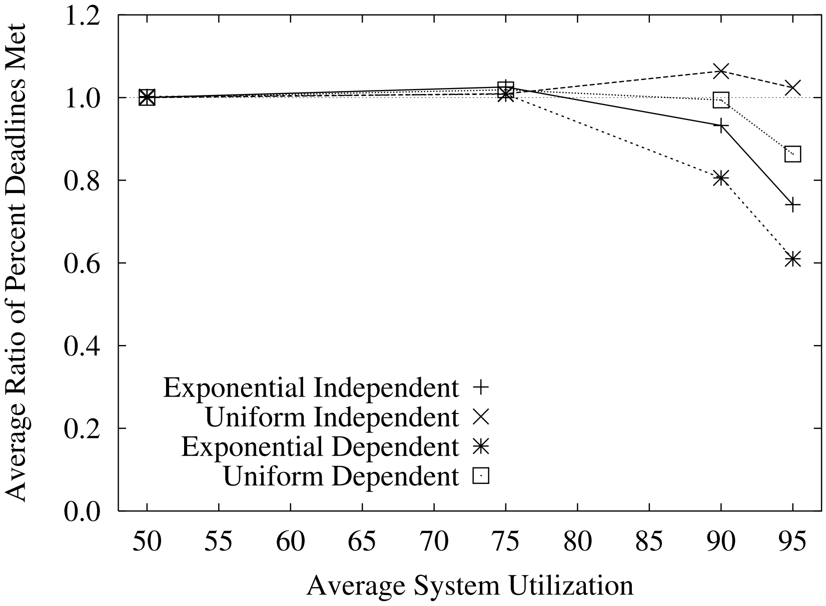


(a) Deadlines Met

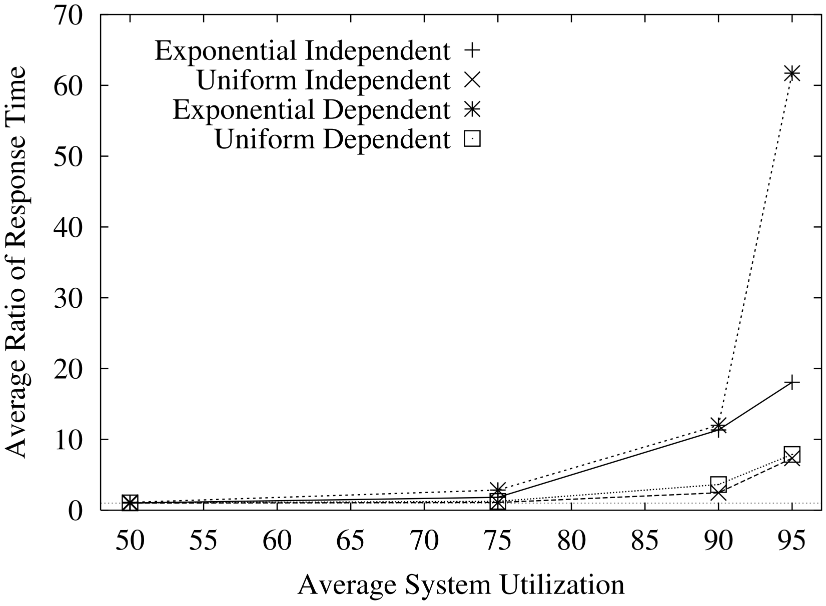


(b) Response Time

Figure 6.10: EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics of systems with a DM scheduler.

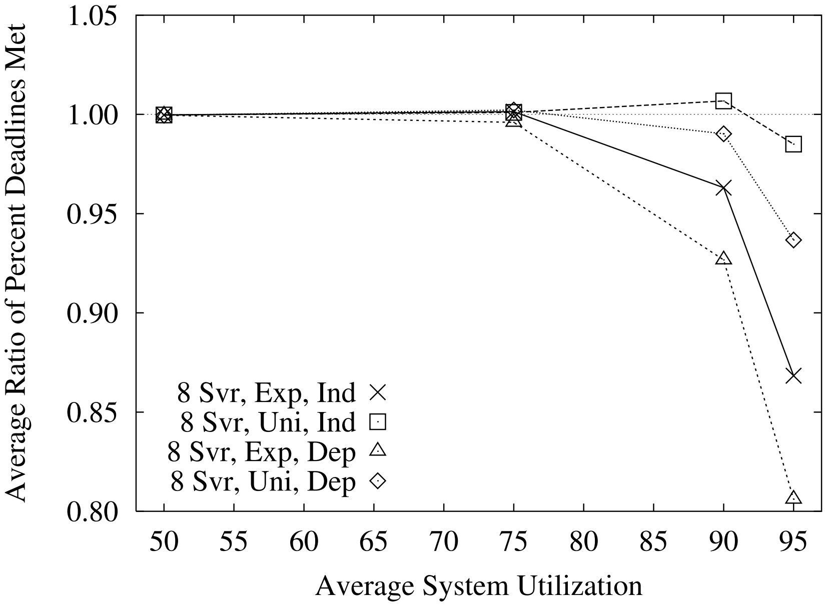


(a) Deadlines Met

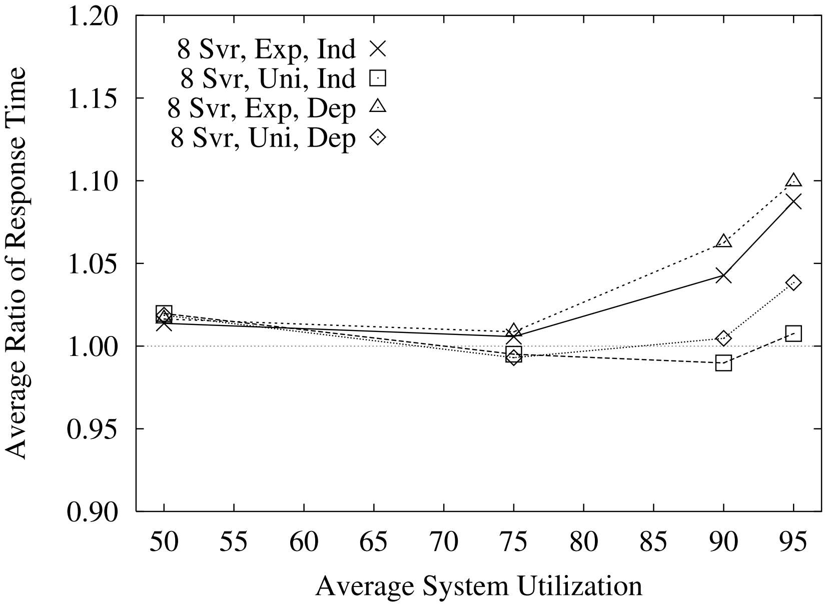


(b) Response Time

Figure 6.11: EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics of systems with a DM scheduler.

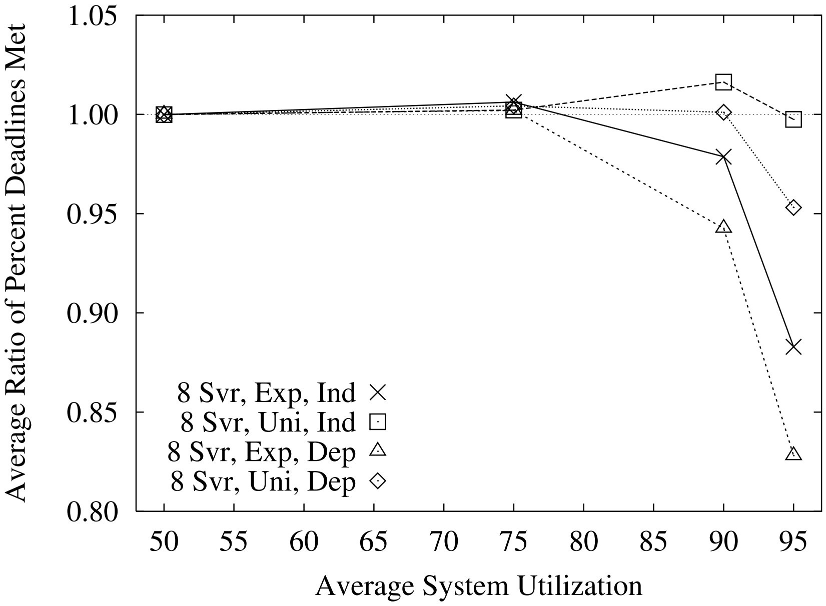


(a) Deadlines Met

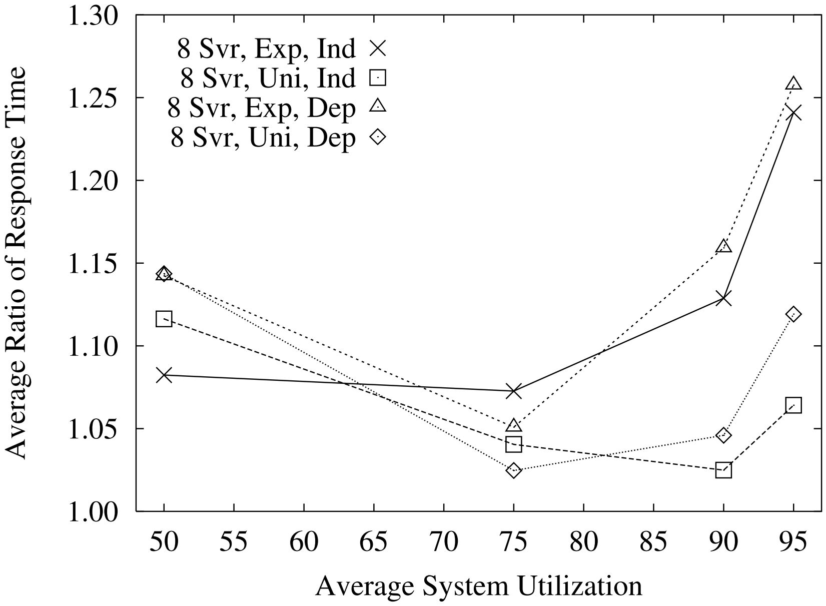


(b) Response Time

Figure 6.12: WFQS vs. TBS from System Perspective: the average of the ratios of the metrics for systems with Weighted Fair Queueing Servers to the metrics for systems with Total Bandwidth Servers.



(a) Deadlines Met



(b) Response Time

Figure 6.13: WFQS vs. TBS from Task Perspective: the average of the ratios of the metrics for systems with Weighted Fair Queueing Servers to the metrics for systems with Total Bandwidth Servers.

This can be explained by considering the effects of using virtual time (round number) to set the scheduling deadline (finish number) of a job under a WFQS rather than physical time under a TBS. As long as all WFQ servers are backlogged, virtual time progresses at the same rate as physical time and a WFQS will perform the same as a TBS. Virtual time progresses at faster rate than physical time during a processor busy interval when not all servers are backlogged. Consequently, the scheduling deadline of a job released to an idle server does not lag the scheduling deadlines of jobs released to the busy servers. (This is how a WFQS improves fairness over a TBS.) This causes an increased likelihood that a job from a busy server will be scheduled before a job released to an idle server. When the average utilization of the system is high, scheduling a job from a busy server in preference to a job released to an idle server will tend to increase the number of deadlines missed because it is likely that the jobs in the busy servers have already missed their deadlines while a job released to an idle server is more likely to met its deadline. Furthermore, scheduling a job from a busy server in preference to one released to an idle server increases average response times because it is likely that the jobs in busy servers are already late.

In summary, allocating bandwidth fairly has a negative impact on the percentages of deadlines met and average response times of tasks. Because of this, we will use Total Bandwidth Servers for the remainder of the performance study.

6.5.2 Performance of ISM vs. Baseline

Figures 6.14 and 6.15 compare the performance of ISM-DM with classical DM on configurations with a single server for all tasks and a server per task. Almost without exception, the performance of ISM-DM is inferior to the performance of classical DM for both percent deadlines met and average response time. (The only exception is for independent uniform distributions and a single server where the performance of ISM-DM is similar to DM when compared on a task basis.)

The performance of ISM-DM on independent workloads is better than on dependent workloads, while the performance on workloads from uniform distributions is better than on workloads from exponential distributions. Thus we see that increased variation causes lower performance. Unlike what was observed for systems with overrun, ISM-DM meets more deadlines and has a shorter average response time with a server per task than with one server for all tasks. This is because clustered releases of jobs in a task are executed as if they had been released periodically, if each task has its own server, while the cluster of work will cause jobs from lower priority tasks to be starved with a single server for all tasks. Thus, the best performance of ISM-DM in the presence of jitter is obtained on independent workloads taken from uniform distributions with a server per task. However, the figures show that classical DM scheduling is still to be preferred.

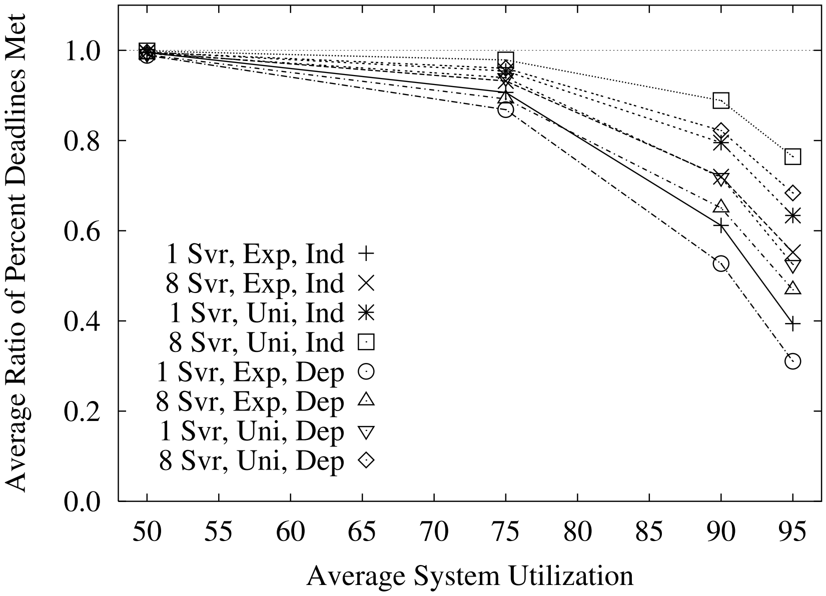
Figures 6.16 and 6.17 compare the performance of ISM-EDF to classical EDF. The performance of ISM-EDF with one server for all tasks is worse than the EDF algorithm while the performance of ISM-EDF with a server per task is better than classical EDF, particularly for exponential workloads and workloads with dependencies. (The width of the 95% confidence intervals for average response time ratios in Figure 6.17 range from around $\pm 25\%$ at 50% utilization to around $\pm 5\%$ at 95% utilization for a single server cases. The width of the 95% confidence intervals for the server per task cases were all less than $\pm 5\%$.) This is consistent with the results obtained for execution time variations in the previous chapter. Clearly, ISM-EDF should be configured with a server per task.

6.5.3 Performance of ISM-EDF vs. ISM-DM and DM

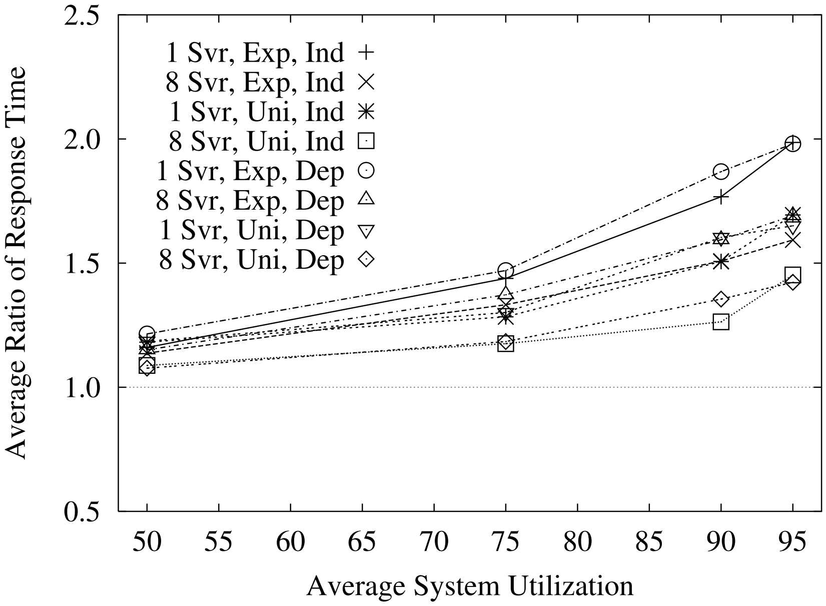
A comparison of the performance of ISM-EDF with the performance of ISM-DM (Figures 6.18 and 6.19) clearly shows that ISM-EDF performs better than ISM-DM when both use a server per task. (A server per task is the best configuration for both algorithms.) Both algorithms have nearly identical performance with a single server. Finally, the performance from the system and task perspectives were also nearly identical.

We now compare the performance of the ISM with the best configuration, namely ISM-EDF with a server per task, with the performance of classical DM. The results are shown in Figures 6.20 and 6.21. From the system perspective, classical DM provides better average response times and percent deadlines met. The reason, stated earlier, is that the highest priority jobs under classical DM also have the most jobs released in a given length of time. The difference in performance is within 20% for the server per task configuration, however. From the task perspective, ISM-EDF meets 5–10% more deadlines than classical DM, but at the cost of a dramatic increase in average response times.[2] This suggests that classical DM should be used to schedule systems with release time variations.

[2]The width of the 95% confidence interval for the single server, dependent exponential workload at 95% utilization is $\pm 52.4\%$.

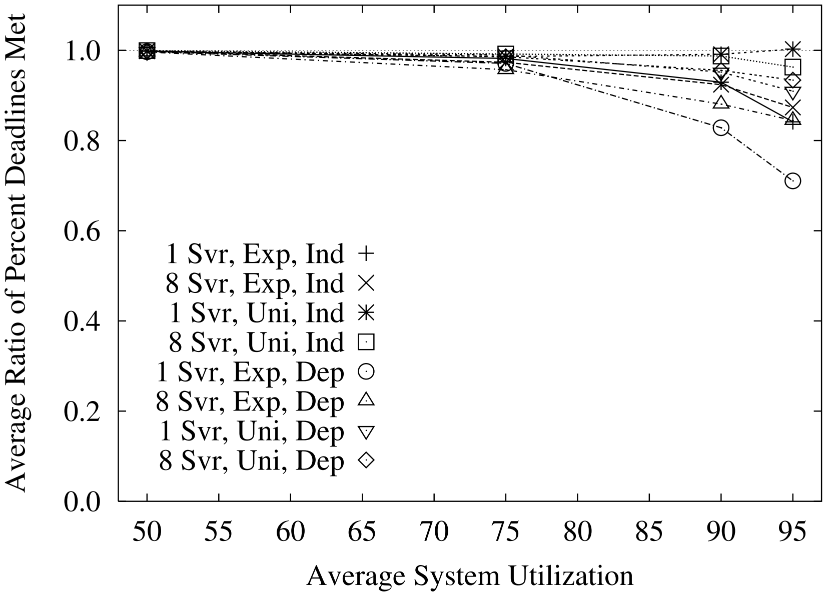


(a) Deadlines Met

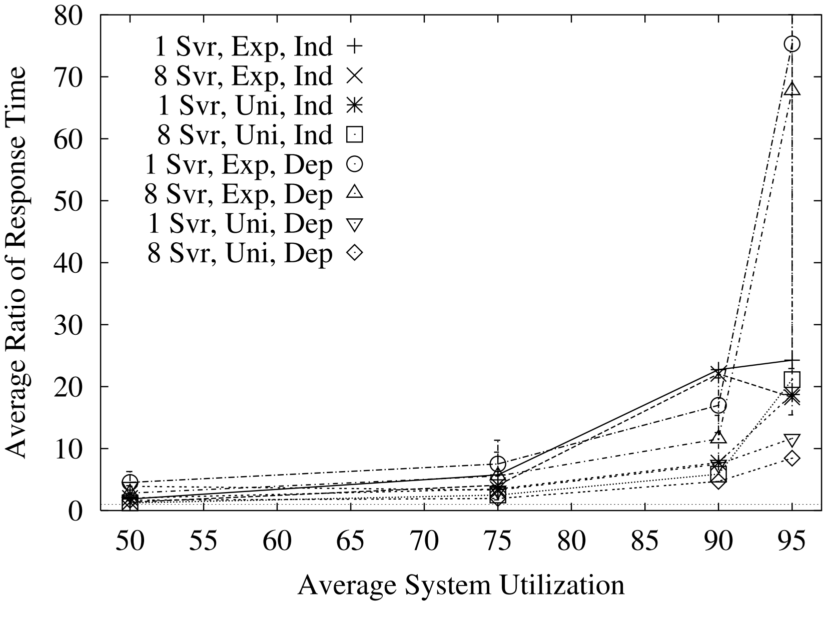


(b) Response Time

Figure 6.14: ISM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

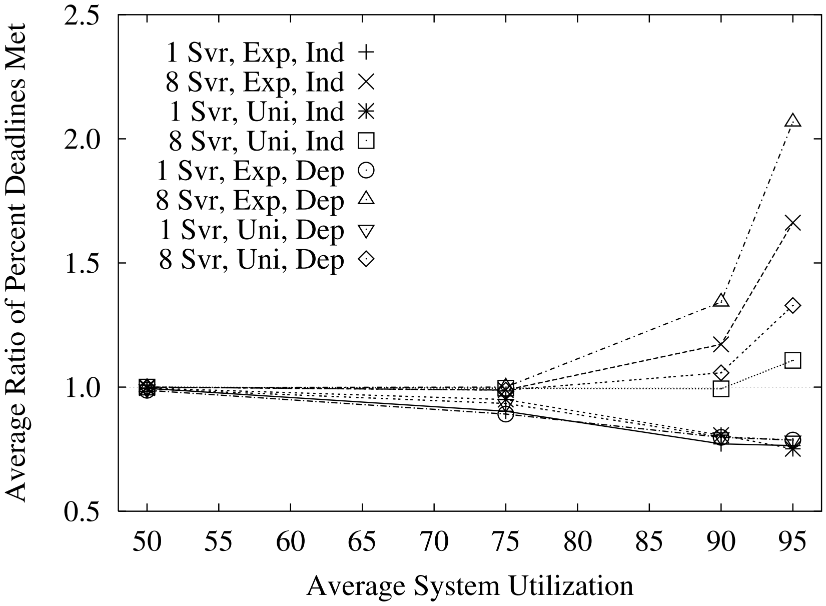


(a) Deadlines Met

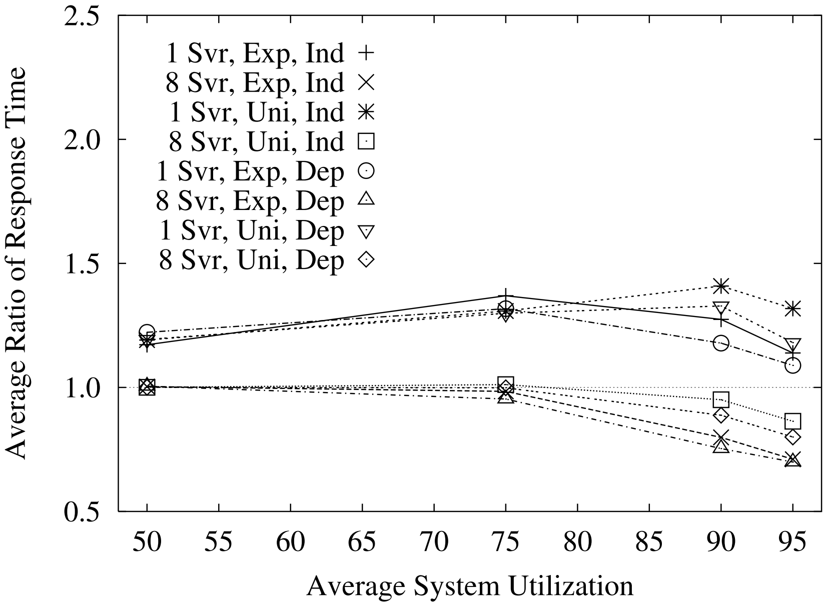


(b) Response Time

Figure 6.15: ISM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

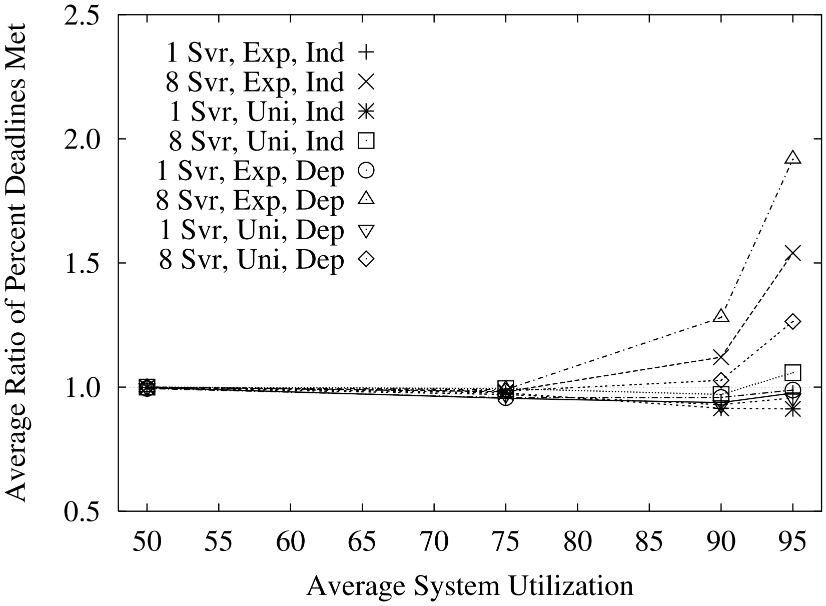


(a) Deadlines Met

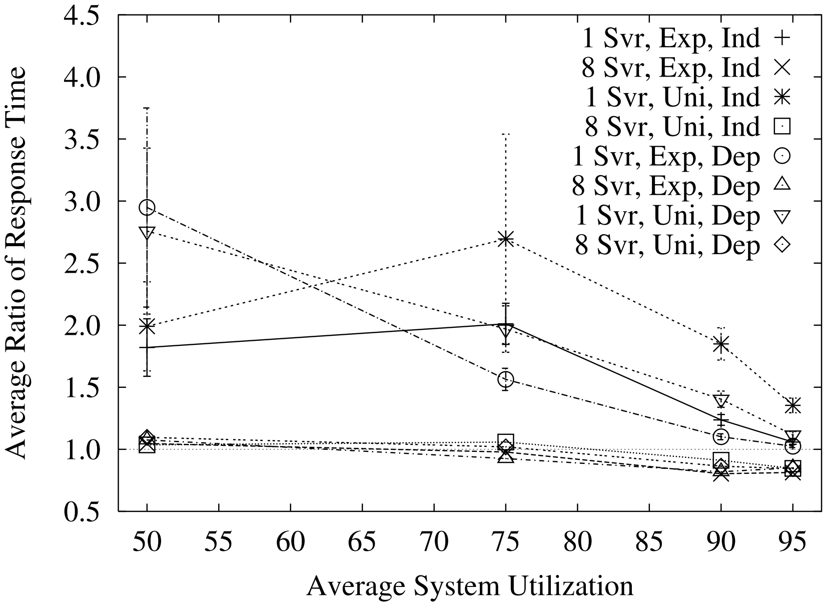


(b) Response Time

Figure 6.16: ISM-EDF vs. EDF from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.

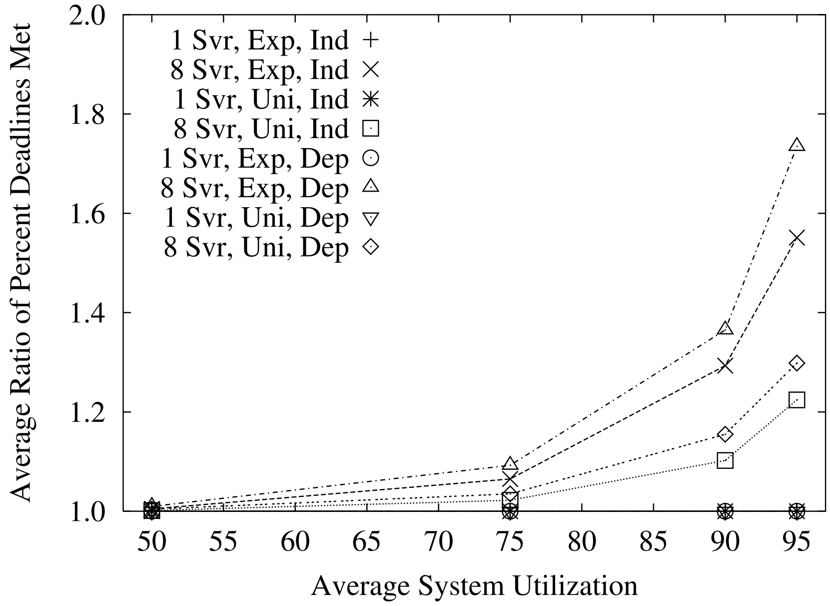


(a) Deadlines Met

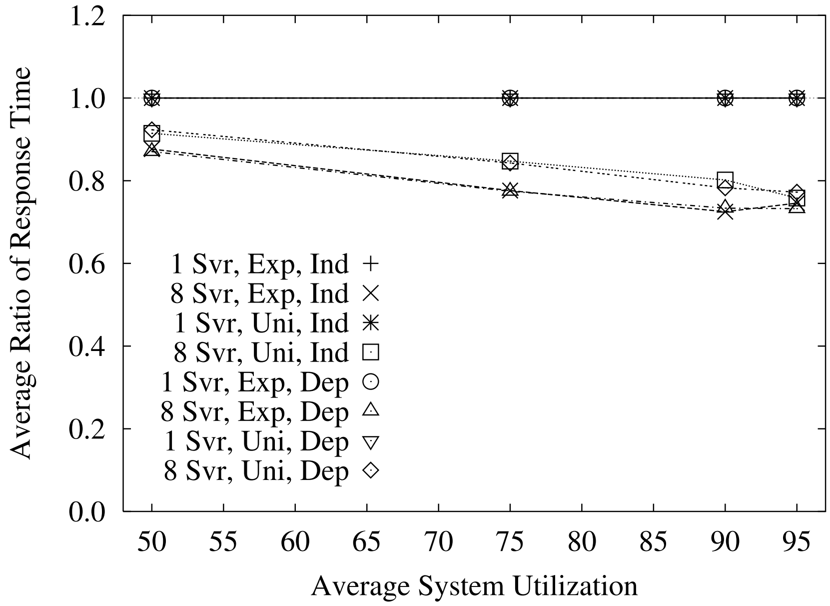


(b) Response Time

Figure 6.17: ISM-EDF vs. EDF from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.

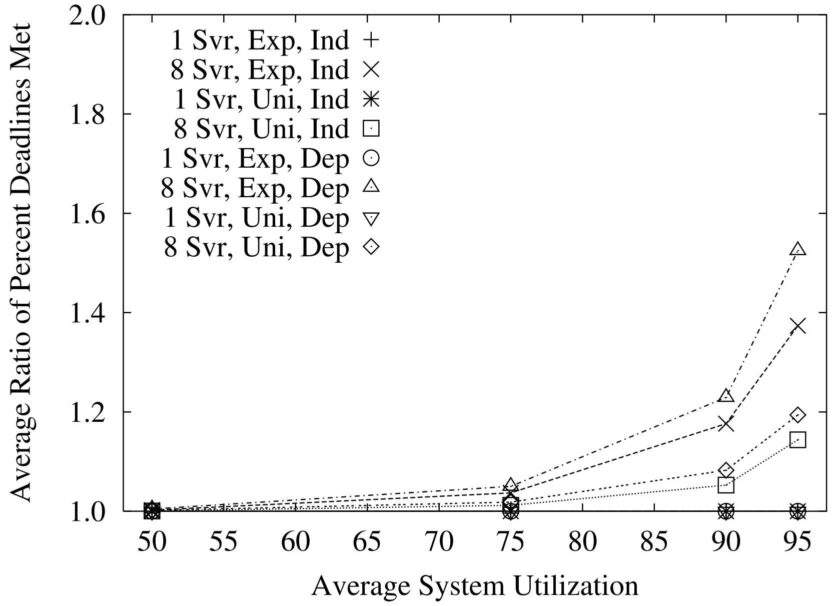


(a) Deadlines Met

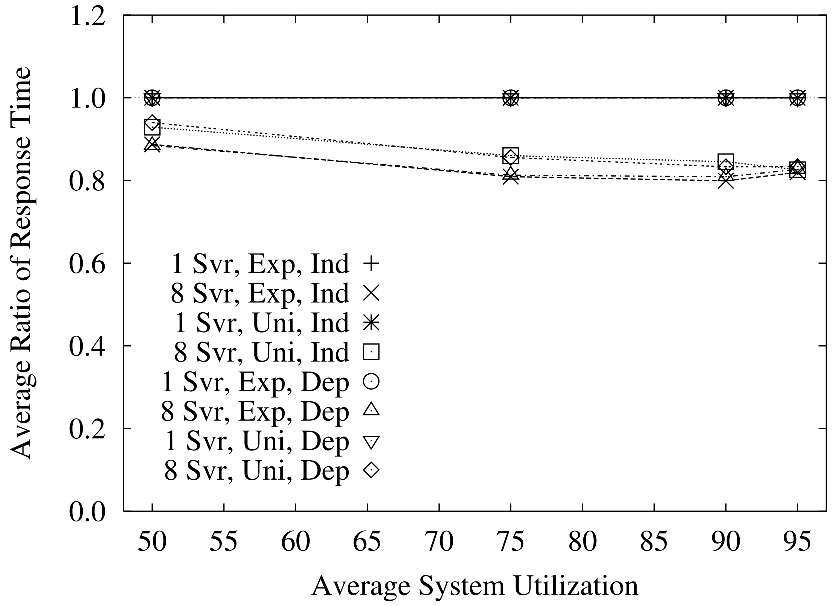


(b) Response Time

Figure 6.18: ISM-EDF vs. ISM-DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

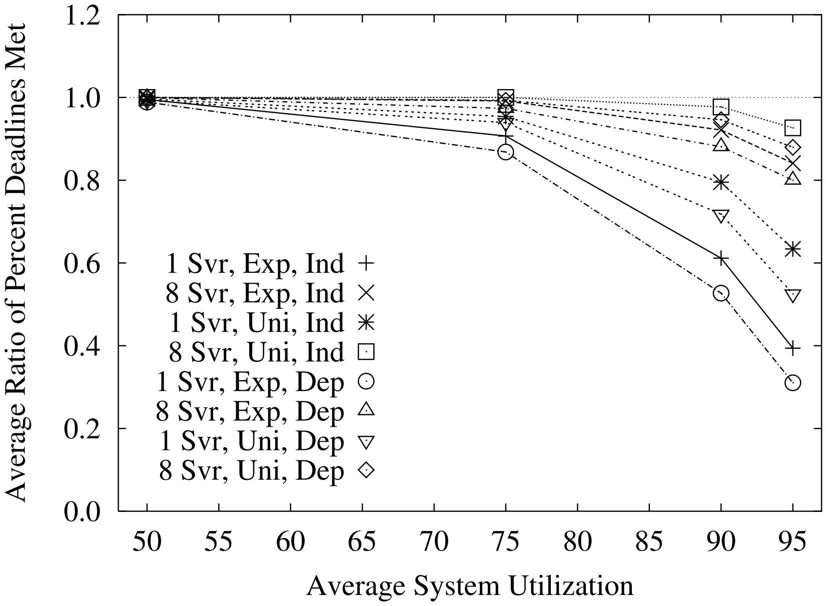


(a) Deadlines Met

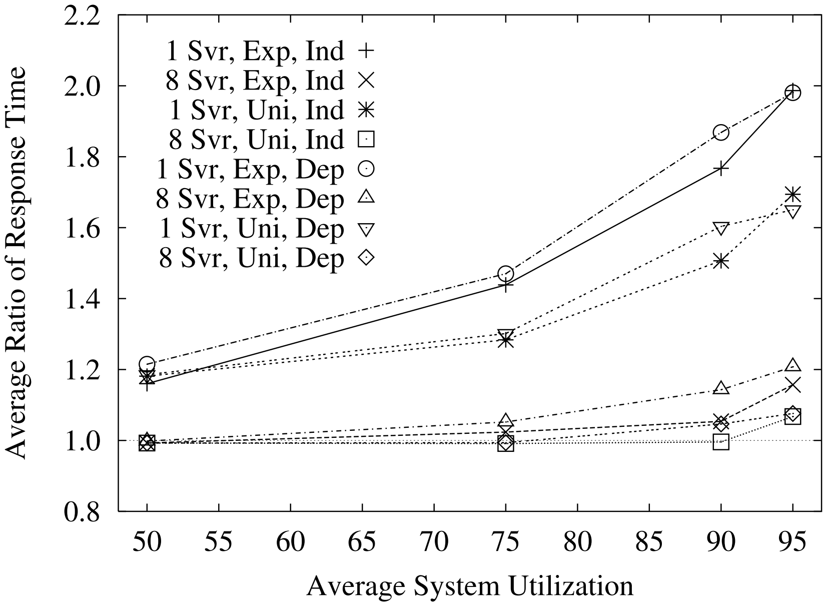


(b) Response Time

Figure 6.19: ISM-EDF vs. ISM-DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

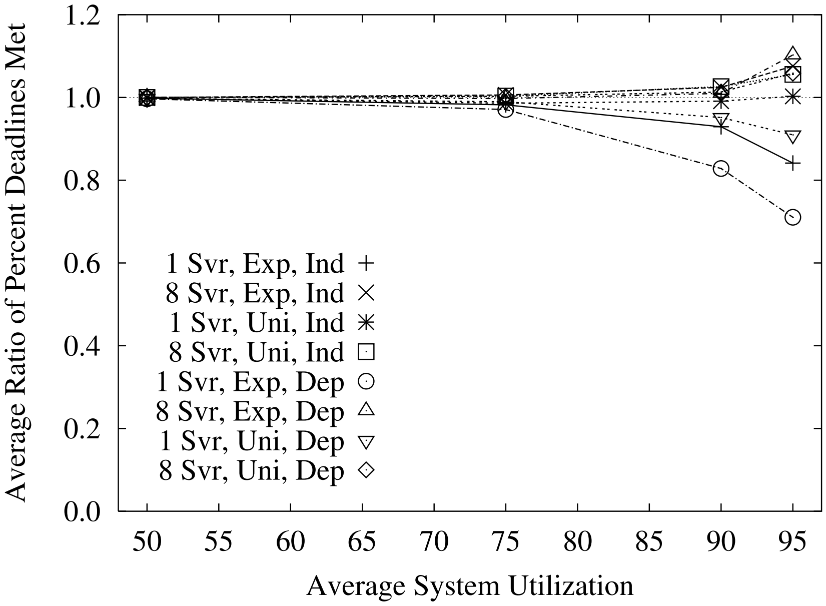


(a) Deadlines Met

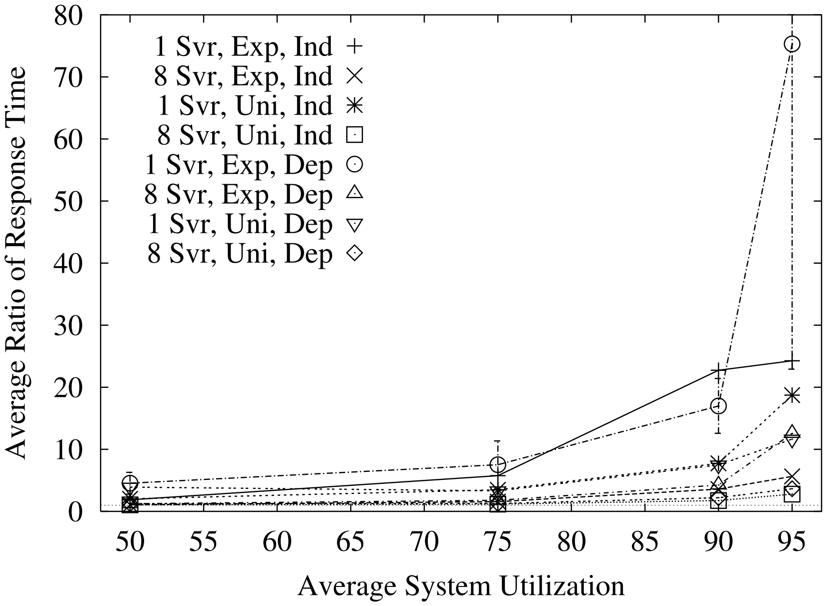


(b) Response Time

Figure 6.20: ISM-EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 6.21: ISM-EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.

6.5.4 Effect of Maximum Period Ratio

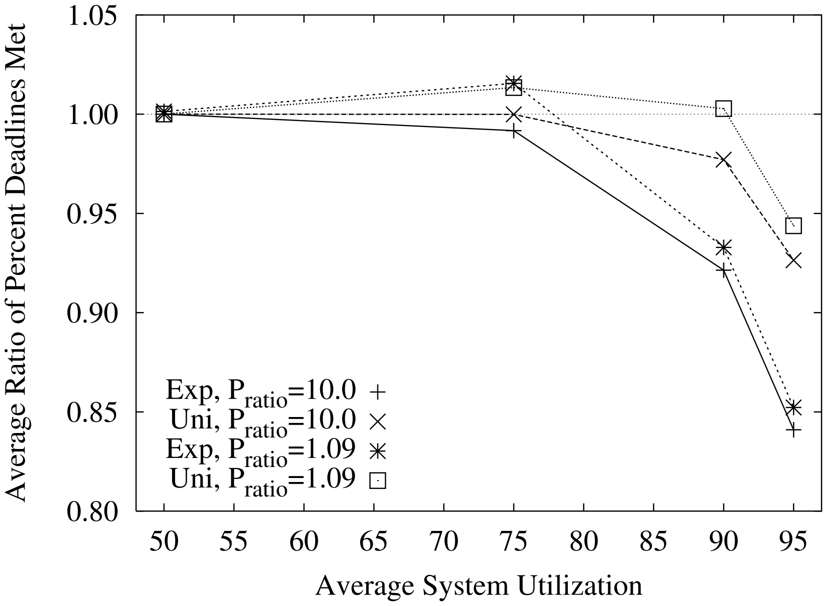
Thus far, the comparisons have shown that the performance of the DM scheduling algorithm is better than the performance of any of the alternative algorithms. One of the reasons is that the tasks with the highest release rate also have the highest priorities. Hence jobs in those tasks have lower response times and are more likely to meet their deadlines. As the guaranteed inter-release times of the tasks in the system become similar, it is expected that the performance of the DM algorithm will decrease.

As Figures 6.22 and 6.23 show, reducing the maximum period ratio (defined as the maximum to the minimum guaranteed inter-release times of tasks in the system) from 10.0 to 1.09 does indeed cause the performance of the ISM-EDF algorithm with a Total Bandwidth Server per task to improve in relation to the DM algorithm. However, the performance of ISM-EDF is still not as good as the performance of the DM algorithm. This is likely due to a fundamental difference between fixed and and dynamic priority assignments. As long as we assign give higher priorities to tasks which release more jobs, a fixed priority scheduling algorithm will perform better than a deadline driven one because a fixed priority scheduling algorithm is more predictable under overload. Because most systems will have a fairly wide range of guaranteed inter-release times, we will continue using a maximum period ratio of 10:1 for the remainder of the chapter.

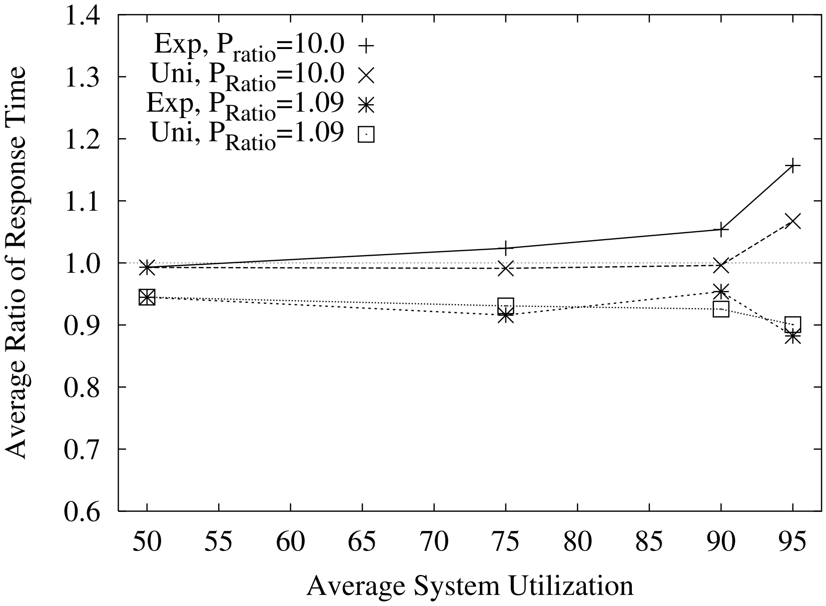
6.6 Execution and Release Time Variations

So far we have only considered the effects of release time variations on the performance of various scheduling algorithms. In this section, we present the effects of release time and execution time variations on the performance of the three classes of scheduling algorithms: classical, OSM and ISM.

In determining the performance of the scheduling algorithms, we employ a similar methodology to Sections 5.2 and 6.3. The mean inter-release times of the tasks were chosen uniformly from the range $[1000, 10000]$, with minimum inter-release times of 1.0 as before. The inter-release time of jobs in a task were taken from a common uniform or exponential distribution. The average utilization of a task was obtained by multiplying the average system utilization by a utilization factor for the task. The utilization factors of the tasks in a system were uniformly distributed in the range $(0, 1]$ and normalized such that the sum of the factors

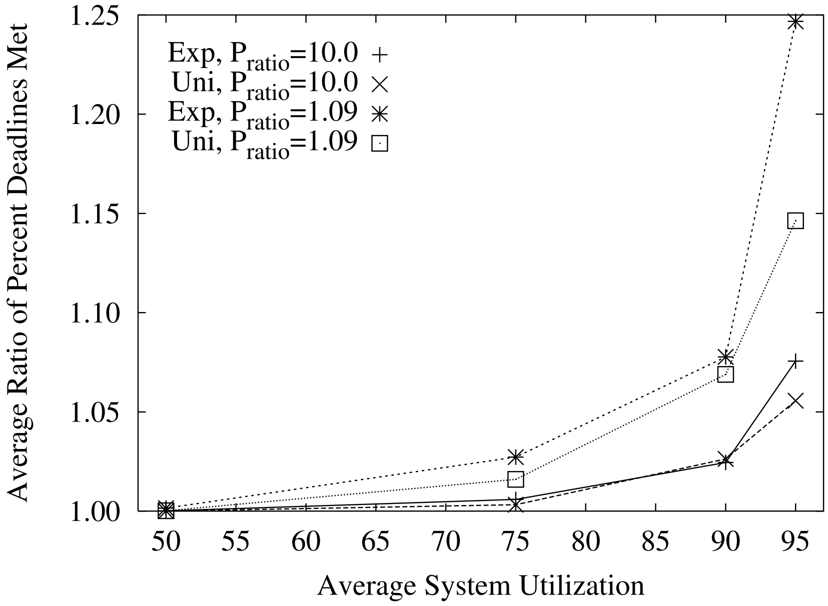


(a) Deadlines Met

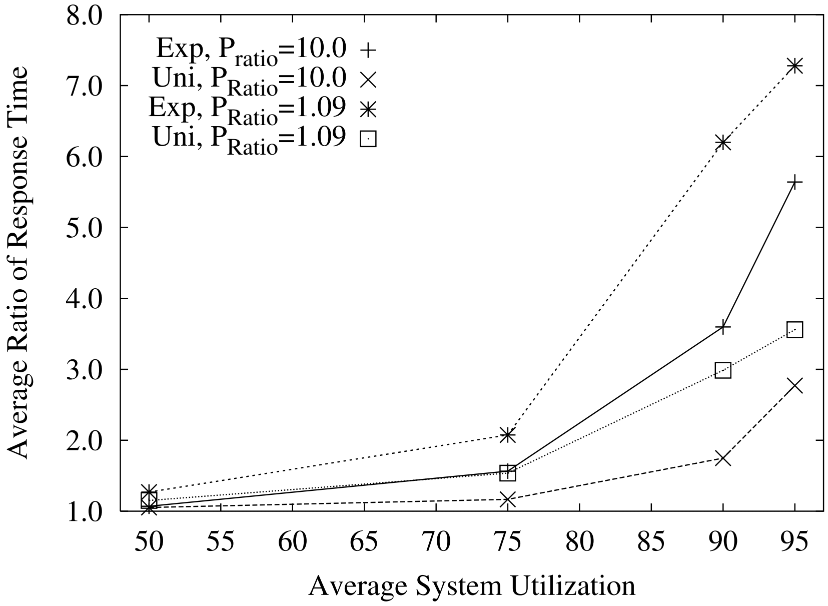


(b) Response Time

Figure 6.22: Effect of Maximum Period Ratio from System Perspective: the performance of ISM-EDF in comparison to the performance of DM for two maximum period ratios as shown by the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 6.23: Effect of Maximum Period Ratio from Task Perspective: the performance of ISM-EDF in comparison to the performance of DM for two maximum period ratios as shown by the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.

Table 6.2: Jitter and Overrun Simulation Parameters.

Parameter	Values
Distribution Types	Uniform, Exponential
Average Utilizations	0.50, 0.75, 0.90, 0.95
Variations	Both period and execution times
Dependency Patterns	$\{1\}$, $\{1, 1\}$

equaled 1.0. The mean execution times of jobs in a task were equal to the mean utilization of the task multiplied by the mean inter-release time. The minimum execution times were 1 time unit. The execution times of jobs in a task were also taken from a common uniform or exponential distribution. Again, dependencies were modeled as being exclusively between jobs in a task and following a fixed pattern, as was done in previous experiments.

Once again, the workloads had average system utilizations of 50, 75, 90 and 95%. For each average system utilization, distribution type and dependency pattern, we generated 100 systems, each consisting of 8 tasks. A minimum of 1,000 jobs in each task were released at 50% average utilization, increasing to a minimum of 4,000 at 95% average utilization. Because both the inter-release times and the execution times vary, the confidence intervals exceeded $\pm 5\%$ more frequently than we would have liked. From the system perspective, the maximum widths of the confidence intervals for deadlines met were strictly below 5%. The maximum widths of the confidence intervals for response times were as high as $\pm 30.8\%$ with an average of $\pm 6.4\%$. The maximum widths of the confidence intervals from the perspective of the tasks were significantly higher: up to $\pm 7.9\%$ for deadlines met and over $\pm 100\%$ for response times. However, the average width of the confidence intervals for deadlines met and response times were not nearly as excessive at $\pm 3.4\%$ and $\pm 16.8\%$. Most of the confidence intervals with widths over $\pm 5\%$ occurred for data appearing in only a few of the figures. Taking into account the widths of the confidence intervals did not qualitatively affect the interpretation of the results so we did not attempt to reduce the intervals. (Since the simulation times of some of the longest running data points already exceeded 30 hours, it would have been impractical to increase the minimum number of jobs enough to significantly decrease the widths of the worst confidence intervals.) We summarize the parameters which we varied in Table 6.2.

6.6.1 Performance of EDF vs. DM

The performance of EDF with respect to DM on workloads containing both release time jitter and execution time variations is consistent with previous comparisons. As Figures 6.24 and 6.25 show, DM behaves much more predictably under overload conditions than EDF. As was observed in Section 5.7, the difference in performance does not become pronounced until higher utilizations.

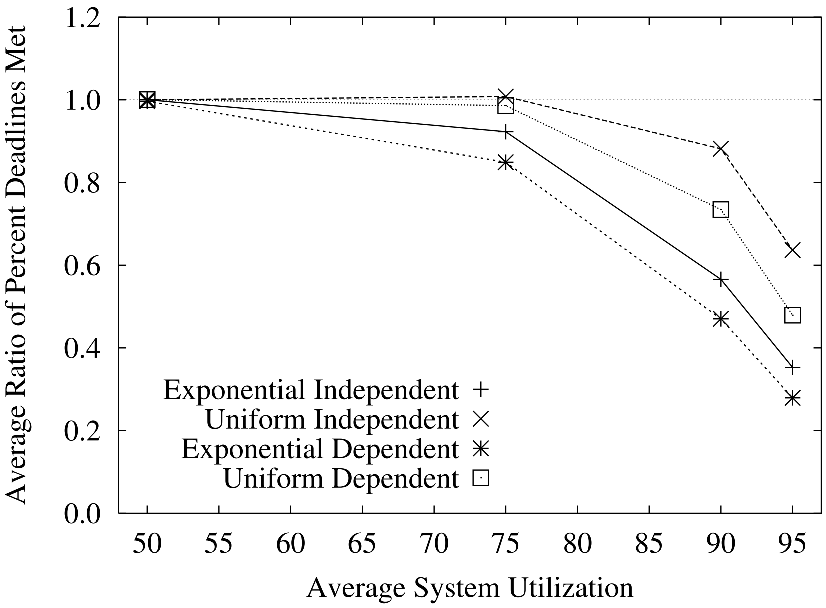
6.6.2 Performance of OSM vs. Baseline

A comparison of OSM-DM with the DM algorithm also yields no surprises (Figures 6.26 and 6.27). The performance of OSM-DM with a single server was better than a server per task and dependencies had only a small effect on the results. The only exception, from the perspective of the system, occurred for distributions with long tails in the single server configuration where the average response times were significantly better for OSM-DM. Even so, the number of deadlines met was still slightly less than classical DM. Overall, the DM algorithm performs as well as or better than OSM-DM from both the system and task perspectives.

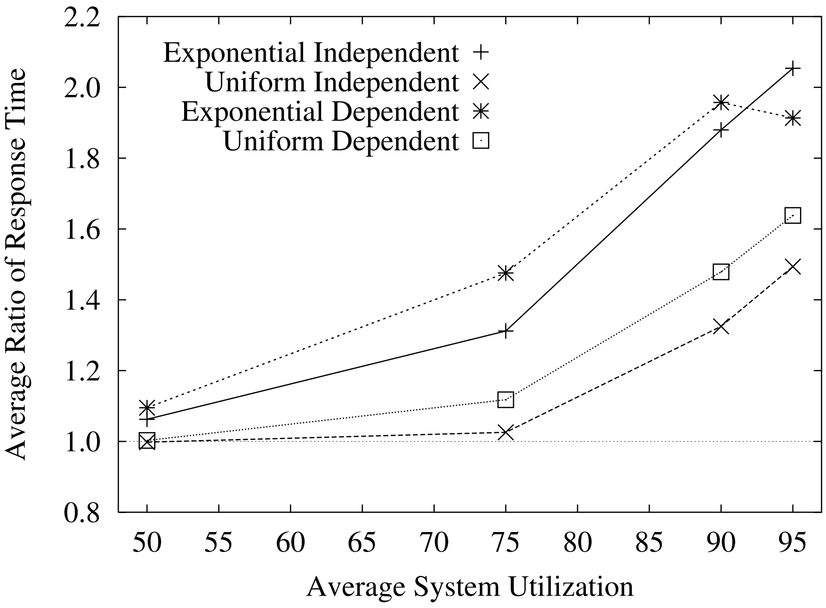
From the task perspective, OSM-DM met significantly more deadlines than DM, particularly in the single server configuration, but at the expense of greatly increased average response times. This is to be expected since OSM defers the execution of overrunning jobs causing their response times to increase. Inter-release time variations accentuate the problem because the non-overrunning portions are no longer guaranteed to meet their deadlines if clusters of releases occur.

A comparison of OSM-EDF with EDF shows that OSM-EDF meets many more deadlines than EDF from both the system and task perspectives. As Figures 6.28 and 6.29 show, the performance on workloads with exponential distributions was significantly better than with uniform distributions. Once again, a server per task performed better than a single server. Surprisingly, the distribution type had a greater effect on the performance than did the number of servers. Clearly, the variability resulting from the use of distributions with long tails for both release times and execution times dwarfs the differences caused by varying the number of servers.

Figures 6.30 and 6.31 compare the performance of the best configurations of OSM-EDF and OSM-DM, i.e., OSM-EDF with a server per task and OSM-DM with one server. Contrary to previously observed results, the best configuration of OSM-DM performs better than the best configuration of OSM-EDF on workloads

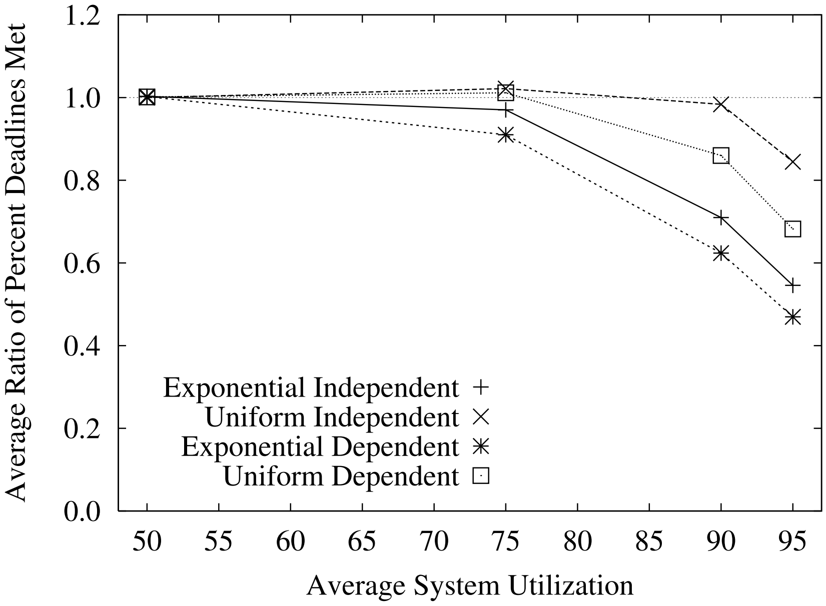


(a) Deadlines Met

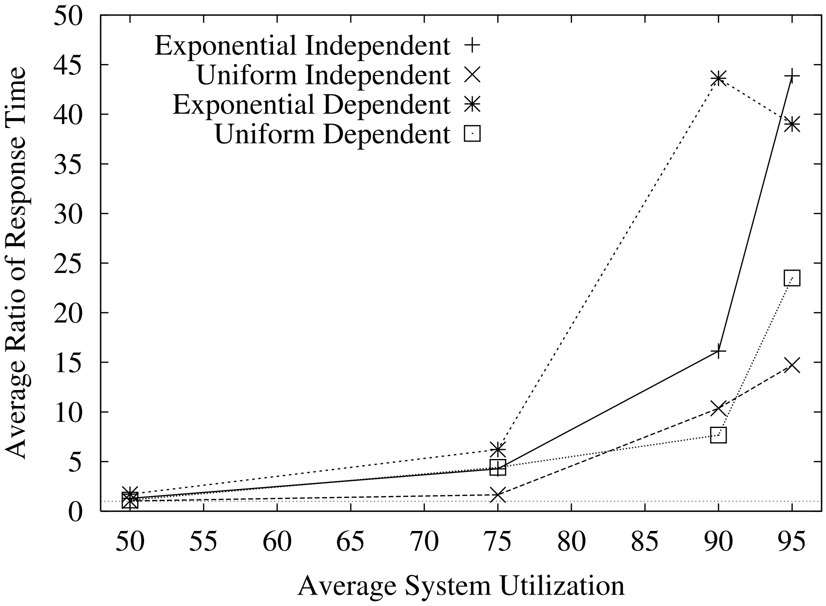


(b) Response Time

Figure 6.24: EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics of systems with a DM scheduler.

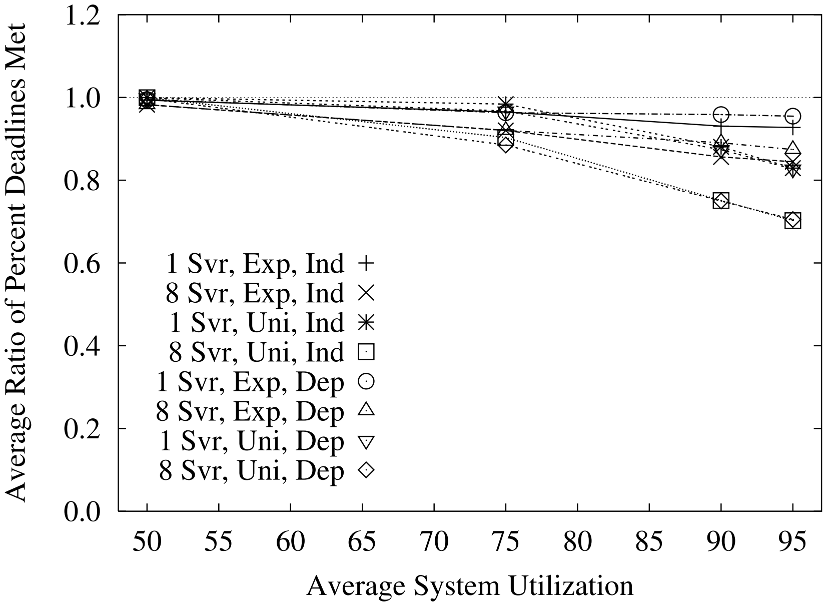


(a) Deadlines Met

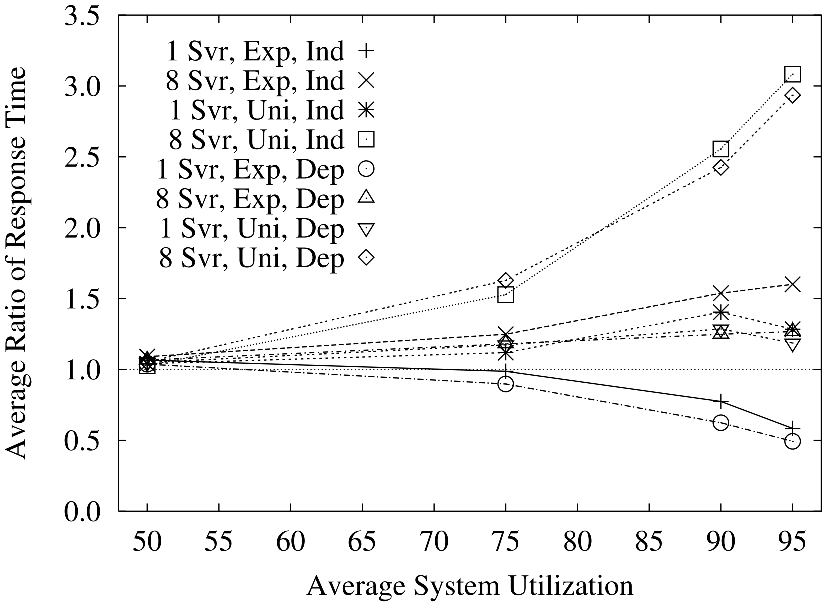


(b) Response Time

Figure 6.25: EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems with an EDF scheduler to the metrics of systems with a DM scheduler.

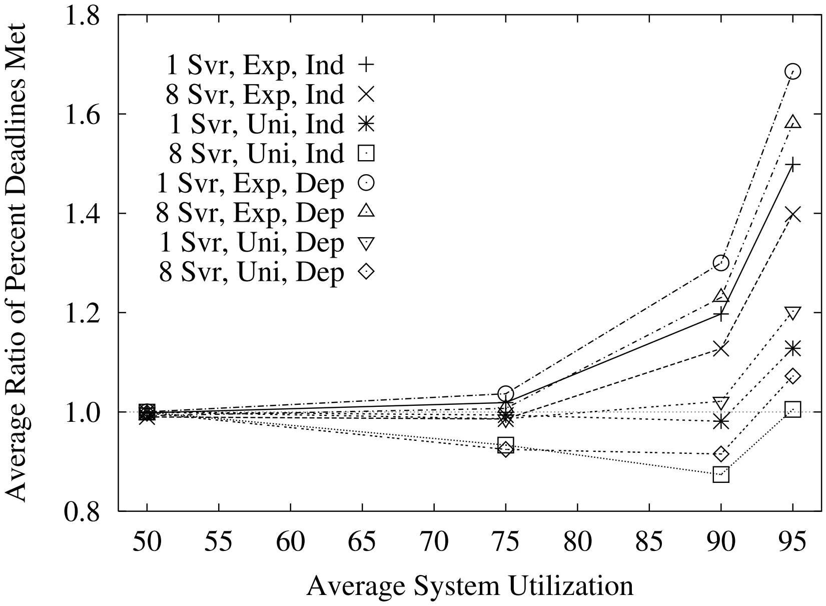


(a) Deadlines Met

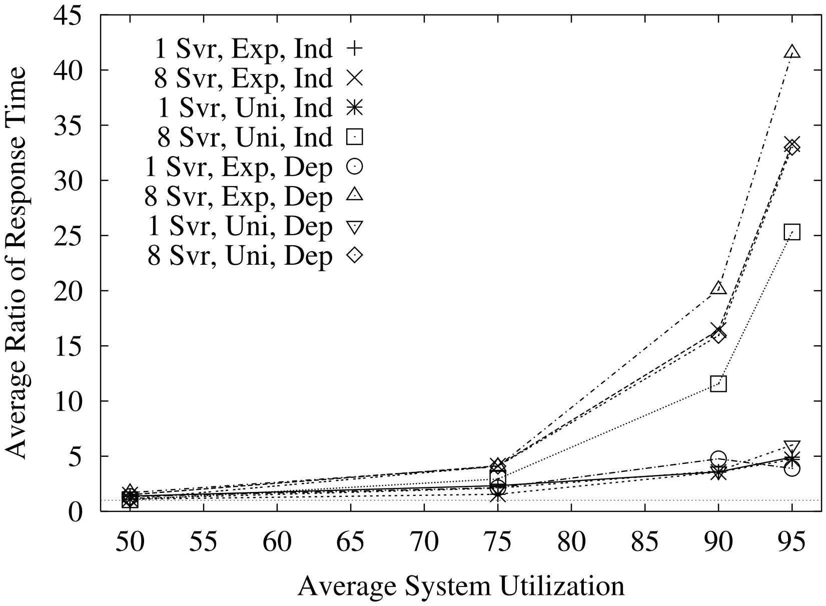


(b) Response Time

Figure 6.26: OSM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to DM.

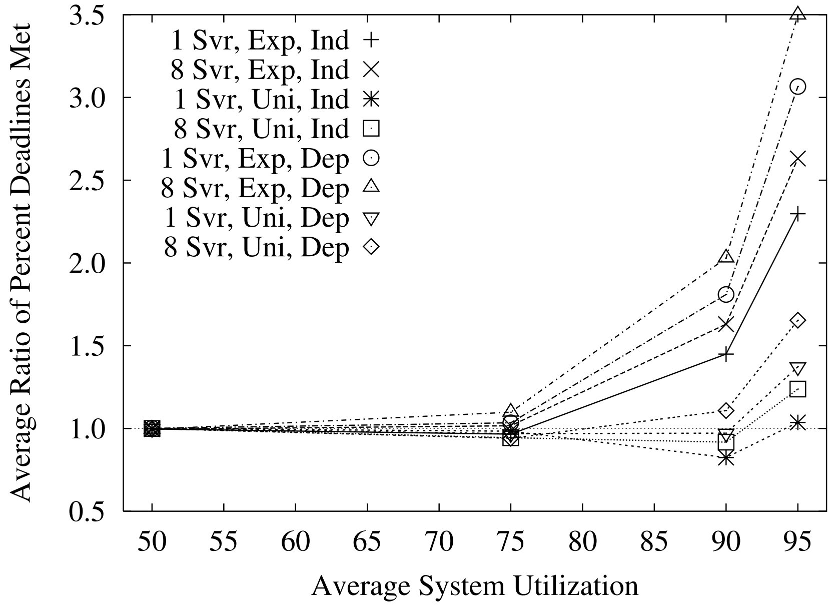


(a) Deadlines Met

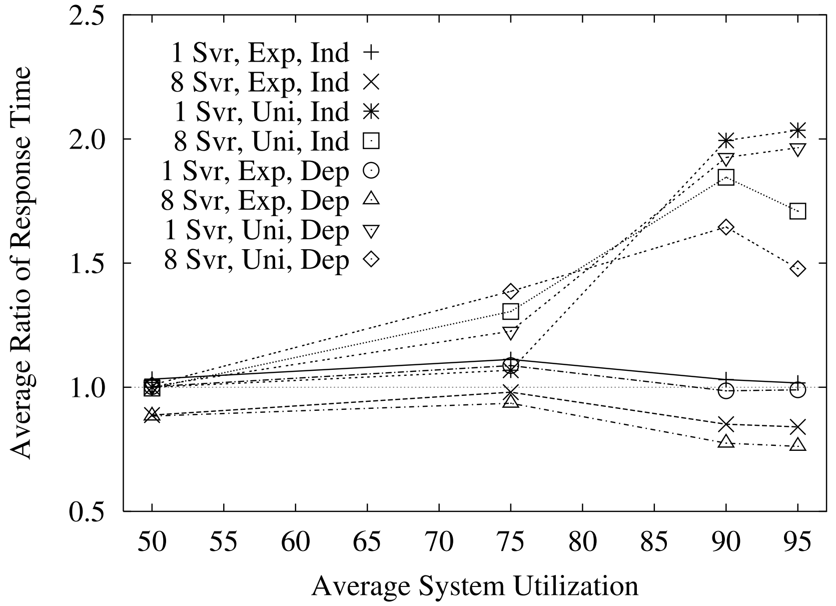


(b) Response Time

Figure 6.27: OSM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to DM.

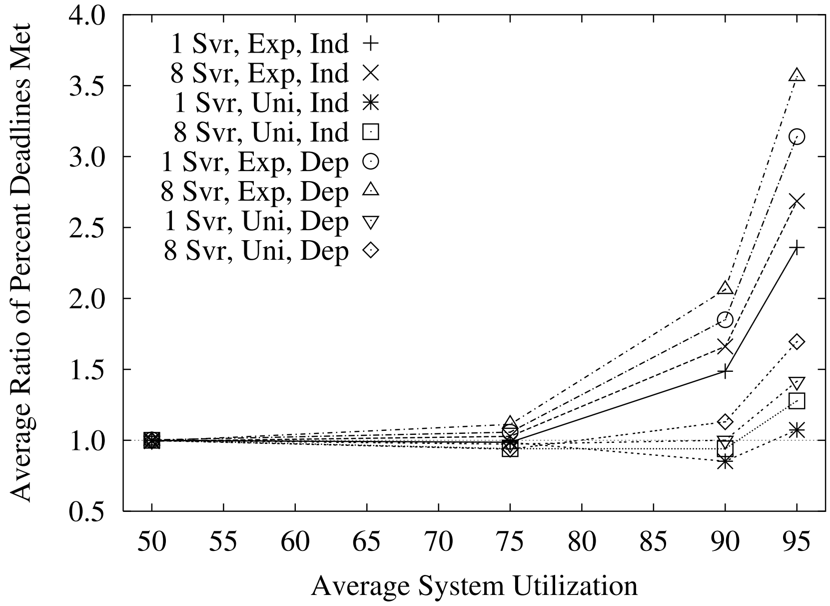


(a) Deadlines Met

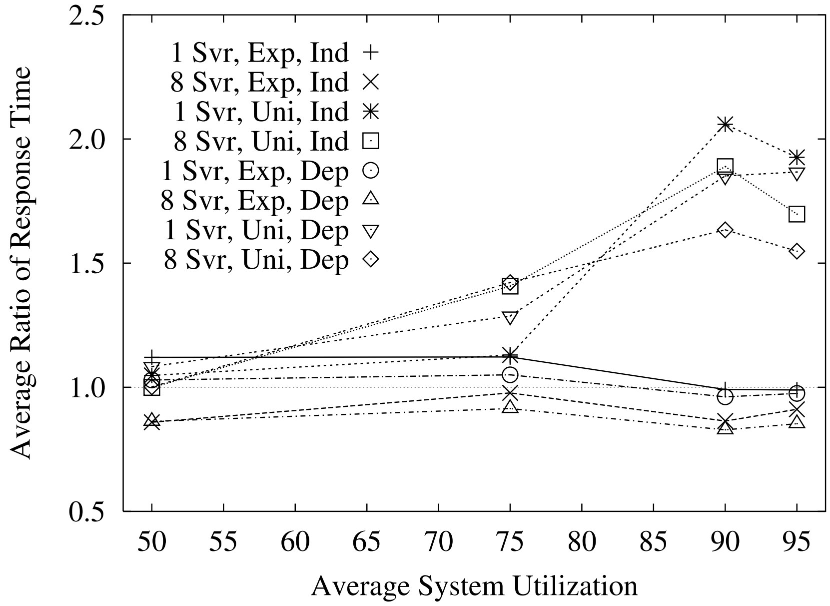


(b) Response Time

Figure 6.28: OSM-EDF vs. EDF from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to EDF.



(a) Deadlines Met



(b) Response Time

Figure 6.29: OSM-EDF vs. EDF from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-EDF to the metrics for systems scheduled according to EDF.

with both release time jitter and execution time variations. This result holds from both the system and the task perspectives. While the percentage of deadlines met differs by a slight but statistically significant amount, the average response times of OSM-DM are much better than OSM-EDF. (The width of the 95% confidence intervals are generally all well below 5%.) Therefore, we now compare OSM-DM to classical DM.

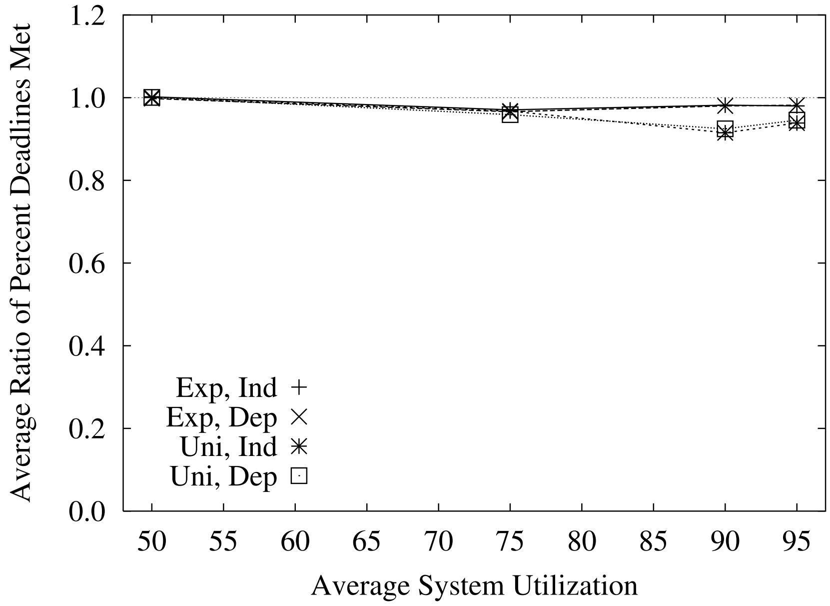
A comparison of OSM-DM to classical DM from the system perspective (Figures 6.32) shows that classical DM performs the best, with the exception of the average response time for the exponential workload and a single server for all tasks. Figure 6.33 shows that OSM-DM meets more deadlines than classical DM for exponential workloads and for uniform workloads at high utilizations. However, the average response times for OSM-DM are definitely inferior to classical DM.

6.6.3 Performance of ISM vs. Baseline

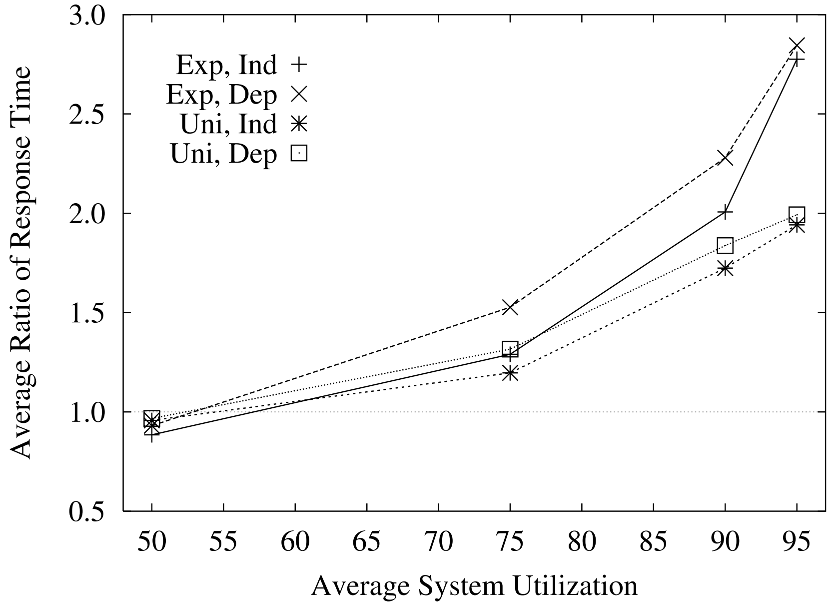
We now compare the performance of ISM-DM to DM and ISM-EDF to EDF for systems in which both the inter-release times and the execution times vary widely. As Figure 6.34 shows, the preferred configuration of ISM-DM from the perspective of the system is one server for all tasks. In this configuration, the number of deadlines met is only slightly less than classical DM. However, average response times are much better than for classical DM. From the perspective of a task (Figure 6.35), ISM-DM meets more deadlines than classical DM in either the single server or the server per task configurations, except for the uniform distribution and a server per task. Once again though, the average response times are all worse for ISM-DM than for classical DM. (Although the confidence intervals the data shown in Figure 6.35(b) are wide, the observed differences are statistically significant.)

Figures 6.36 and 6.37 compare ISM-EDF to classical EDF. While the single server configuration performs worse than classical EDF, the server per task configuration performs much better. (The confidence intervals for the single server data in Figure 6.37(b) exceed 5% at 50% average utilization, becoming less than 5% by 95% average utilization.)

We now compare ISM-EDF to ISM-DM. As Figures 6.38 and 6.39 show, ISM-EDF with a server per task generally performs better than ISM-DM in the same configuration, but worse in the single server configuration. ISM-EDF with a server

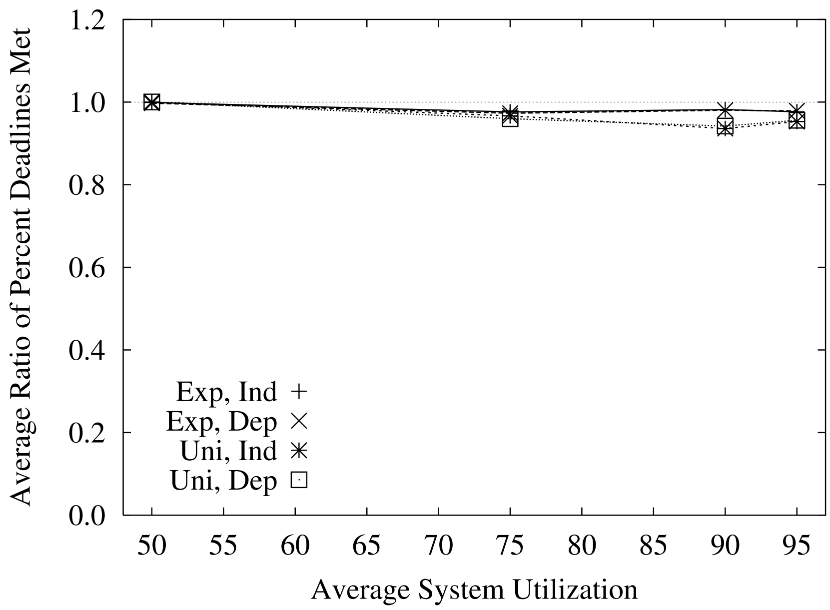


(a) Deadlines Met

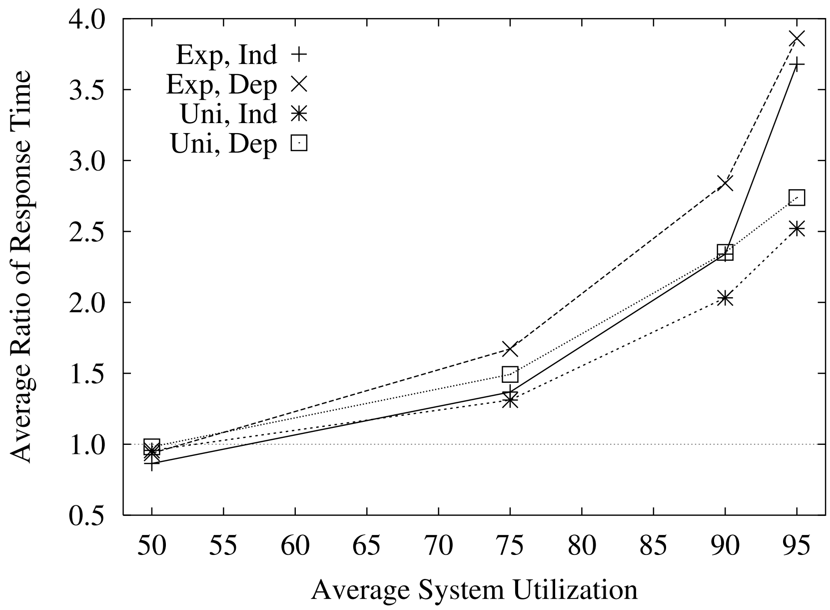


(b) Response Time

Figure 6.30: OSM-EDF Best vs. OSM-DM Best from System Perspective: the average of the ratios of the metrics for systems scheduled according the best OSM-EDF configuration to the metrics for systems scheduled according to the best OSM-DM configuration.

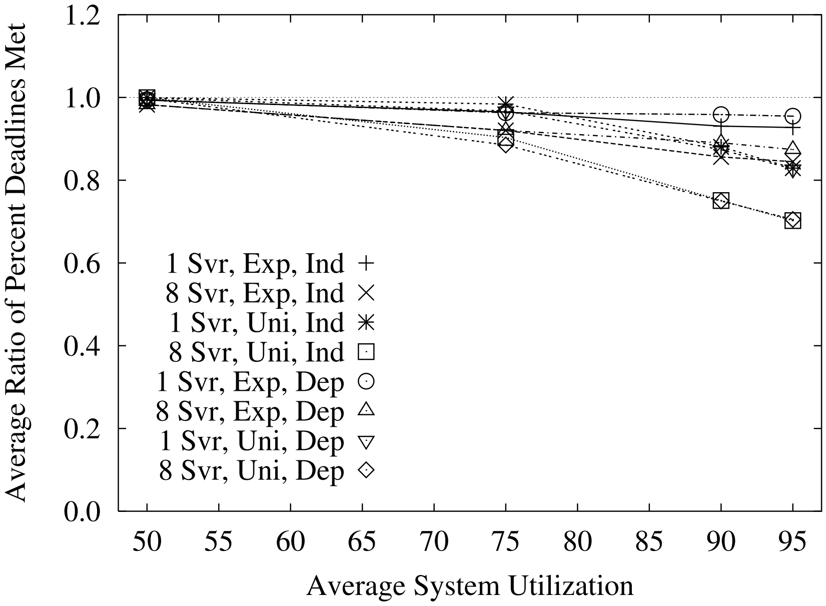


(a) Deadlines Met

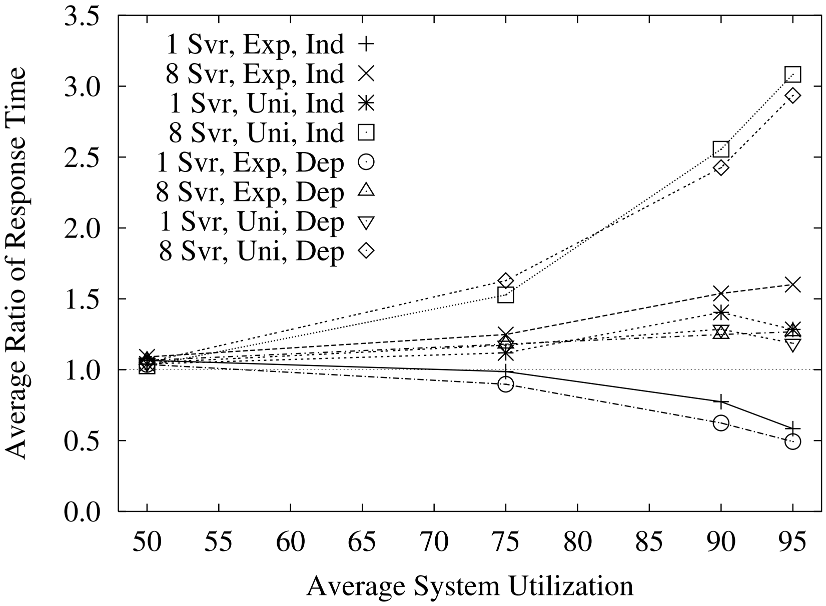


(b) Response Time

Figure 6.31: OSM-EDF Best vs. OSM-DM Best from Task Perspective: the average of the ratios of the metrics for systems scheduled according the best OSM-EDF configuration to the metrics for systems scheduled according to the best OSM-DM configuration.

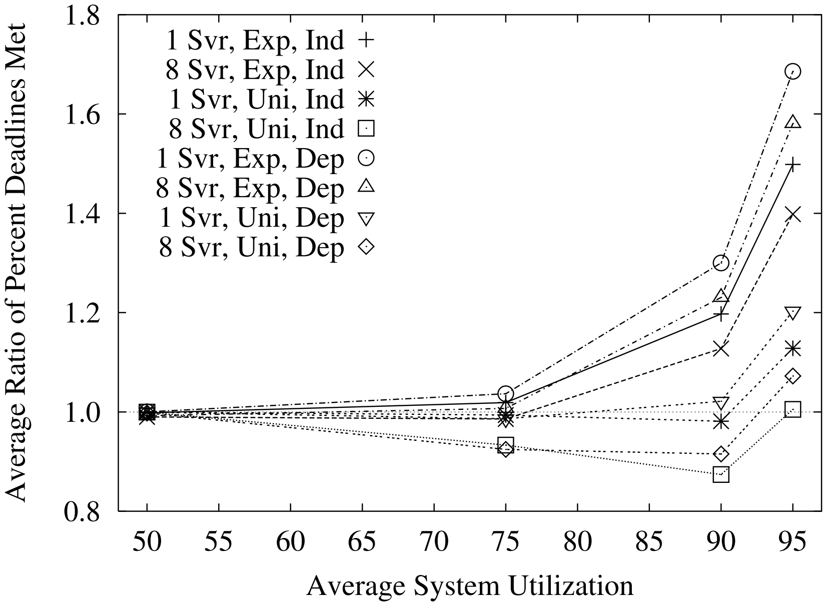


(a) Deadlines Met

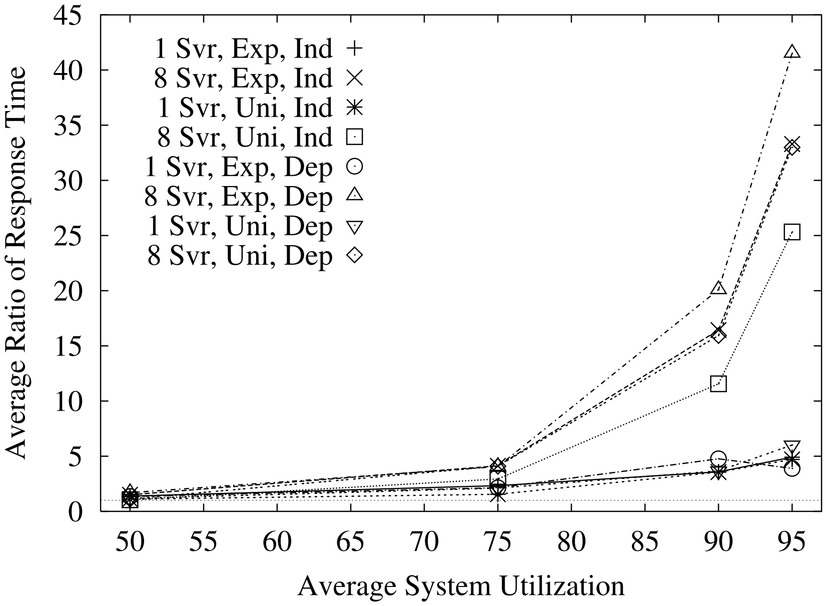


(b) Response Time

Figure 6.32: OSM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to Classical DM.

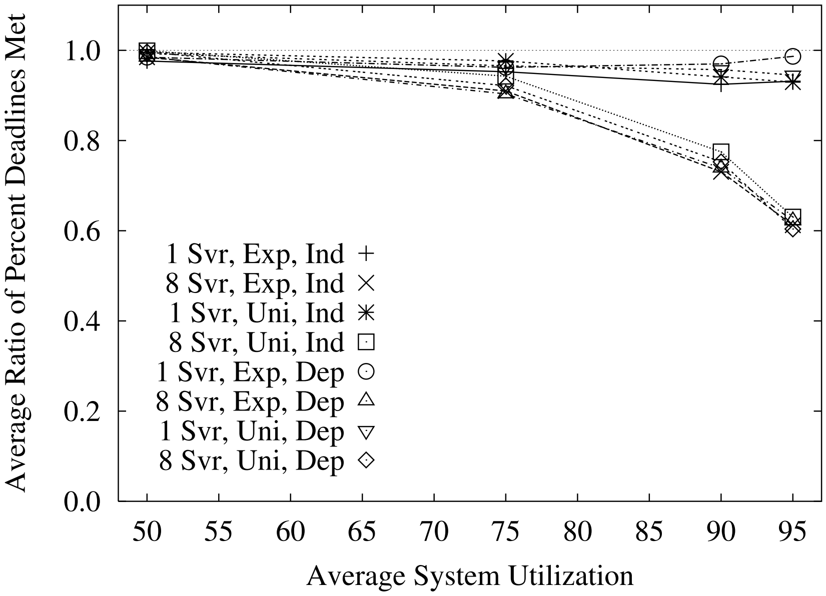


(a) Deadlines Met

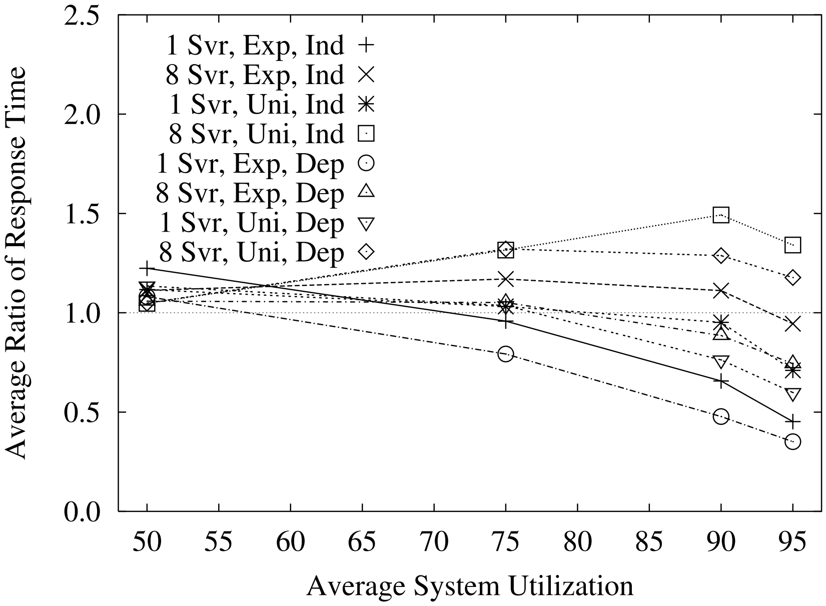


(b) Response Time

Figure 6.33: OSM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according OSM-DM to the metrics for systems scheduled according to Classical DM.

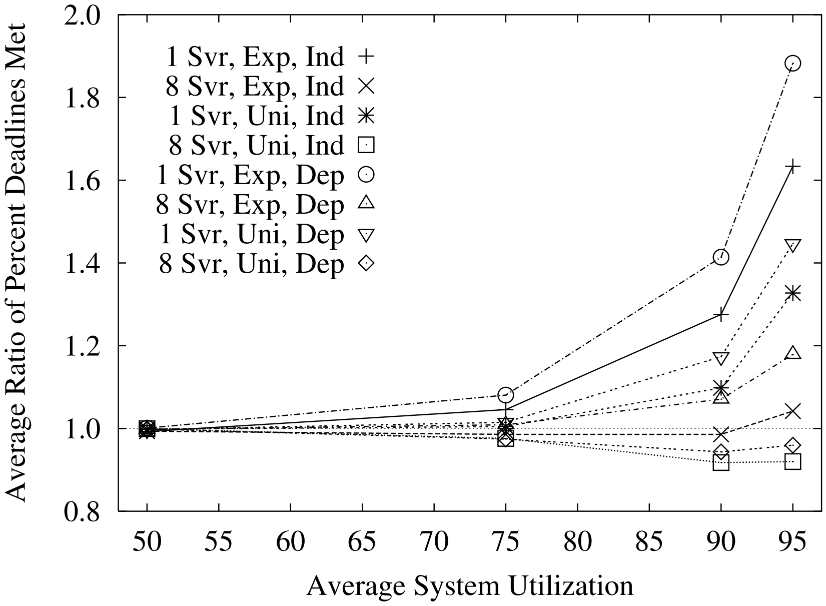


(a) Deadlines Met

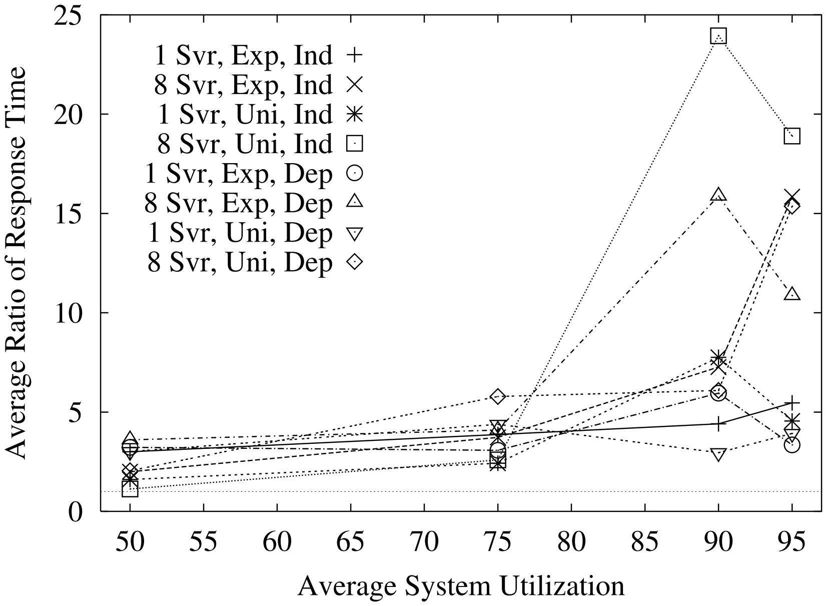


(b) Response Time

Figure 6.34: ISM-DM vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

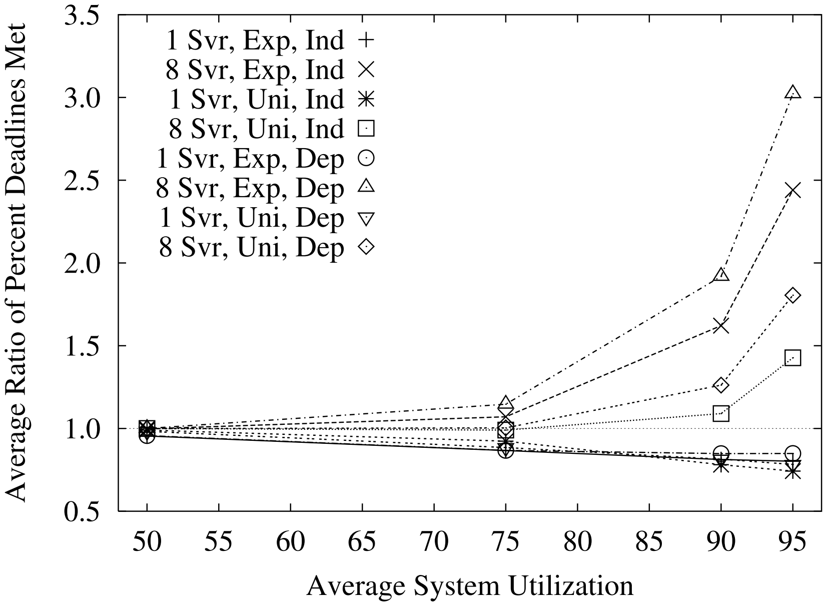


(a) Deadlines Met

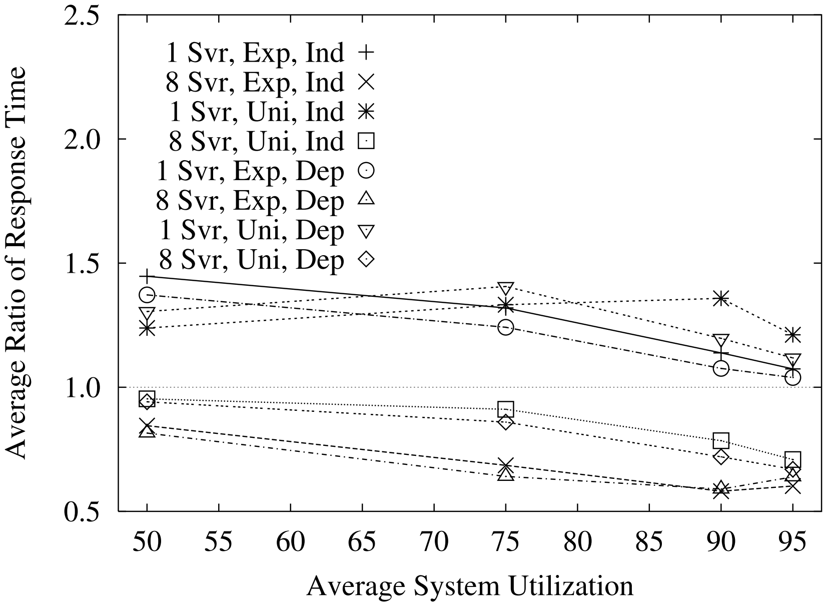


(b) Response Time

Figure 6.35: ISM-DM vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-DM to the metrics for systems scheduled according to DM.

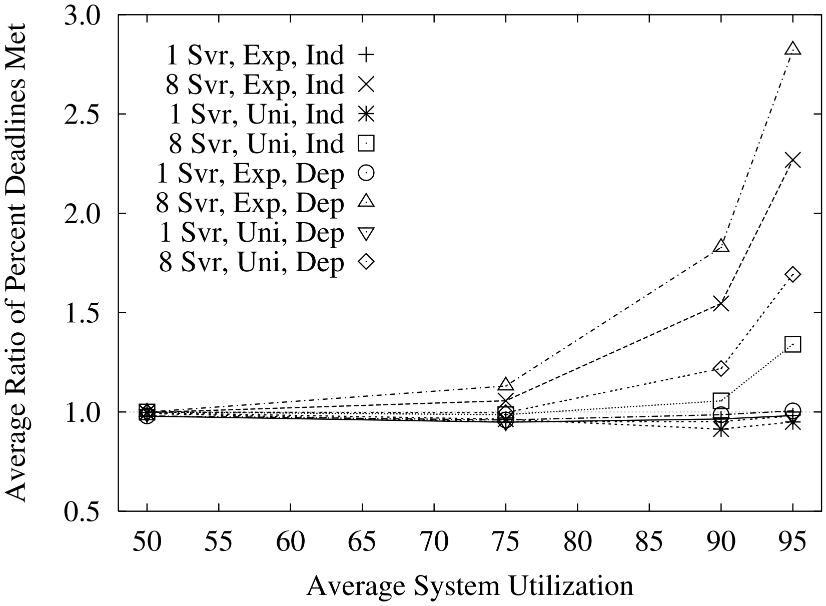


(a) Deadlines Met

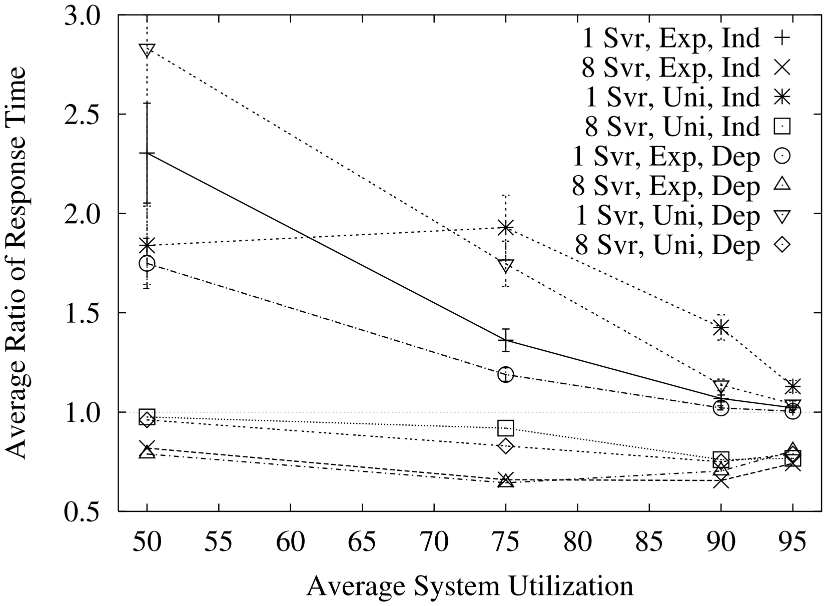


(b) Response Time

Figure 6.36: ISM-EDF vs. EDF from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.



(a) Deadlines Met



(b) Response Time

Figure 6.37: ISM-EDF vs. EDF from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to EDF.

per task clearly meets more deadlines from either the system or the task perspectives. It also has better better average response times, except at very high average utilizations. Therefore, ISM-EDF is generally preferable to ISM-DM.

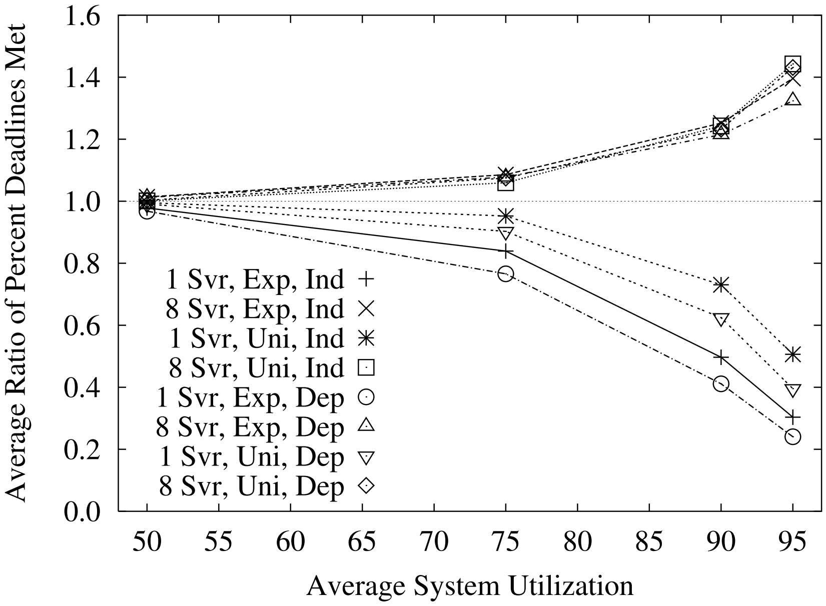
Finally, we compare the performance of ISM-EDF to the DM algorithm (Figures 6.40 and 6.41). ISM-EDF with a single server for all tasks clearly performs worse than classical DM from either perspective. It also performs slightly worse than classical DM from the system perspective in the server per task configuration. From the task perspective, however, the server per task configuration meets up to around 20% more deadlines at the cost of a dramatic increase in response times.

6.7 Summary

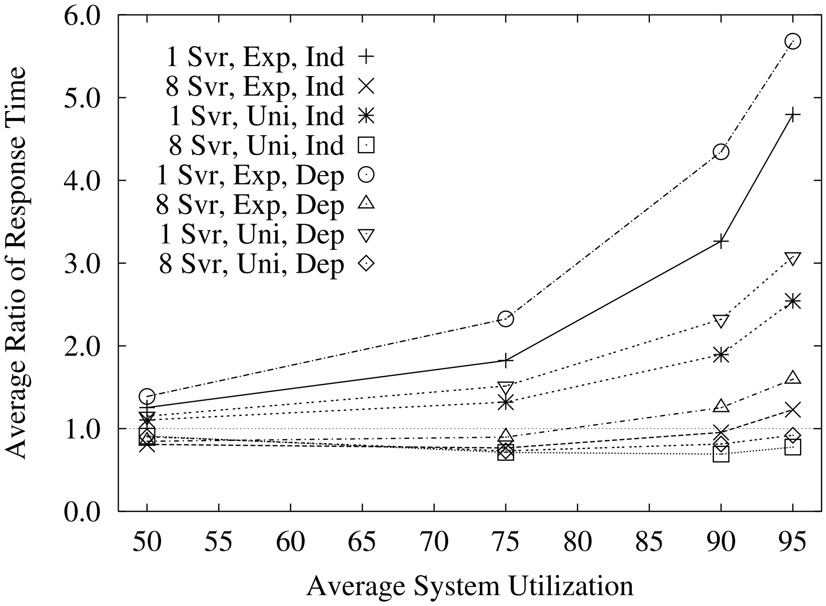
In summary, we have compared the performance of various scheduling algorithms on workloads with release time jitter, both with constant and varying execution times. The metrics used were the percentage of deadlines met and the average response time which can be computed from the perspective of the system or of the tasks in the system. The former view is useful for considering overall system performance, while the latter is important for the designers of critical soft real-time tasks. In general, the performance from the two perspectives differed less for workloads with release time jitter than they did for overrun.

The performance of only two classes of scheduling algorithms were compared when inter-release times varied but execution times were held constant: the classical fixed-priority and deadline-driven scheduling algorithms, characterized by the Deadline Monotonic and Earliest Deadline First algorithms, and the Isolation Server Method. (The performance of the Overrun Server Method was not considered as the execution times of jobs were constant and hence could not overrun. Therefore the performance of the Overrun Server Method should be identical to the performance of the classical algorithms.)

We also considered the effects of fairness by comparing the performance of Weighted Fair Queueing Servers (WFQS) with the performance of Total Bandwidth Servers (TBS) when scheduled according to ISM-EDF. In the configuration in which all tasks are assigned to a single server, the performance of a WFQS and a TBS are identical because the scheduling deadlines of jobs computed by each type of server are the same. In the server per task configuration, the performance

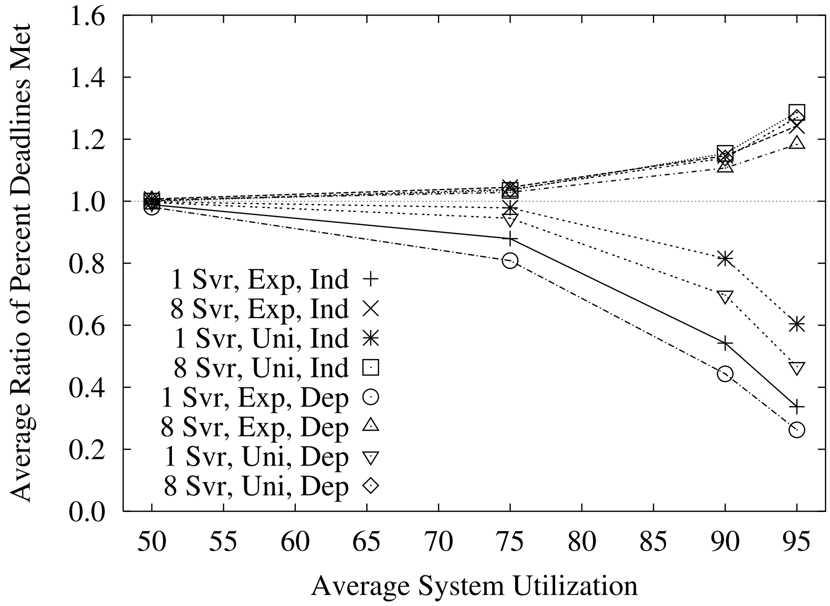


(a) Deadlines Met

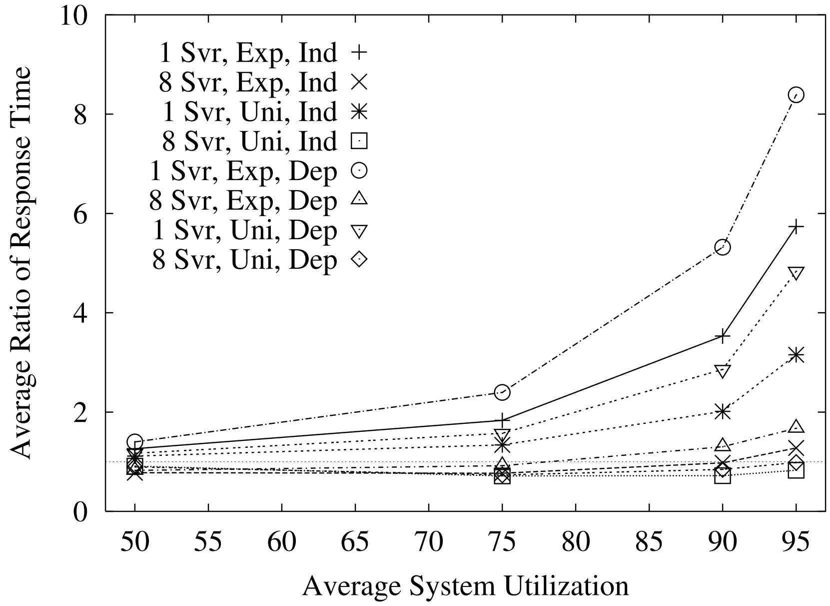


(b) Response Time

Figure 6.38: ISM-EDF vs. ISM-DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

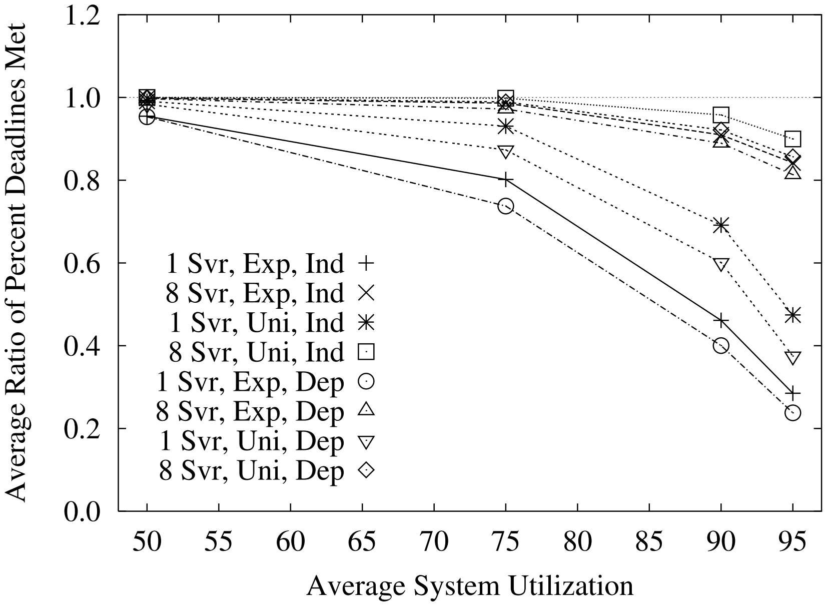


(a) Deadlines Met

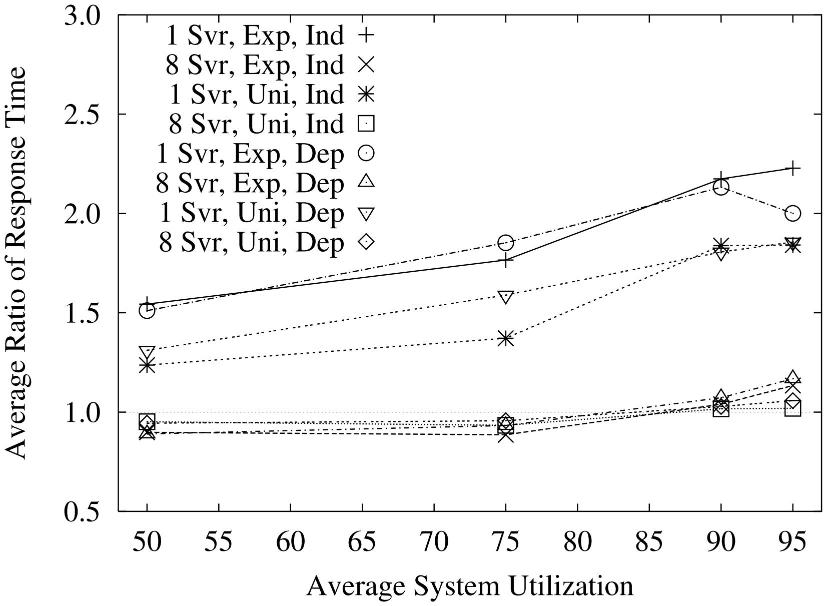


(b) Response Time

Figure 6.39: ISM-EDF vs. ISM-DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to ISM-DM.

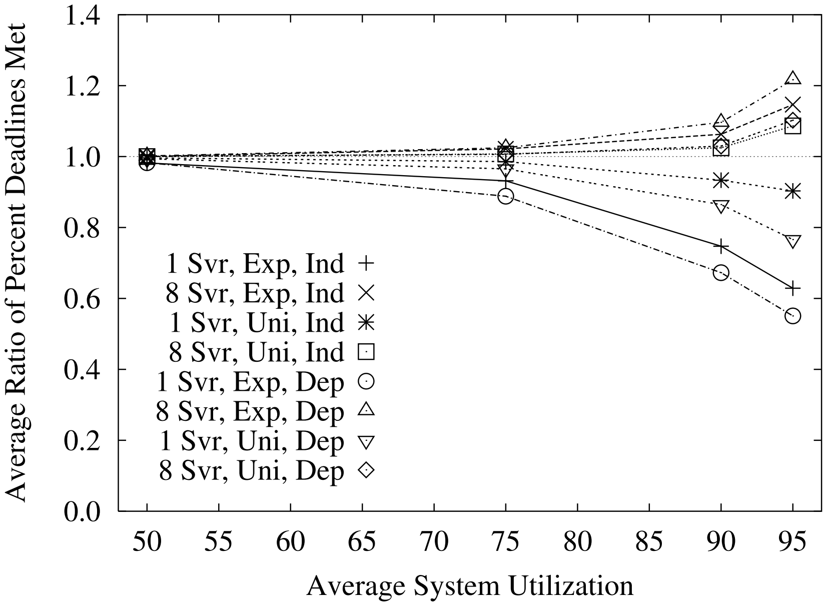


(a) Deadlines Met

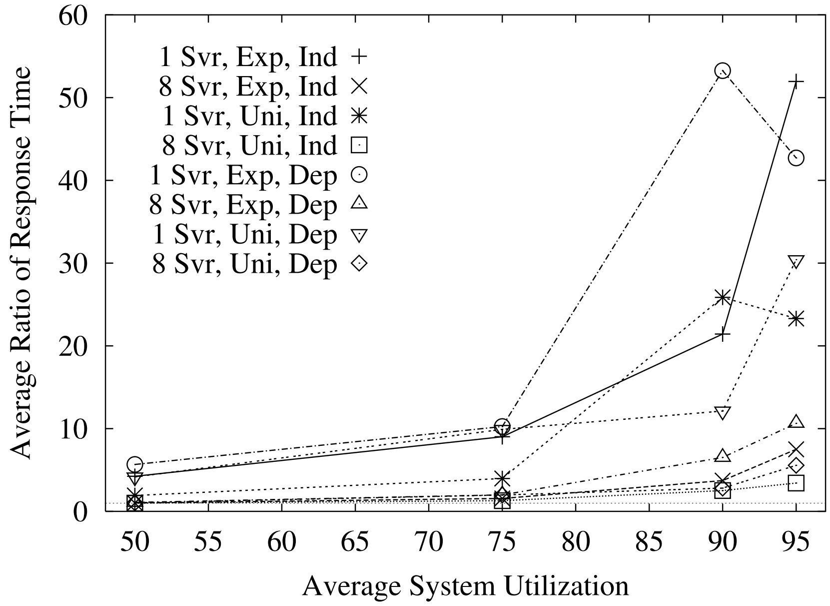


(b) Response Time

Figure 6.40: ISM-EDF vs. DM from System Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.



(a) Deadlines Met



(b) Response Time

Figure 6.41: ISM-EDF vs. DM from Task Perspective: the average of the ratios of the metrics for systems scheduled according ISM-EDF to the metrics for systems scheduled according to DM.

of TB servers was better than the performance of WFQ servers, particularly for high average system utilizations. This occurs because in the process of giving each busy server its fair share of the processor bandwidth a WFQS may schedule a job from a busy server (which has already missed its deadline) instead of a job released to an idle server (which is more likely to meet its deadline). We used TB servers for the remainder of the performance evaluations.

The Isolation Server Method performed best with a server per task scheduled according to EDF. From both the system and task perspectives, the performance was generally comparable but not equal to the best classical algorithm, DM. We investigated the effect of the maximum difference in inter-release times on performance. Decreasing the maximum period ratio causes ISM-EDF to perform better with respect to the DM algorithm, but the DM algorithm still met more deadlines and had lower average response times. We concluded that as long as priorities are correlated with release rate, a fixed priority assignment is likely to yield better performance.

We also compared the performance of the classical scheduling algorithms with the OSM and the ISM on workloads in which both the inter-release times and the execution times varied. Once again, ISM-EDF with a server per task performs better than either the fixed priority or deadline driven variants of the ISM and the OSM, but did not exceed the performance of classical DM in general.

From purely a performance point of view, highly overloaded systems should be scheduled using classical DM because of its predictability, especially from the perspective of the system. However, classical DM cannot guarantee that jobs which do not overrun their processor allocations will meet their deadlines like OSM can. Likewise, the DM algorithm does not meet as many deadlines from the perspective of a task as the ISM does. Thus in the wider context of critical soft real-time system design, the Overrun Server and Isolation Server Methods may still find use.

As with overload caused by overrun, the results indicate that all the methods perform similarly for average system utilizations below about 75% regardless of how the release times or execution times vary. Likewise, the presence or lack of dependencies had little effect below 75% average utilization. (A 75% average system utilization corresponds to a maximum system utilization of at least 150% for the distributions considered.) Thus, the choice of scheduling algorithm is probably not crucial for all but the most highly overloaded systems.

Chapter 7

Conclusions

In this thesis, we extended the periodic task model to allow systems of hard, soft and non-real-time tasks to be described in a uniform and seamless manner. We used the model to develop the Stochastic Time Demand Analysis method which allows fixed priority critical soft real-time systems to be designed for better resource utilization and average performance than is possible using deterministic real-time analysis methods. In particular, Stochastic Time Demand Analysis allows the effect of performance enhancing features, such as multi-level memory hierarchies in processors or statistical multiplexing in networks, to be accounted for in determining the probability that jobs of a task will meet their deadlines. When applied to applications which are allowed to miss a few deadlines, Stochastic Time Demand Analysis enables systems to be designed with higher average performance and lower cost.

Allowing a real-time system to miss some deadlines in exchange for better average performance increases the possibility of overload. We have compared the performance of various scheduling algorithms for critical soft real-time systems under overload conditions to determine which algorithms meet the most deadlines and provide the shortest average response times. The information assists the designers of critical soft real-time systems in the selection of the appropriate scheduling algorithm for the best performance.

7.1 Extended Periodic Task Model

In Chapter 3, we extended the periodic task model, to allow systems of hard, soft and non-real-time tasks to be described in a uniform and seamless manner. The extended model requires each task to be assigned a guaranteed execution time and a guaranteed inter-release time, with the constraint that the system be schedulable on the basis of these guaranteed parameters. If the execution time of a job is less than the guaranteed execution time of its task and the release time of the job is no earlier than the release time of its successor plus the guaranteed inter-release time of its task, then the system guarantees that the job will meet its deadline if higher priority tasks do not violate their guaranteed parameters.

The guaranteed execution time of a hard real-time task is equal to the maximum execution time of its jobs. The guaranteed inter-release time of a hard real-time task is equal to the minimum inter-release time of its jobs. In contrast, the guaranteed execution times of non-real-time tasks are equal to zero and the guaranteed inter-release times are equal to infinity. The guaranteed execution and inter-release times of soft real-time tasks (whether critical or not) have values which lie between the values for hard and non-real-time tasks. By appropriate choices for the guaranteed execution and inter-release times of the tasks in a system, we can describe systems containing hard, soft and non-real-time tasks within a single framework. This provides a flexible basis for design and analysis. We made use of this flexibility in the formulation of the Stochastic Time Demand Analysis method in Chapter 4 and in the performance comparison of algorithms for scheduling overload in Chapters 5 and 6. An early version of the extended model was published in [58].

7.2 Stochastic Time Demand Analysis

The Stochastic Time Demand Analysis method for bounding the frequency of missed deadlines in fixed priority systems makes it possible to account for the variability introduced by performance enhancing features during the design of critical soft real-time systems thereby achieving better average performance than would be possible using deterministic real-time analysis methods. It is used to bound the frequency of deadlines met by jobs in a task.

A lower bound on the probability that jobs in a task complete by their deadlines is computed by considering the time demand of each job, along with other jobs of equal or higher priority, in the interval of time from when the job is released until it completes. In other words, the time demand is considered from the perspective of the job being analyzed rather than from the perspective of the entire busy interval as in the Time Demand Analysis method [9]. The probability that a job completes by its deadline is the probability that the time between its release and its deadline is sufficient to meet the time demand of the system during the interval.

For systems in which the maximum utilization is less than 1.0, the probability that a busy interval will end is 100% and the worst-case time demand occurs when jobs are released in-phase. Even though we do not know what combination of release times leads to the worst-case time demand for jobs in a task when the maximum utilization exceeds 100%, simulation results support the use of in-phase busy intervals in analysis. We have not observed a case where the probability of completion of tasks released in-phase is higher than when the tasks are released with arbitrary phases. Thus, the minimum probability of timely completion of any job in a task in an in-phase busy interval is likely to be a lower bound on the probability that all jobs in the task will complete in time. An early version of these results were published in [70].

In many real-time systems, tasks share resources. To maintain the integrity of the system, access to shared resources often needs to be controlled. This is frequently accomplished by ensuring that a job holding a resource cannot be preempted until the resource is released. Non-preemption, however, gives rise to priority inversion where a high priority job waits while a low priority job executions in a critical section. Blocking delay due to priority inversion can be accounted for in Stochastic Time Demand Analysis by increasing the time demand at the beginning of an in-phase busy interval by the maximum delay caused by critical sections in lower priority tasks. If critical sections are of short duration, as they typically are since long duration critical sections reduce performance, adding the maximum blocking delay is not overly pessimistic. We also outlined a more complex approach for systems with long duration critical sections or ones in which the durations vary widely.

Finally, we have shown how Stochastic Time Demand Analysis can be used, along with the Release Guard Protocol [25], to compute the probability of meeting end-to-end deadlines in distributed systems. The Release Guard Protocol ensures that the separation between the releases of two instances of a subjob in an end-to-end task is at least as large as the guaranteed inter-release time. Assuming that the phase of the first instance of a subjob in a busy interval is equal to the maximum response time of its successor in the job, the response time distribution of an end-to-end job is computed via a straight forward application of Stochastic Time Demand Analysis. The probability of jobs in a task meeting their end-to-end deadlines is determined from the end-to-end response time distribution. A more precise bound can be computed by taking into account the response time distribution of the first subjob as well. This leads to a more complex approach which requires conditioning on an event whose time of occurrence is itself a random variable.

7.3 Scheduling Overrun and Jitter

In this thesis, we also characterized the performance of scheduling algorithms for overloaded systems. A system is overloaded as when it is not schedulable according to deterministic real-time theory on the basis of maximum execution times and minimum inter-release times and hence some jobs may miss their deadlines. We considered the effects of both execution and inter-release time variations. The metrics used to compare the algorithms were percent deadlines met and average response times. The metrics can be computed from the perspective of the system (i.e., the performance of each job has equal weight) or from the perspective of the tasks in the system (i.e., the average performance of each task has equal weight). The former is useful for considering overall system performance, while the latter is important for designing real-time tasks.

Specifically, we proposed two new classes of algorithms for scheduling overloads due to overruns. We also compared the performance of the two classes with a baseline class of scheduling algorithms on workloads with independent and dependent execution times. The baseline class contains the Deadline Monotonic and Earliest Deadline First scheduling algorithms. In the second class of algorithms, called the Overrun Server Method, jobs are initially scheduled according to one of the baseline algorithms. If a job overruns it is interrupted and its remaining part is submitted as an aperiodic request to a server. We used a Sporadic Server to execute overruns in fixed priority systems and either a Total Bandwidth Server or a Constant Utilization Server in deadline-driven systems. In the final class of algorithms, called the Isolation Server Method, jobs of a task are released as aperiodic requests to the server assigned to the task. An overrunning job can only delay the completion of other jobs assigned to the same server, effectively isolating the jobs of one server from the jobs of other servers. Preliminary results were published in [58].

The average utilizations of the workloads used in the study ranging from 50% to 95%. We also considered the effects of fixed correlations between consecutive jobs in a task since the underlying causes of overload are likely to cause dependencies. Generally speaking, we found that fixed priority scheduling performed better than deadline-driven scheduling when judged from the system perspective since higher priority jobs also release more jobs. A comparison of the classical scheduling algorithms showed that the Deadline Monotonic algorithm performs much better than the Earliest Deadline First algorithm under overload.

The comparisons indicated that the Deadline Monotonic algorithm performs better than the Overrun Server Method for fixed priority systems from either the system or the task perspective. For deadline-driven systems, the Overrun Server Method performs better than the Earliest Deadline First algorithm for exponential workloads, particularly with a server per task. Otherwise, the Earliest Deadline First algorithm performs best. The best performance of the Overrun Server Method occurred under the deadline-driven scheduler when individual overrun servers were assigned to each task. A comparison of Constant Utilization Servers and Total Bandwidth Servers showed that the latter yields dramatically better performance through the use of background time. The comparison also showed that Constant Utilization Servers should not be used for systems with high average utilization because the overhead of wait queue management grows until it overwhelms the scheduling of jobs.

A comparison between the Isolation Server Method and the classical scheduling algorithms showed that the Deadline Monotonic algorithm had the best performance for fixed-priority systems from either perspective. In deadline-driven systems, however, the Isolation Server Method with a server per task had the best performance on the dependent workloads, particularly for high average system utilizations and from the perspective of the tasks. To further characterize the performance of the the Isolation Server Method under overload, we conducted simulations using execution time traces obtained from an actual application and compared the results with the performance of the Earliest Deadline First and Deadline Monotonic scheduling algorithms. Once again, the Deadline Monotonic scheduling algorithm performed better than the Earliest Deadline First algorithm from either perspective. From the system perspective, Deadline Monotonic met more deadlines than the deadline-driven Isolation Server Method but less from the task perspective. The response time results for the algorithms were exactly opposite with the Isolation Server Method having better average response times from the perspective of the system and Deadline Monotonic having better average response times from the perspective of the tasks.

We considered the effects of release time jitter on the performance of the three classes of scheduling algorithms. Initially, the execution times of jobs in a task were constant. We only compared the Isolation Server Method with the baseline algorithms because the performance of the Overrun Server Method is identical with the baseline when jobs do not overrun. We noted that fairness becomes an issue when inter-release times vary and defined a variant of the Isolation Server Method which used Weighted Fair Queueing Servers instead of Total Bandwidth Servers. With a single server for all tasks, the operation of a Weighted Fair Queueing Server and a Total Bandwidth Server are identical. Because of improved fairness, however, it was expected that the performance of the Isolation Server Method with multiple Weight Fair Queueing Servers would be better than the performance of the Isolation Server Method with multiple Total Bandwidth Servers. Surprisingly, the Isolation Server Method with a Total Bandwidth Server per task performed better, from either perspective, than the Isolation Server Method with a Weighted Fair Queueing Server per task. Because of the way in which scheduling deadlines are assigned to jobs to improve fairness, a job with a high probability of missing its deadline can be scheduled by a busy server before a job which is released to an idle server even though the latter has a higher likelihood of completing in time, especially for heavily loaded systems. Because of its performance advantage, we selected the Total Bandwidth Server for use in the remainder of the simulations.

A comparison of the Earliest Deadline First and Deadline Monotonic algorithms once again showed that the latter performs better. While the deadline-driven Isolation Server Method performed better than the Earliest Deadline First algorithm in general, its performance was largely inferior to Deadline Monotonic scheduling indicating that the Deadline Monotonic algorithm is generally preferable for scheduling systems with release time jitter. One reason is that the task with the highest release rate also has the highest priority with the Deadline Monotonic algorithm (assuming the relative deadline equals the guaranteed inter-release time) and hence the results are biased in its favor. Reducing the maximum period ratio caused performance of the deadline-driven Isolation Server Method to improve relative to the Deadline Monotonic algorithm. However, the Deadline Monotonic algorithm still performed better suggesting that the tasks with the highest release rate should have the highest (fixed) priority.

When both execution times and inter-release times were varied, the Deadline Monotonic algorithm once again performed better than Earliest Deadline First algorithm. It also performed better than the fixed priority Overrun Server Method. Although the deadline-driven Overrun Server Method performed better than Earliest Deadline First algorithm, it did not perform better than the Deadline Monotonic algorithm. The best performance of the Isolation Server Method occurred for the deadline-driven variant with a server per task. Even though it performed better than Earliest Deadline First algorithm, it did not perform better than Deadline Monotonic algorithm except in limited situations.

Based on the simulations performed, the Deadline Monotonic scheduling algorithm performed the best overall and hence should generally be used when overload is possible. However, there are non-performance reasons for choosing one of the other methods. For example, the Overrun Server Method is the only scheduling algorithm that can guaranteed that a job which does not overrun will meet its deadline (as long as the guaranteed inter-release times of tasks equal the minimum inter-release times). As such, it should be used for systems in which hard guarantees are required. Furthermore, the Isolation Server Method met significantly more deadlines from the task perspective than the other methods, especially when the workload exhibited dependencies. Therefore the Isolation Server Method is preferable to the Deadline Monotonic algorithm if meeting deadlines is more important than minimizing average response times.

Finally, the results indicated that all the methods perform similarly for average system utilizations below about 75% regardless of the distribution of execution times and the presence or lack of dependencies. (A 75% average system utilization corresponded to a maximum system utilization of at least 150% for the distributions considered.) Thus, the choice of scheduling algorithm is probably not crucial for all but the most highly overloaded systems.

7.4 Future Work

While Stochastic Time Demand Analysis improves our ability to predict the behavior of critical soft real-time systems, it is restricted to fixed priority assignments. Similar techniques need to be developed for systems with dynamic priority assignments, such as those scheduled Earliest Deadline First. The probability that consecutive jobs will miss their deadlines also needs to be computed as many critical soft real-time applications cannot afford to miss more than a certain number of deadlines in a row. Finally, the behavior of systems in which execution times are dependent across tasks or jobs have precedence constraints between them also needs to be considered.

The performance comparisons in Chapters 5 and 6 either assumed that jobs were independent or that correlations could be modeled as fixed dependencies between consecutive jobs in a task. More realistic correlations need to be considered, including probabilistic dependencies which span variable numbers of (possibly) non-consecutive jobs in different tasks. Finally, the performance of servers employing the TB* algorithm [71], a successor to the Total Bandwidth Server algorithm, needs to be considered.

Appendix A

Simulation Environment

Chapters 5 and 6 compare the performance of various algorithms for scheduling overloaded systems. The performance was obtained by discrete event simulation using an object-oriented simulation framework developed prior to initial explorations of the topics. In this appendix, we discuss the hardware and software used to obtain the simulation results and make particular mention of things that we would do differently.

A.1 Simulation Framework

The simulation framework follows an object-oriented design and is implemented in the hybrid object-oriented language Java. The design is a straight forward modeling of the basic entities in a real-time system (e.g., jobs, tasks, schedulers and events). The framework consists of 244 files totaling 24,956 lines (including white space and comments) containing 247 classes and 2 interfaces with 852 methods and 400 variables. Of the classes, 66 are used for testing or demonstration purposes. Some of the most important classes are listed in Tables A.1, A.2, A.3, and A.4.

The code that must be written to perform a new simulation consists primarily of a class with a `main` class method. The `main` class method creates an instance of the `Processor` class (or a user-specified subclass thereof) with the appropriate scheduler queue discipline. Next, the tasks of the system and their initial release events are created and the events are added to the event queue of the `Processor`. Finally, the `Processor` event loop is entered and the simulation runs until a `Doomsday` event occurs or until the event loop runs dry. The default behavior of each event's `occur` method (also called a handler) only performs the

Table A.1: Major Classes of the Simulator.

Event	Queue Discipline
Event Queue	Random Number Generator
Exponential Random Variable	Random Variable
Job	Server
Job Queue	Task
Processor (Scheduler)	Uniform Random Variable

Table A.2: Major Subclasses of Simulator Events.

Blocked	Deadline Missed	Preemption
Budget Consumed	Delayed Release	Ready
Budget Replenishment	Enter NPS	Release
Budget Reset	Exit NPS	Resume
Completion	Idle	Schedule
Deadline Met	Overrun	Server Release

Table A.3: Subclasses of Simulator Queue Disciplines.

Deadline Monotonic
Earliest Deadline First
First-Come First-Served
Rate Monotonic
Shortest Job First
Shortest Time Remaining at Overrun
Shortest Remaining Time

Table A.4: Subclasses of Simulator Servers.

Constant Utilization Server
Sporadic Server
Total Bandwidth Server
Weighted Fair Queueing Server

actions required for proper simulator operation; events of interest to a simulation must be subclassed and their `occur` methods overridden to provide the desired behavior. (The overridden method must include calling the default method.) Typically, the overridden handler prints an appropriate message to `stdout` for post-processing. For the simulations in this work, the released, completion, deadline met and deadline missed event handlers were overridden to print the time an event occurred, the job involved and the task of the job. From this, the response time and percentage of deadlines met were computed.

At the time that work began on the framework, "Ahead-of-Time" (conventional) or "Just-In-Time" (JIT) compilers did not exist. The only tools available were bundled with Sun Microsystems's Java Development Kit (JDK). Of these tools, the bytecode compiler, `javac` and bytecode interpreter, `java` (an implementation of the Java virtual machine) were immediately useful. Early in the development of the framework it became apparent that the JVM was no speed demon. Simple tests indicated that executing simulations under the JVM would result in a 20–30 fold loss in speed over natively compiled code from languages such as C++. However, it also became apparent that development time would be reduced by using Java instead of C++ because of features such as garbage collection, less complicated semantics, a hierarchical name space for class identifiers and the unification of class declarations and definitions[1]. Luckily, several options for speeding up the execution of programs written in Java became available before the framework was targeted toward serious research. The option chosen was the Toba compiler, developed at the University of Arizona, which compiles and links classes into stand alone executables by way of intermediate files containing C source. The biggest advantage of this approach over JIT compilers is the small memory requirements resulting from not having to have a class loader or JIT compiler resident. It was estimated that the Toba compiler reduced execution times to within 25% of C++. Since that time, several other ahead-of-time compilers have been or are being developed. One is the GNU Java Compiler (GCJ) in the GNU Compiler Collection (GCC) from the Free Software Foundation[2]. One

[1]In general, C++ requires classes to be declared separate from their definitions. (The exception being inline definitions of functions in class declarations.) Java classes are defined by their declarations hence there is a single source which serves both purposes. Personal experience indicates that having a single source reduces code size, eliminates synchronization errors between declarations and definitions, and generally simplifies development and maintenance.

[2]The initial public release of GCJ occurred on July 31, 1999 in version 2.95 of GCC. It is available from `<http://egcs.cygnus.com/>`. The runtime library is currently available separately from `<http://sourceware.cygnus.com/java/>`.

of the expected advantages of GCJ is better performance as the result of more optimizations performed by the GCC back end. Another advantage is that object files from various languages, including Java, C and C++, can be incorporated into one stand alone executable.

The task of invoking the simulator for various configurations was automated by a set of Perl scripts which also created directories for the results and provided a visual indication of progress by writing a character to the screen before the simulation of each system. Perl scripts were also used to compute the percentage deadlines met and average response times of each task or for the system as a whole. In addition, Perl scripts were used to compute descriptive statistics about the simulations and to compute distributions of random variables. Perl proved to be a very good language for these purposes since it excels at orchestrating program invocations and at string handling.

A.2 Hardware

As mentioned in the experimental methodology of the previous chapters, each line on a graph represents the ratio of the performance of one algorithm to another for a given workload and consists of data points for four different utilizations. In general, each data point required the simulation of 100 system configurations for two different algorithms. While in the process of computing preliminary results [58], it was discovered that simulating all the systems for a single data point took several hours to complete. Overall, many months of continuous simulation time would be necessary. Because each data point is independent of the others, the series of simulations represented a potentially large amount of coarse grained parallelism which we set out to exploit.

Table A.5 lists the machines which were ultimately assembled to perform the simulations. (One of the machines, rtsl7.cs.uiuc.edu, was only available for four weeks while a colleague was on vacation.) Except for the two RTSL machines, the remainder were borrowed from colleagues in the Concurrent Systems Architecture Group. All but one of the machines were connected to a single monitor, keyboard and mouse using a Raritan console switch. The machines were connected to each other (and to the outside world) via a 10base2 Ethernet hub.

The operating system was RedHat Linux v5.2 upgraded to kernel version 2.2.3 with SMP extensions enabled so that both processors could be used on the dual

Table A.5: Machines Used for Simulations.

Machine	Manufacturer	Processor	Clock Speed (MHz)	RAM (MB)	Disk (GB)
rtsl3	Gateway 2000	Pentium Pro	200	64	3 IDE
rtsl7	Gateway 2000	Pentium Pro	266	64	8.5 IDE
stda1	HP Vectra	Pentium Pro Pentium Pro	200 200	64	9 SCSI
stda2	HP Vectra	Pentium Pro Pentium Pro	200 200	64	9 SCSI
stda3	HP Vectra	Pentium Pro Pentium Pro	200 200	64	9 SCSI
stda4	Dell Optiplex	Pentium Pro	200	64	2 SCSI
stda5	Dell Optiplex	Pentium Pro	200	64	2 SCSI
stda6	Dell Optiplex	Pentium Pro	200	64	2 SCSI
stda7	Dell Optiplex	Pentium Pro	200	64	2 SCSI

processor machines. Most of the computations required only local use of the disk for temporary files but the final results were stored on a 2 GByte partition which was NFS mounted to each machine from rtsl3.cs.uiuc.edu. Even though only a small portion of the generated data was ever stored to disk, the archived and compressed data, including the input data sets, totalled nearly 1.0 GByte in 68,799 files. Uncompressed, the results consume nearly 7.9 GBytes of disk space in 1,220,367 files. Because the average size of the result files was small, the i-nodes on the 2 GByte partition were frequently exhausted before all the disk space was consumed. The problem was exacerbated by the relatively small size of the archive partition. Each time the i-nodes were exhausted, the simulator would halt but with manual intervention the simulations had could be restarted where they left off. The solution was to archive and compress the data to conserve i-nodes even though this made data reduction less convenient.

An anticipated problem was saturation of the network bandwidth. Because temporary results were cached locally on each machine, comparatively little data was sent to the server. The only time that saturation became significant was during data reduction when each processor would read pairs of summary files for each system of a workload. The time required for comparison would increase from approximately 15 minutes with one machine to 30–45 minutes with all 12 machines. Surprisingly, the dual processor HP machines always completed the comparison computations more quickly than did the single processor Dell machines. We have no explanation for this phenomenon but hypothesize that the HP machines had higher performance I/O subsystems since the simulation times showed little difference in processor performance. We also observed that the cost of symmetric multiprocessing on the dual processor HP machines was a 10–15% increase in simulation time when using both processors compared to using a single processor. The cause was most likely memory contention, mutual exclusion and cache effects.

A.3 Simulation Times

As mentioned earlier, obtaining the data presented in this work required a substantial amount of simulation time. Tables A.6, A.7 and A.8 give average simulation times to obtain the final results for one data point on a graph. In the preliminary results reported in [58], each system of a workload executed a minimum of 1,000 jobs in each task. To improve the quality of the results, the overrun simulations were run again with a minimum of 2,000 jobs in each task. Although this was an improvement, a higher quality at 90 and 95% utilization was desired for the release time variation results and the results with both release time and execution time variations. To keep simulation times reasonable, the minimum number of jobs were increased in increments of 1,000 from 1,000 at 50% utilization to 4,000 at 95% utilization. Likewise, the minimum number of jobs was doubled at each step while investigating the effects of overrun using the MPEG traces (e.g., 1,000 at 50% utilization to 8,000 at 95% utilization). All together, the final results presented in this thesis took over 274 days of simulation time. However, the total amount of simulation time, including exploration and rerunning to obtain more accurate results, is approximately 2.5–3.5 times that amount.[3]

A.4 Lessons Learned

At the conclusion of any substantial project, it is appropriate to reflect upon the lessons learned. (We note that during a substantial project the ability to backtrack in order to recover from an earlier decision is often lost as the project proceeds. Typically, the cost of backtracking becomes prohibitive during a project because

[3]The simulation results for the Constant Utilization Server shown in Figure 5.8 on page 67 took nearly 26 days to obtain at an average of 6.5 days per data point. Conveniently, the performance of the Total Bandwidth Server was much better, as was its simulation times.

Table A.6: Average Simulation Times for Baseline Algorithms in Hours.

Workload	Average	Utilization 50%	75%	90%	95%
Overrun	5.3	5.5	5.3	5.3	5.3
Jitter	6.6	2.9	5.2	7.3	9.3
Both	6.3	2.6	5.0	7.4	9.8
Museum	4.3	1.3	2.3	4.8	9.0
A Close	4.2	1.2	2.3	4.5	9.0
Red's	4.2	1.2	2.3	4.5	8.6

Table A.7: Average Simulation Times for OSM in Hours.

Workload	Base	Servers	Average	Utilization 50%	75%	90%	95%
Overrun	DM	1	6.3	6.2	6.1	6.2	6.7
	DM	8	9.6	7.0	7.6	9.8	13.5
	EDF	1	5.9	6.0	5.8	5.9	5.8
	EDF	8	5.8	6.0	5.8	5.7	5.8
Jitter	DM	1	6.8	2.8	5.5	8.3	10.6
	DM	8	7.8	4.1	5.6	8.8	12.8
	EDF	1	6.5	2.8	5.1	7.8	10.1
	EDF	8	6.3	2.8	5.1	7.8	10.3

Table A.8: Average Simulation Times for ISM in Hours.

Workload	Base	Servers	Average	Utilization 50%	75%	90%	95%
Overrun	DM	1	8.3	6.0	6.0	6.8	14.3
	DM	8	7.0	6.7	6.8	6.9	7.8
	EDF	1	5.3	5.5	5.3	5.2	5.2
	EDF	8	5.3	5.5	5.4	5.4	5.3
Jitter	DM	1	10.0	3.0	5.6	9.9	21.5
	DM	8	10.5	3.4	6.3	11.0	21.0
	EDF	1	6.4	2.8	5.1	7.7	10.0
	EDF	8	6.5	3.0	5.3	7.6	10.0
Both	DM	1	12.8	3.1	5.7	11.5	30.9
	DM	8	11.5	3.5	6.6	12.2	23.7
	EDF	1	12.8	5.7	10.3	15.1	20.0
	EDF	8	12.8	5.7	10.2	15.1	20.0

of time or cost constraints.) Thus it is very important to learn from mistakes. In this case, the lessons are both positive and negative.

It is reasonable to ask why an existing simulation framework was not used rather than writing one from scratch. The answer is that the simulator framework was largely written as part of a separate project to gain experience with Java and to develop an applet to display schedules for real-time systems in a web browser. Making modifications to adapt the framework for discrete event simulation required substantially less effort than learning another simulation framework. While the framework has performed well for the task at hand, it may be wise to learn a more general framework as insurance against future need.

Developing the simulation framework lead to greater intuition about when to use inheritance and when to use parameterization. The framework was designed so that specialized behavior is obtained by subclassing a parent class and overriding appropriate methods. This approach was used extensively to develop subclasses of class `Event`, for example. The primary difference between events is the behavior that occurs when the event handler is invoked. Rather than subclass to override the `occur` method, the `Event` class should be parameterized with the code that is called when an event occurs. Using a single `Event` class with a parameter specifying its handler would reduce the amount of code that needs to be written to create a simulation. It is recommended that programs written in Java should make extensive use of *inner classes* ([72]) to specialize behavior rather than use inheritance.[4]

Even with inner classes, it is still not as convenient to parameterize behavior in Java as one would like. Inner classes are a heavy weight solution for many parameterization needs because an inner class is a complete anonymous class that is defined textually within another class. Therefore, inner classes still require at least one constructor and one method to specialize behavior. They also require instance variables to be set explicitly in order to save the appropriate state for later use. On the other hand, languages like Lisp and Smalltalk have closures which capture the lexical environment implicitly and do not require construction of complete classes. An additional benefit of Lisp and Smalltalk is the flexibility that comes from the dynamic typing of variables rather than the static typing offered by Java

[4]Until the addition of inner classes in Java 1.1, the language did not conveniently support the parameterization of behavior except by the creation of named classes in separate files. Hence, inheritance was used in the design of the framework instead of parameterization. Also, the Toba compiler did not support inner classes until well after the framework was in use.

(albeit, at the cost of requiring better type inference in the compiler to achieve good performance). Thus, Lisp or Smalltalk may be better languages for writing simulations than Java.

On the positive side, the choice of Java over C or C++ for writing the simulator framework was vindicated by faster development times. Because the performance of the simulator when compiled with Toba was comparable to a simulator written in C or C++, the initial concern about adequate performance faded. On the negative side, the lack of generic containers in Java proved to be a constant nuisance.[5] Extending Java to have generic containers would make the use of containers more convenient and enable the compiler to eliminate many run time type checks. There are several proposal for adding generic containers to Java but none have been adopted as standard yet.

The final lesson learned concerns the robustness of Linux as a production environment. Over the past year, Linux has seen extremely heavy use without once crashing. In particular, the file system has performed flawlessly in spite of the tremendous amount of data created by the performance studies. In addition, reconfiguring the Linux kernel to support additional capabilities, such as SMP, is relatively painless. It takes approximately 11 minutes to perform a clean rebuild of the kernel, install it and reboot. In contrast, colleagues who are extending the Windows NT kernel with real-time support report that a clean rebuild of the kernel takes approximately 11 hours and requires manual intervention in strategic spots. While Windows NT seems to be the politically correct choice for systems research, stability and ease of modification have proven Linux to be an excellent alternative.

[5] By a generic container, we mean an object in which a homogeneous collection of other objects (or references to them) are stored. Examples include lists, heaps, and binary trees. The containers in Java hold references to objects descended from class `Object`. An object retrieved from a Java container must undergo a run time type check, called a "type cast", to recover its specific type before use. Generic containers know the type of their contents making type checks unnecessary.

Appendix B

MPEG Video Decoding

In this thesis we have shown that a statistical approach better characterizes the performance of critical soft real-time systems than does a deterministic approach, especially for systems in which the execution times and inter-release times of tasks vary widely. Following standard practice in the statistics and simulation communities, we used uniform or exponential distributions of execution times for most of our analysis and simulation. However, real world tasks have execution time dependencies. One task which exhibits widely varying execution times and dependencies is a decoder for MPEG video.

A MPEG video stream is composed of frames which correspond to images to be displayed. In order to reduce storage and transmission costs, the raw video frames are transformed into one of three types of MPEG frames. The first is the *I-frame* in which the raw pixel data is divided into 2-D blocks and compressed by the discrete cosine transform. The next is the *P-frame* which encodes data used to "predict" where features of the previous I-frame will be in the future. The last is the *B-frame* which "bi-directionally interpolates" features of surrounding I and P frames in relation to the current frame. Because of dependencies, some MPEG frames are positioned earlier in the MPEG video stream than the raw video frames they represent would indicate. The frames that are decoded early are buffered until they are displayed. The excellent compression ratios achieved by the MPEG algorithm are possible because the sequence of P and B frames are often significantly smaller than if the stream had been encode as I-frames alone. Thus a sequence of MPEG frames, such as I-B_2-P-B_2[1], is smaller than an encoding containing only I-frames. Due to the sophisticated encoding scheme, the

[1]The notation I-B_2-P-B_2 is shorthand for sequence I-B-B-P-B-B. We give all frame sequences in decode order rather than in display order.

execution time of jobs in a MPEG video task exhibit variability for which a statistical approach is especially well suited. (For more information on MPEG visit `<http://www.mpeg.org>`.)

B.1 Trace Acquisition

Traces of frame decode times for MPEG video streams were created specifically for the study. Measured execution times were limited to the decode times of individual frames rather than including the time spent reading the encoded frames from disk or displaying the decoded frames. In other words, we measured executions times from the perspective of a real-time task whose only function is to decode MPEG frames. As a result, the execution times are lower than other studies (such as [44]) which measure the complete time required to read, decode and display a frame.

The decode times for each frame were obtained by surrounding the body of `mpegVidRsrc` of the `mpeg_play` application, developed at the University of California Berkeley[2], with calls to a high resolution time source (explained further below). For convenience, the options were set to decode the video stream as fast as possible (`framerate 0`) and to suppress displaying to the screen (`no_display`). Setting the options to achieve the fastest possible running of `mpeg_play` did not bias the results because only the decode function was being timed.

The data was collected under RedHat Linux 5.2 with the Linux 2.2.3 kernel running on a 200 MHz Pentium Pro machine. The load on the machine at the time of the test was light, consisting primarily of daemon processes. Initial measurements were taken via standard Unix `gettimeofday` calls. However, the resolution was insufficient to obtain accurate data because of interference from daemon processes and interrupt handlers. To improve the quality of the data, the `rtime` function of the`libpperf` library, available from `<http://qso.lanl.gov/~mpg/libpperf-0.5.tar.gz>`, was used to yield a granularity of 1 clock cycle or 5 nanoseconds. (The `libpperf` library accesses a model-specific status register of the Pentium Pro which contains a count of the number of CPU cycles since startup or reset.) Each video was decoded at least 100 times and the average decode time was computed for each frame.

[2]The source code for `mpeg_play` is available at `<http://bmrc.berkeley.edu/projects/mpeg/mpeg_play.html>`.

In computing the average for each frame, it was noticed that decode times up to 20 times longer than the average occurred occasionally. These extraordinarily long decode times are likely caused by the execution of an interrupt handler during frame decode. Clearly, long decode times could affect the average decode time of a frame. Assuming that the variations in decode times for a frame are the result of a large number of random events, the Central Limit Theorem states that the distribution of decode times for a specific frame will converge to the Gaussian distribution in the limit. We therefore disregarded decode times outside $\pm 4\sigma$ and recomputed the mean. This process was repeated until no data points were discarded. In general, less than 5 points were discarded for each frame. The widths of the 95% confidence intervals were less than 0.7% for each frame. (Because of the large number of samples for the decode times of each frame, the average decode times with outlying points discarded differed only slightly from the average with no points discarded.)

We now present the statistical characterization of the traces along with their distributions. The traces themselves are available electronically.[3]

B.2 The Incredible Museum Video

The first video stream is a walk-through of a ray traced model of an imaginary museum.[4] It was chosen because of its length (3909 frames) and because its decode times vary by over a decimal order of magnitude. (See Table B.1 for a summary of the statistical characteristics of the trace.) The video stream lasts 130.3 seconds at a natural frame rate of 30 fps and has a width and height of 352x240. The size of the video stream is 25,320,484 bytes and has the (rather typical) frame sequence I-B_2-P-B_2.

Of the selected video streams, the Incredible Museum video stream showed the most pronounced periodicity of decode times. A look at the decode times as a function of the frame number, Figure B.1, shows that the decode times of each frame type changes drastically with scene changes. For example, the frames up to 330 and from 3475 to the end are the title and production credits. Frames 660–850, 930–1430, 1715–1987, 2226–2470, 2740–3115, and 3240–3475 are scenes

[3]The traces are available at <ftp://ftp.cs.uiuc.edu/pub/research-groups/perts/video-traces>.

[4]The video stream is available from <http://private.homepages.intershop.de/rene/museum.html>.

Table B.1: Characteristics of the Incredible Museum video.

Frame	Count	Average Decode Time, $msec$ Minimum	Maximum	Mean	Variance
All	3909	0.64	9.03	3.44	2.93
I	652	1.17	8.30	5.24	2.42
P	652	0.68	9.03	4.65	4.10
B	2065	0.64	7.34	2.68	1.01

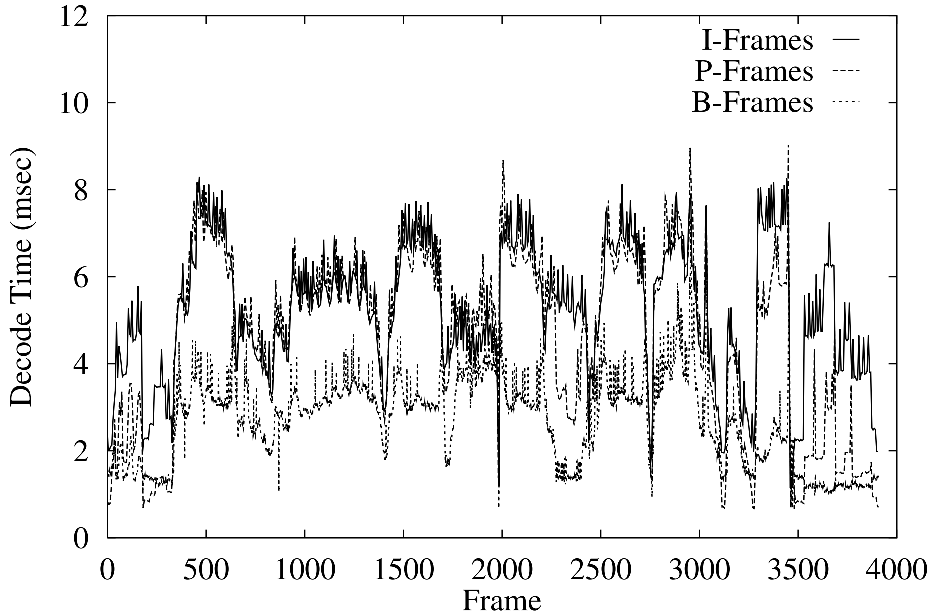
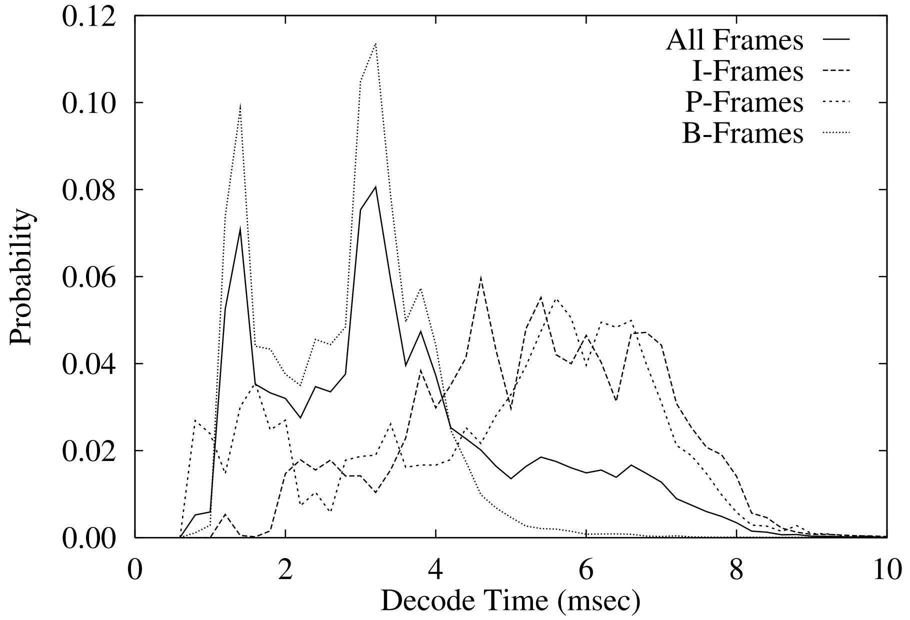


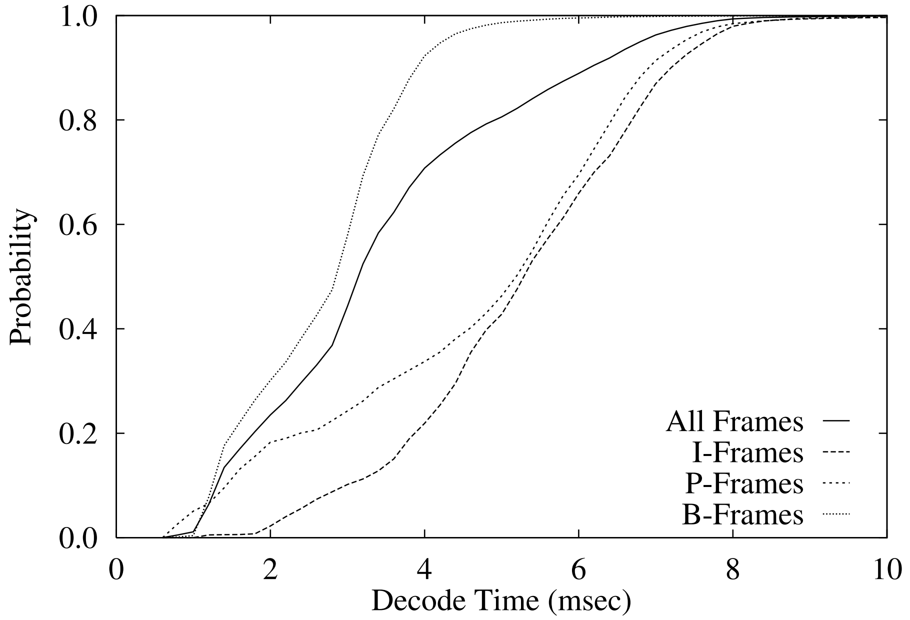
Figure B.1: Decode Times for the Incredible Museum video.

within different rooms of the museum. The remaining frames are of the central hall. Transitions from a room to the central hall or from the central hall to another room correspond quite closely to the dramatic changes in the decode times.

In Figure B.2, we show the density function and distribution of decode times for the Incredible Museum video stream. This video stream has the second greatest spread of execution times and the most gradual rate of change of the probability distribution functions of all the selected videos. Note that the density function of the B-frames is bi-modal. The density function of all the frames together is also bi-modal in direct consequence of there being four B-frames for every I or P frame.



(a) Density



(b) Distribution

Figure B.2: Statistical Characterization of the Incredible Museum video.

B.3 “A Close Shave” Video

The next video stream is of a delightful Wallace and Grommit animation by Nick Park entitled “A Close Shave”.[5] It was chosen because its length is over 1000 frames. Its was also chosen because its decode times vary by nearly a factor of 9, as shown in Table B.2, although most of the decode times were within the range 2–8 msec. The video stream lasts 56.84 seconds at natural frame rate of 25 fps and has a width and height of 352x288. The size of the video stream is 8,755,204 bytes and it has the frame sequence I-B_3-P-B_3-P-B_3.

Table B.2: Characteristics of the “A Close Shave” video.

		Average Decode Time, *msec*			
Frame	Count	Minimum	Maximum	Mean	Variance
All	1421	1.60	14.41	4.05	0.93
I	119	3.98	9.44	5.77	0.69
P	237	3.62	7.58	5.21	0.34
B	1065	1.60	14.41	3.60	0.16

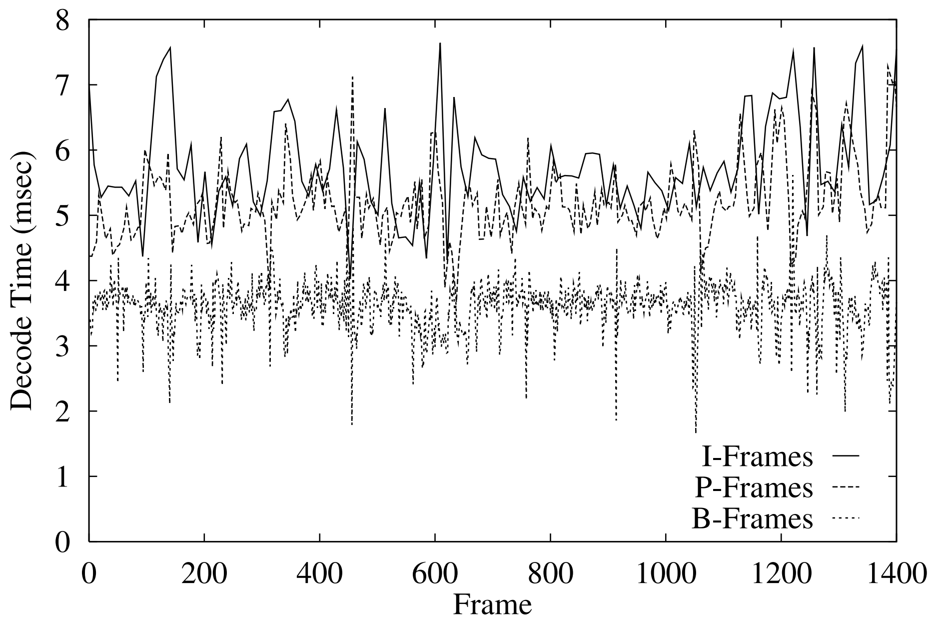
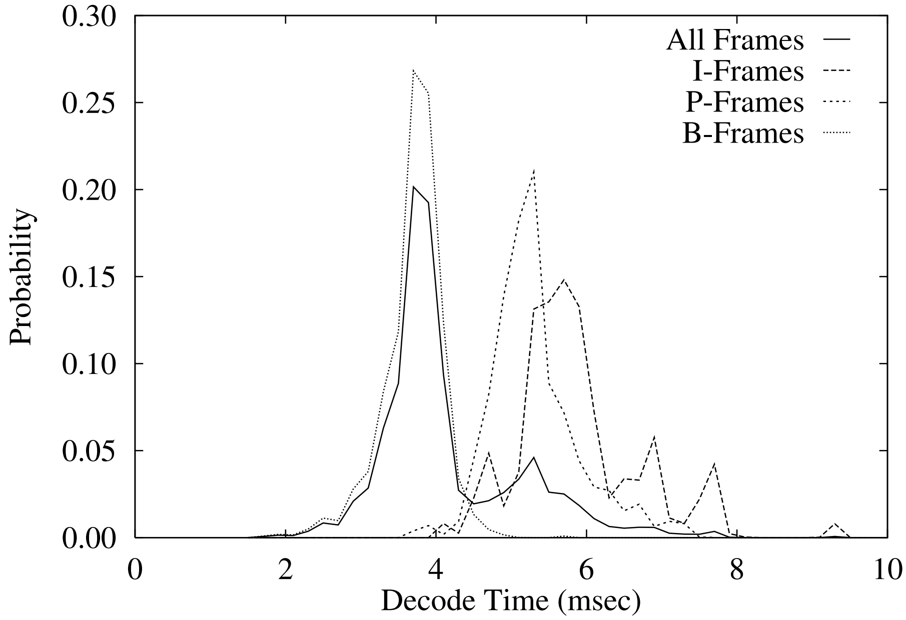
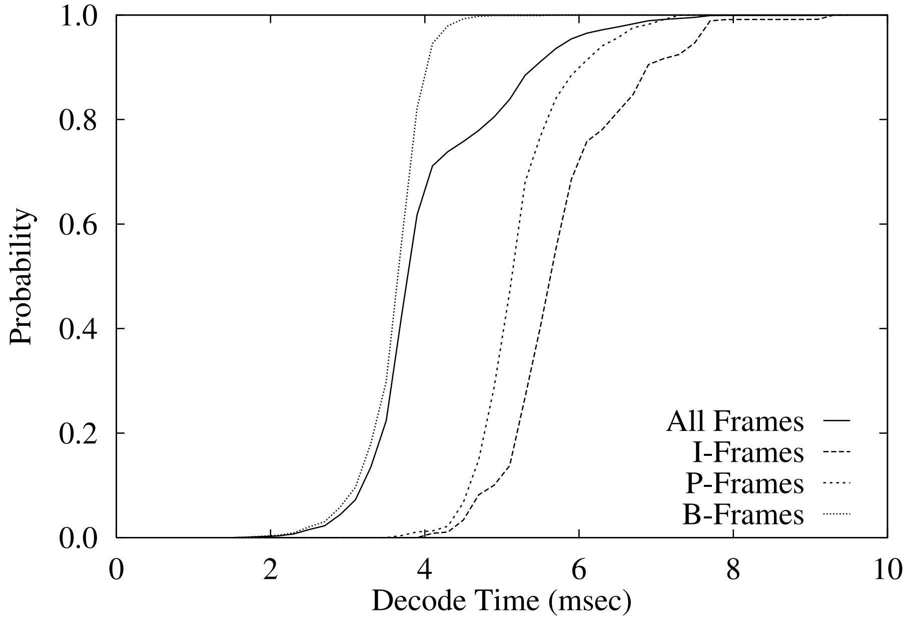


Figure B.3: Decode Times for the “A Close Shave” video.

[5]The video stream is available from `<http://animafest.hr/z96/mpeg/aclose_v.mpg>`.



(a) Density



(b) Distribution

Figure B.4: Statistical Characterization of the "A Close Shave" video.

B.4 "Red's Nightmare" Video

The final video stream is an animation entitled "Reds Nightmare".[6] It too was chosen because of a length over 1000 frames. Its decode times vary by a factor of 15, as shown in Table B.3. The video stream lasts 48.4 seconds at a natural frame rate of 25 frames per second and has a width and height of 320x240. The stream is the shortest and smallest of the selected videos, with a size of 3,619,896 bytes. It has the frame sequence I-B_9-P-B_9-P-B_9.

Table B.3: Characteristics of the "Red's Nightmare" video.

		Average Decode Time, *msec*			
Frame	Count	Minimum	Maximum	Mean	Variance
All	1210	0.60	9.32	2.30	1.01
I	41	3.96	7.82	4.99	0.44
P	81	0.61	9.32	3.68	1.95
B	1088	0.60	8.02	2.09	0.50

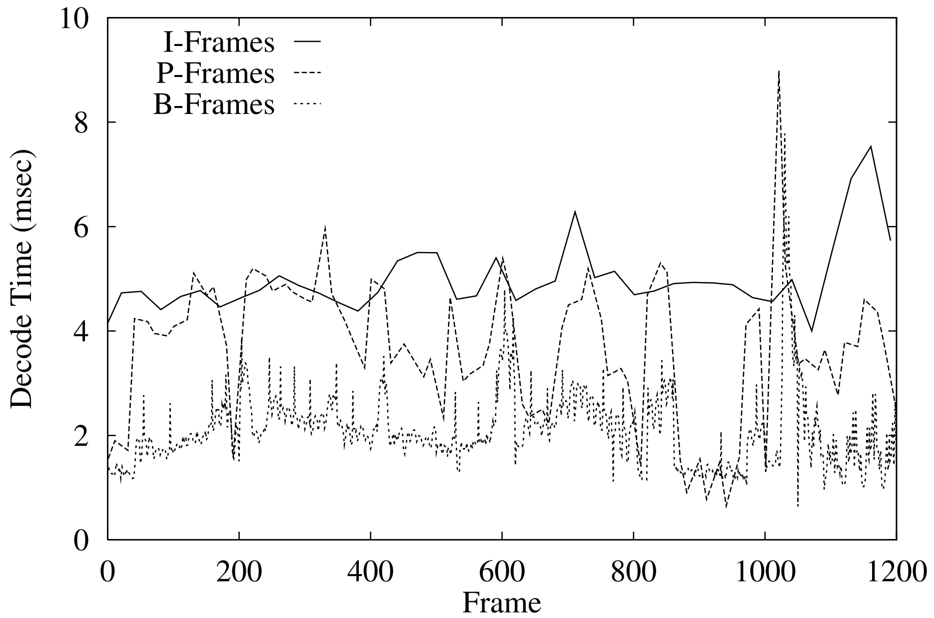
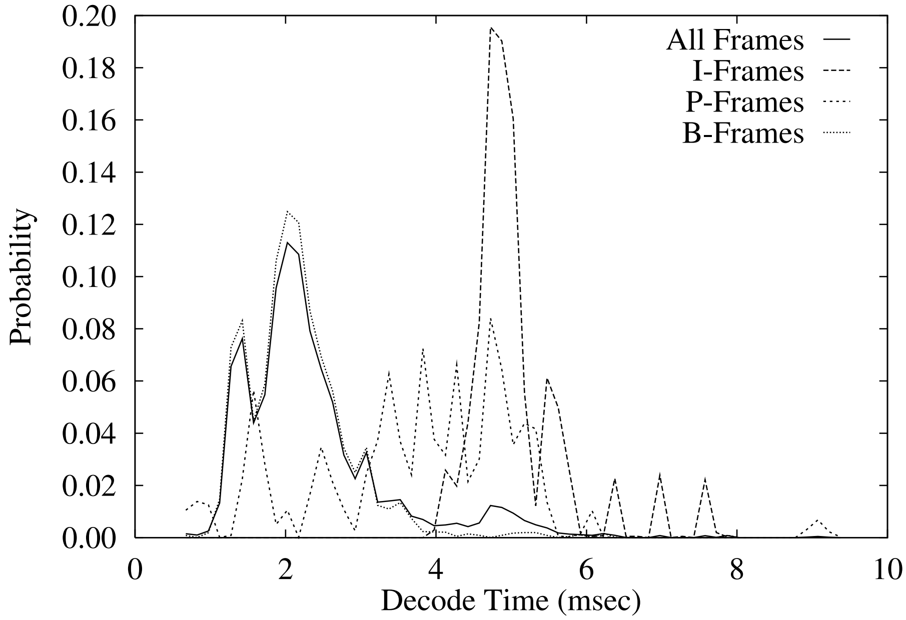
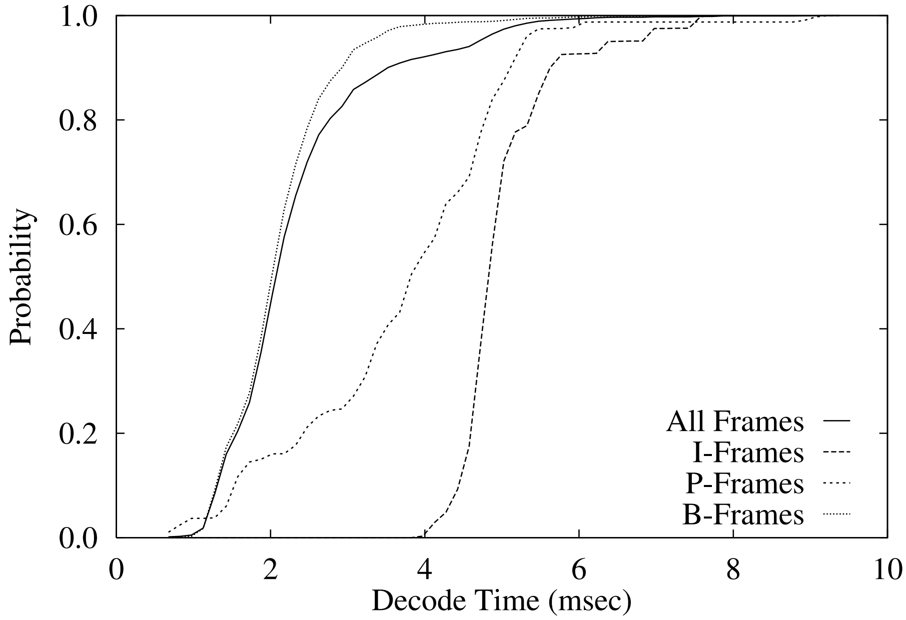


Figure B.5: Decode Times for the "Red's Nightmare" video.

[6]The video stream is available from <ftp://ftp.luth.se/pub/misc/anim/anim/RedsNightmare.mpg>.



(a) Density



(b) Distribution

Figure B.6: Statistical Characterization of the “Red’s Nightmare” video.

B.5 MPEG Models

The average execution time distributions in the previous sections provided the execution times for the MPEG decode tasks in Section 5.6. We observe that the size of the coded frames is dependent on the composition of the original frames and hence the decode times are also variable. There are wide variations in composition in most video streams, often for artistic impact. This suggests that the raw video stream could be separated into "scenes", which are groups of raw frames that are "similar" from an MPEG point of view, with the encoding of each scene performed independently. Thus, a video stream may contain many different frame sequences. Standard practice is to encode a video stream with a single frame sequence[7], but the development of accurate models which adequately capture the characteristics of general video streams is the subject of ongoing research. As understanding of video streams improves, we expect MPEG encodings to become more efficient which will lead to wider variations in decode times, increasing the utility of the techniques we have developed.

[7]The primary exception is that the first sequence differs occasionally.

Appendix C

Adaptive Statistical Characterization

Unlike deterministic approaches for hard real-time systems, the probabilistic real-time approaches described in this work assume that distributions for inter-release times and execution times are known with sufficient accuracy for each task in the system. However, accurate distributions may not be readily available a priori. The system must then discern the characteristics of the tasks "on-the-fly". We will now address approaches for learning the statistical properties of tasks as they execute and present an adaptable approach that allows tasks with varying behavior to influence how the system schedules them.

Suppose a task with unknown characteristics is admitted to the system. There is a risk that its admission will cause the miss rate of previously admitted tasks to increase. Thus, rather than admitting the task unconditionally before its behavior is known, we admit the task conditionally and allow it to consume time when the processor is idle while we have monitor its behavior. The task is reconsidered for unconditional admission when its behavior is quantified with sufficient accuracy. Initially, the jobs of a conditionally admitted task are assumed to have constant execution times and no release time jitter. (In the absence of further information, we may safely assume that the task has a zero execution time and that its period is infinite.) The inter-release time and execution time distributions are updated after each job of a conditionally admitted task completes. When the profile of the probability distribution functions are known to within the required tolerance, the task is unconditionally admitted to the system.[1]

We note that maintaining the information necessary to compute distributions

[1] To ensure that the system does not prematurely consider a distribution accurate, some minimum number of jobs need to be executed.

is costly in terms of both processing time and memory. One approach to limiting the cost is to maintain a circular buffer of past history from which the distributions can be computed. The buffer is either empty initially or it is primed with default values. The capacity of the buffer represents a window on the behavior of the task. Too small a buffer size and the of behavior the system will not be adequately captured. Too large a buffer size and the cost, both in time and space, will be excessive.

If the range of values can be estimated, the storage of individual values and repeated computation of the distribution can be eliminated. The estimated range is divided into a suitable number of bins with the first and last bins representing values below and above the range, respectively. Each time a job completes, the count within the bin in which the value falls is incremented. The total number of values is also incremented. The probability represented by each bin is computed as the number of values which lie within the range of the bin divided by the total number of values. When the change in probability of all bins is within some threshold, the distribution is deemed sufficiently accurate for use in admission control.

Another approach, which may work for some systems and which requires very little time or space, is to maintain a few descriptive statistics, e.g., minimum, maximum, mean, and variance. Suppose X is the random variable representing the quantity of interest. Initially, the sample minimum is $X_0^- = +\infty$, the sample maximum is $X_0^+ = -\infty$, the sample mean is $\bar{X}_0 = 0$, and the sample variance is $S_0^2 = 0$. The statistics are updated with each new sample according to the following recursive definitions [73][2].

$$X_{i+1}^- = \min(X_{i+1}, X_i^-) \tag{C.1}$$

$$X_{i+1}^+ = \max(X_{i+1}, X_i^+) \tag{C.2}$$

$$\bar{X}_{i+1} = \bar{X}_i + \frac{X_{i+1} - \bar{X}_i}{i+1} \tag{C.3}$$

$$S_{i+1}^2 = (1 - \frac{1}{i})S_i^2 + (i+1)(\bar{X}_{i+1} - \bar{X}_i)^2 \tag{C.4}$$

When the relative change in the statistics are less than some threshold, the task can be admitted. The minimum inter-release time and maximum execution time can

[2]There are subtle issues involving numerical roundoff when computing the mean and variance of a sample sequence using recursive definitions implemented with floating point numbers. We will ignore these issues because the effect is likely to be less than that caused by the use of descriptive statistics in the place of actual distributions.

be used for admission control via the Liu and Layland bound or Time Demand Analysis. Alternatively, the task can apply for admission based on guaranteed execution and inter-release times chosen from within the observed bounds. In that case, the inter-release times and execution times can be assumed to follow a normal distribution and Statistical Time Demand Analysis used to estimate the percentage of deadlines which will be met.

We have shown how the use of adaptive statistical characterization mechanisms can be employed to gather information for use with admission control algorithms. However, the mechanisms may also be used to adapt the characterizations of tasks for scheduling purposes. While the statistical characterization of the behavior of a task measured within a finite window of jobs can be used directly, we note that the programmatic interface to the scheduler may be simplified, at the expense of providing less direct control, by specifying a percentage of the range between the minimum and average inter-release times and a percentage between the average and maximum execution time as indications to the scheduler whether the program prefers improved utilization or guaranteed deadlines.[3] We call the two percentages the *period control* and the *execution time control*, respectively. (A further simplification arises when we define the previous two controls in terms of a single parameter.) Once the inter-release time and execution time parameters have been estimated through conditional execution, the guaranteed inter-release and execution times are computed from the specified controls and unconditional admission requested. Although the percentage of deadlines met is not a linear function of the controls in general, the value of a control is correlated to the performance and provides a means of influencing the processor allocation given to a task in the absence of known statistical properties.

Another advantage of specifying period and execution time controls rather than guaranteed inter-release times and execution times is that the guaranteed inter-release times and execution times can change as the behavior of the task changes. To do this requires continued measurements after a task has been unconditionally admitted. If circular buffers of values are maintained while computing the statistical properties of task, the effect of a particular value can be retired before it is removed from the buffer, thereby allowing adaptation as the program behavior changes. Alternatively, an exponential decay can be used in the place of the execution history contained in the circular buffers.

[3] Slightly more complex specifications involving variances are also possible with attendant improvements in accuracy.

Adaptations which increase the minimum or average inter-release times or adaptations which decrease the average or maximum execution times can be enacted immediately without causing the average miss rates of other tasks to increase. Adaptations in the opposite direction require that the system evaluate whether the guaranteed inter-release time can be decreased or the guaranteed execution time can be increased without causing other tasks to miss more deadlines.

Appendix D

Sums of Random Variables

Bounding sums of random variables has been the goal of an enormous amount of research in statistics. The following are some results which can be used to bound the sums of random variables representing the execution times of jobs in a real-time system.

D.1 Chebyshev Inequality

The Chebyshev Inequality is commonly used to provide a bound on the probability distribution function of a random variable when only the mean or standard deviation is known. For a derivation of the inequality or for alternative formulations, see [46, 74, 75].

Lemma D.1.1 *Given a random variable X with mean μ and variance σ^2, the probability that a value of X differs from the mean by more than $\delta \geq 0$ is*

$$\mathcal{P}[|X - \mu| \geq \delta] \leq \frac{\sigma^2}{\delta^2}$$

D.2 Bennett Inequality

The Bennett Inequality, as stated in [75], requires a zero mean for each random variable in a sum of independent random variables. The formulation given here allows a non-zero mean and corresponds directly to a probability distribution function.

Lemma D.2.1 *Given independent random variables, $X_1, X_2, \ldots, X_n$, with each random variable X_i having mean μ_i and variance σ_i^2, if all values of X_i are* *bounded by a constant b such that $|X_i - \mu_i| \leq b$, for $1 \leq i \leq n$, then the probability that a value of $\mathbf{X}_n = X_1 + X_2 + \cdots + X_n$ differs from the mean of $\mathbf{X}_n$, denoted $\mu_n = \mu_1 + \mu_2 + \cdots + \mu_n$, by more than $\delta \geq 0$ is*

$$\begin{aligned}\mathcal{P}[\mathbf{X}_n \geq x] &\leq \exp\left(\frac{-n\lambda}{2\sigma^{\mathbf{2}}{}_n}\psi\left(\frac{\sqrt{n}\lambda b}{\sigma^{\mathbf{2}}{}_n}\right)\right) \\ \psi(y) &= \frac{2}{y^2}[(1+y)\log(1+y) - y]\end{aligned}$$

where $x = \lambda\sqrt{n} + \mu_n$ and $\sigma_n^s = \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2$ is the variance of $\mathbf{X}_n$.

D.3 Bernstein Inequality

The Bernstein Inequality, as stated in [75], requires a zero mean for each random variable in a sum of independent random variables. The formulation given here allows a non-zero mean and follows directly by way of algebraic manipulation.

Lemma D.3.1 *Given independent random variables, $X_1, X_2, \ldots, X_n$, with each random variable X_i having mean μ_i and variance σ_i^2, if*

$$\mathcal{E}\left[|X_i - \mu_i|^n\right] \leq \frac{v_i n! c^{n-2}}{2}$$

for $n \geq 2$ and $c > 0$, then the probability that a value of $\mathbf{X}_n = X_1 + X_2 + \cdots + X_n$ differs from the mean of $\mathbf{X}_n$, denoted $\mu_n = \mu_1 + \mu_2 + \cdots + \mu_n$, by more than $\delta \geq 0$ is

$$\mathcal{P}[\mathbf{X}_n - \mu_n \geq \delta] \leq \exp\left(\frac{-\delta^2}{2(\mathbf{v}_n + c\delta)}\right)$$

where $\delta = \lambda\sqrt{n}\,\mathbf{v}_n = v_1 + v_2 + \cdots + v_n$. Note that $c = b/3$ is suggested in [75], if $|X_i - \mu_i| \leq b$.

D.4 Berry-Esseen Inequality

The Berry-Esseen Inequality, as stated in [75, 76], requires a zero mean for each random variable in a sum of independent random variables. The formulation given here allows a non-zero mean and follows by way of algebraic manipulation.

Lemma D.4.1 *Given independent random variables, $X_1, X_2, \ldots, X_n$, with each random variable X_i having mean μ_i and variance σ_i^2, if the absolute third central moment, $\mathcal{E}\left[|X_i - \mu_i|^3\right]$ of each random variables is finite, then the probability distribution function of $\mathbf{X}_n = X_1 + X_2 + \cdots + X_n$ differs from the probability distribution function of a normal distribution with the same mean and variance as $\mathbf{X}_n$, denoted $\mu_n = \mu_1 + \mu_2 + \cdots + \mu_n$ and $\sigma^{\mathbf{2}}{}_n = \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2$, by*

$$\sup_{x \in \mathcal{R}} |\mathcal{P}[\mathbf{X}_n \leq x] - N(x)| \leq c \frac{\mathcal{E}\left[|X_i - \mu_i|^3\right]}{\sigma_n^3}.$$

The smallest known value of c for which the inequality holds when distributions are identical is

$$c = \frac{\sqrt{10} + 3}{6\sqrt{2\pi}}.$$

When distributions are not identical, the smallest known value $c = 0.7975$.

D.5 Hoeffding Inequality

The Hoeffding Inequality (see [75]) requires each distribution to be bounded, but does not require a knowledge of the means and variances.

Lemma D.5.1 *Given independent random variables, $X_1, X_2, \ldots, X_n$. If each random variable X_i has bounds $a_i \leq X_i \leq b_i$, then the probability that a value of $\mathbf{X}_n = X_1 + X_2 + \cdots + X_n$ differs from the mean of $\mathbf{X}_n$, denoted $\mu_n = \mu_1 + \mu_2 + \cdots + \mu_n$, by more than $\delta \geq 0$ is*

$$\mathcal{P}[\mathbf{X}_n \geq x] \leq \exp\left(\frac{-2\delta^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right)$$

where $x = \delta + \mu_n$ and $\delta = \lambda\sqrt{n}$.

Appendix E

Implementing STDA

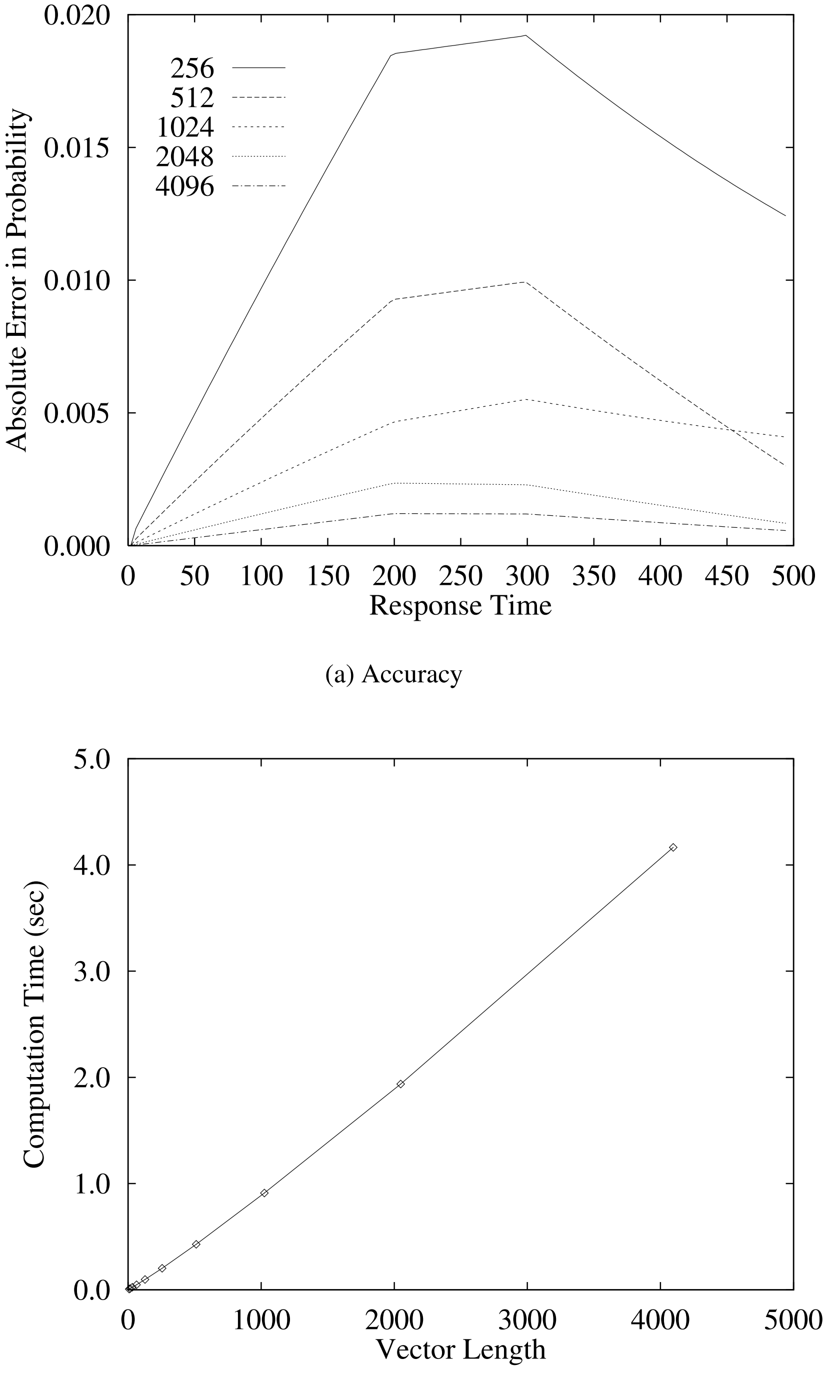
In this chapter, we discuss an implementation of STDA in the PERTS real-time prototyping environment [77, 78]. PERTS is a tool which facilitates the design and analysis of real-time systems by applying theoretical results, where possible, or by simulating the system to determine its behavior. The issues we discuss are not particular to PERTS and must be addressed by any implementation of STDA.

One of the main operations in STDA is the summing of random variables representing execution times. It is well known that the probability density function of the sum of two statistically independent random variables can be obtained by convolution $f(t) = g(t) \otimes h(t)$.The direct way to perform convolution on a digital computer is to discretize the integral using a constant spacing between samples $f_i = \sum_{j=0}^{N-1} g_i h_{i-j}$. Computing f by direct convolution is an $\mathcal{O}(N^2)$ operation, where N is the number of points in the discrete representations of g and h. It has long been known that the asymptotic cost of convolution can be reduced by applying the *Convolution Theorem* $g(t) \otimes h(t) \iff G(f)H(f)$, where $G(f)$ and $H(f)$ are the Fourier transforms of $g(t)$ and $h(t)$ respectively. The result is an $\mathcal{O}(N \log_2 N)$ algorithm for convolution. There are many descriptions and implementations of the FFT readily available (e.g., [51, 79, 80]).

Three issues need to be considered when using FFT to perform convolution. First, the discrete representations of the probability density functions being convolved must have the same sampling rate and consist of the same number of points. In STDA, the vectors containing the discretized probability density functions will almost always have different sample rates and numbers of points as a result of the conditioning process. Thus new vectors must be formed by interpolation before every convolution. Since interpolation can be performed in $\mathcal{O}(N \log_2 N)$ time, the asymptotic complexity of convolution is not increased.

Second, sufficient “zero padding” is required to ensure that aliasing does not occur [80]. The length of the vectors are also required to be a power of two for most implementations of the FFT. As a result, the vectors are likely to be large and sparsely populated in our application. Our experience indicates that the vectors are often only 50–75% filled with non-zero data. The final issue concerns the number of points used to represent the probability density functions for sufficient accuracy.

Figure E.1(a) shows the error between the computed and exact distributions of response time corresponding to Fig. 4.1(b) as a function of the number of points in the discrete representation. Figure E.1(b) shows the computation time as a function of the number of points. In order to maintain acceptable interactive response, we have chosen a default of 1024 points in the PERTS implementation of STDA, which yields a maximum absolute error of slightly over 0.005 for this example.



(a) Accuracy

(b) Time

Figure E.1: Convolution via FFT versus number of points.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[2] N. Audsley, A. Burns, K. Tindell, M. Richardson, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[3] T. P. Baker, "A stack-based resource allocation policy for real-time processes," in *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Dec. 1990, pp. 191–200.

[4] A. Burns, K. Tindell, and A. J. Wellings, "Fixed priority scheduling with deadlines prior to completion," in *Sixth Euromicro Workshop on Real-Time Systems*, June 1994, pp. 138–142.

[5] W.C. Feng and J. W.S. Liu, "Algorithms for scheduling real-time tasks with input error and end-to-end deadlines," *IEEE Transactions on Software Engineering*, vol. 23, no. 2, pp. 93–106, Feb. 1997.

[6] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, vol. 9, no. 1, pp. 31–67, July 1995.

[7] D. Gillies and J. Liu, "Scheduling tasks with and/or precedence constraints," *SIAM Journal on Computing*, vol. 24, no. 4, pp. 797–810, Aug. 1995.

[8] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Press, 1993.

[9] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the 10th Real-Time System Symposium*, Dec. 1989, pp. 166–171.

[10] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the 11th Real-Time System Symposium*, Dec. 1990, pp. 201–209.

[11] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *IEEE Real-Time Systems Symposium*, Dec. 1992, pp. 110–123.

[12] J. W.S. Liu, K. J. Lin, W. K. Shih, J. Y. Chung, A. Yu, and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer*, pp. 58–68, May 1991.

[13] A. K. Mok, *Fundamental design problems of distributed systems for the hard real-time environment*, Ph.D. thesis, MIT, 1983.

[14] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *IEEE Real-Time Systems Symposium*, Dec. 1988, pp. 259–269.

[15] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings: The 10th International Conference on Distributed Computing Systems*, May 1990, pp. 116–123.

[16] R. Rajkumar, *Synchronization in Real-Time Systems - A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

[17] R. Rajkumar, "Dealing with suspending periodic tasks," Tech. Rep., IBM T. J. Watson Research Center, Yorktown Heights, July 1991.

[18] S. Ramos-Thuel and J. P. Lehoczky, "Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing," in *Real-Time System Symposium*, 1994, pp. 22–33.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

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

[21] M. Spuri, G. Buttazzo, and F. Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *Proceedings of the 17th Real-Time System Symposium*, Dec. 1996, pp. 210–219.

[22] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo, "Implications of classical scheduling results for real-time systems," *Computer*, vol. 28, no. 6, pp. 16–25, June 1995.

[23] J. Sun, R. Bettati, and J. W.S. Liu, "An end-to-end approach to scheduling periodic tasks with shared resources in multiprocessor systems," in *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994, pp. 18–22.

[24] J. Sun and J. Liu, "Bounding the end-to-end response time in multiprocessor real-time systems," in *The Third Workshop on Parallel and Distributed Real-Time Systems*, Apr. 1995, pp. 91–98.

[25] J. Sun and J. W.S. Liu, "Synchronization protocols in distributed real-time systems," in *The 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[26] J. Sun and J. W.S. Liu, "Bounding completion times of jobs with arbitrary release times and variable execution times," in *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1996, pp. 2–12.

[27] J. Sun, M. K. Gardner, and J. W.S. Liu, "Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing," *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 603–615, Oct. 1997.

[28] K. W. Tindell, A. Burns, and A. J. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–152, Mar. 1994.

[29] J. Xu and D. L. Parnas, "On satisfying timing constraints in hard-real-time systems," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 70–84, Jan. 1993.

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

[31] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," in *Proceedings of the 17th Real-Time System Symposium*, Dec. 1996, pp. 22–29.

[32] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu, "Probabilistic performance guarantee for real-time tasks with varying computation times," in *Proceedings of the Real-Time Technology and Applications Symposium*, Chicago, Illinois, May 1995, IEEE, pp. 164–173.

[33] A. K. Atlas and A. Bestavros, "Statistical rate monotonic scheduling," Tech. Rep. BUCS-TR-98-010, Boston University, 1998.

[34] A. K. Atlas and A. Bestavros, "Statistical rate monotonic scheduling," in *Proceedings of the 19th Real-Time System Symposium*, Dec. 1998, pp. 123–132.

[35] Z. Deng, J. W.S. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *Proceedings of the Ninth Euromicro Workshop on Real-Time System*, Toledo, Spain, June 1997, pp. 191–199.

[36] J.Y. Chung, J. W.S. Liu, and K.J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1156–1174, Sept. 1990.

[37] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of the 17th Real-Time System Symposium*, Dec. 1996.

[38] D. I. Katcher, S. S. Sathaye, and J. K. Strosnider, "Fixed priority scheduling with limited priority levels," *IEEE Transactions on Computers*, vol. 44, no. 9, pp. 1140–1144, Sept. 1995.

[39] S. Keshav, *Congestion Control in Computer Networks*, Ph.D. thesis, University of California Berkeley, 1991.

[40] A. Demers, S. Keshav, and S. Shenkar, "Analysis and simulation of a fair queueing algorithm," *Internetworking: Research and Experience*, vol. 1, no. 1, pp. 3–26, Sept. 1990.

[41] P. Ramanathan, "Overload management in real-time control applications using (m, k)-firm guarantee," *IEEE Transactions on Parallel and Distributed Systems*, June 1999.

[42] M. Hamdaoui and P. Ramanathan, "Evaluating dynamic failure probability for streams with (m, k)-firm deadlines," *IEEE Transactions on Computers*, vol. 46, pp. 1325–1337, Dec. 1997.

[43] H. Chu and K. Nahrstedt, "Cpu service classes for multimedia applications," in *IEEE Multimedia Computing and Systems*, June 1999.

[44] H. Chu, K. Nahrstedt, and J. Qian, "Dynamic soft real time scheduling framework for multimedia environment," submitted to Real-Time Systems Symposium, Dec. 1999.

[45] J. Nieh and M. S. Lam, "SMART: A processor scheduler for multimedia applications," in *Proceedings of SOSP-15*, Dec. 1995, p. 223.

[46] L. Kleinrock, *Queueing Systems, Volume I: Theory*, vol. 1, Wiley-Interscience, 1975.

[47] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, vol. 2, Wiley-Interscience, 1976.

[48] J. P. Lehoczky, "Real-time queueing theory," in *Proceedings of the 17th Real-Time System Symposium*, Dec. 1996, pp. 186–195.

[49] J. P. Lehoczky, "Real-time queueing network theory," in *Proceedings of the 18th Real-Time System Symposium*, Dec. 1997, pp. 58–67.

[50] J. P. Lehoczky, "Scheduling communication networks carrying real-time traffic," in *Proceedings of the 19th Real-Time System Symposium*, Dec. 1998, pp. 470–479.

[51] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, second edition, 1982.

[52] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[53] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proceedings of the 19th Real-Time System Symposium*, Dec. 1998.

[54] K. Tindell, A. Burns, and A. Wellings, "Allocating real-time tasks. An NP-hard problem made easy," *Real-Time Systems Journal*, vol. 4, no. 2, May 1992.

[55] B. Kao and H. Garcia-Molina, "Deadline assignment in a distributed soft real-time system," in *The 13th International Conference on Distributed Computing Systems*, May 1993, pp. 428–437.

[56] J. Garcia and M. Harbour, "Optimized priority assignment for tasks and messages in distributed hard real-time systems," in *The Third Workshop on Parallel and Distributed Real-Time Systems*, Apr. 1995, pp. 124–132.

[57] J. Sun, *Fixed Priority Scheduling to Meet End-to-End Deadlines in Distributed Real-Time Systems*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1997.

[58] Mark K. Gardner and Jane W.S. Liu, "Performance of algorithms for scheduling real-time systems with overrun and overload," in *Eleventh Euromicro Conference on Real-Time Systems*. June 1999, IEEE.

[59] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control," in *Proceedings of IEEE INFOCOM*, 1992, vol. 2, pp. 915–924.

[60] D. Stiliadis and A. Varma, "Rate-proportional servers: A design methodology for fair queueing algorithms," *IEEE Transactions on Networking*, vol. 6, no. 2, pp. 164–174, Apr. 1998.

[61] L. Zhang, *A New Architecture for Packet Switched Networks*, Ph.D. thesis, Massachusetts Institute of Technology, July 1989.

[62] L. Zhang, "Virtualclock: A new traffic control algorithm for packet switching networks," in *Proceedings of ACM SIGMOD*, 1990.

[63] L. Zhang, "Virtualclock: A new traffic control algorithm for packet switched networks," *ACM Transactions on Computing Systems*, vol. 9, no. 2, pp. 101–124, May 1991.

[64] D. Stiliadis and A. Varma, "Latency-rate servers: A general model for analysis of traffic scheduling algorithms," in *Proceedings of IEEE INFOCOM*, 1996, vol. 1, pp. 111–119.

[65] D. Stiliadis and A. Varma, "Efficient fair queueing algorithms for packet-switched networks," *IEEE Transactions on Networking*, vol. 6, no. 2, pp. 175–185, Apr. 1998.

[66] H. Zhang and S. Keshav, "Comparison of rate-based service disciplines," in *Proceedings of ACM SIGCOMM*, Sept. 1991.

[67] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single node case," *IEEE Transactions on Networking*, vol. 1, 1993.

[68] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The multiple node case," *IEEE Transactions on Networking*, vol. 2, no. 2, pp. 137–150, Apr. 1994.

[69] J. W.S. Liu, "Real-time systems," manuscript in preparation for publication.

[70] M. K. Gardner and J. W.S. Liu, "Analyzing stochastic fixed-priority real-time systems," in *Lecture Notes in Computer Science 1579*, W. Rance Cleaveland, Ed. Mar. 1999, pp. 44–58, Springer-Verlag.

[71] G. Buttazzo and F. Sensini, "Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments," in *Proceedings of the Third IEEE International Conference on Engineering of Complex Systems*, Sept. 1997, pp. 39–48.

[72] J. Gosling, B. Joy, and G.L. Steele Jr., *The Java Language Specification*, Addison-Wesley, Sept. 1996.

[73] S. M. Ross, *Simulation*, Academic Press, Inc., second edition edition, 1996.

[74] R. C. Dubes, *The Theory of Applied Probability*, Prentice-Hall, Englewood Cliffs, NJ, 1968.

[75] G. R. Shorack and J. A. Wellner, *Empirical Processes with Applications to Statistics*, John Wiley and Sons, Inc., New York, NY, 1986.

[76] R. N. Bhattacharya and R. R. Rao, *Normal Approximation and Asymptotic Expansion*, Robert E. Krieger Publishing Co., Malabar, Florida, reprint edition, 1986.

[77] J. W. S. Liu, C. L. Liu, Z. Deng, T. S. Tia, J. Sun, M. Storch, D. Hull, J. L. Redondo, R. Bettati, and A. Silberman, "PERTS: A prototyping environment for real-time systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 161–177, 1996.

[78] 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 IEEE Real-Time Systems Symposium*, Raleigh-Durham, North Carolina, Dec. 1993, pp. 184–188.

[79] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[80] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, second edition, 1992.

Vita

Mark K. Gardner was born on July 25, 1960 in Madison, Wisconsin. He attended Brigham Young University (Provo, Utah) where he received a B.S. degree with honors in Mechanical Engineering in April 1986. From May 1986 to December 1990, he worked as an aerodynamic engineer for Allied-Signal Aerospace, Garrett Auxiliary Power Division (Phoenix, Arizona) designing centrifugal compressors and related components for Brayton cycle power systems. Returning to school in 1991, he completed a M.S. degree in Computer Science from Brigham Young University in December 1994. He continued his studies at the University of Illinois at Urbana-Champaign where he joined the Real-Time Systems Laboratory in August 1995 and completed the requirements for a Ph.D. degree in Computer Science in September 1999.