SCHEDULING IN DISTRIBUTED SOFT REAL-TIME SYSTEMS WITH AUTONOMOUS COMPONENTS

By

Benjamin Chi-Ming Kao

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

COMPUTER SCIENCE

November 1995

To my parents, Tchu-San and Chun-Hor

Abstract

Traditional real-time systems are usually used by very specific, mission-critical applications, such as a nuclear reactor monitor. For these applications task deadlines must not be missed. However, with the advances of computer and communication technology, and as computers become a commonplace utility, more applications are requiring a casual use of real-time technology. Task deadlines in these applications are *soft* in the sense that missing a deadline occasionally is, although undesirable, not a disaster. Example soft real-time applications include 800-number translation, cellular telecommunication, financial databases, patient-monitoring systems and so on. In these applications, deadlines are used as a tool to express performance goals. For example, a buy-sell action triggered by a stock-price update in a financial database system should be implemented within a specified period of time due to the volatility of the market. To conserve resources, soft real-time systems are usually not designed to guarantee every single deadline. Instead, the performance of a system is generally measured as the percentage of deadlines it meets. Deadline driven heuristic scheduling thus plays a major role in the design of soft real-time systems.

This thesis studies the scheduling aspect of distributed soft real-time systems. In particular, it addresses the following cost-performance tradeoff. When designing a

distributed real-time system, one approach is to build it from scratch, coding each component with algorithms and features that match the applications' needs. However, the development and maintenance cost of such a system would be high, and the development cycle could be long. Alternatively, one can build the system by piecing together well-tested, easy-to-maintain standard components, such as a commercial database server, a token ring network, etc. The disadvantage of this approach is that global real-time scheduling is not possible because it demands cooperation and coordination among the individual local schedulers. In this thesis, we discuss different scheduling techniques in a distributed soft real-time system that is constructed out of a variety of existing conventional non-real-time and real-time components, and show how these techniques improve the real-time performance of such a system without requiring the components to sacrifice autonomy. These techniques include *subtask deadline assignment*, *server prioritization*, and *real-time emulation*.

Acknowledgements

My six years as a Ph.D. student were made enjoyable by my mentor, Professor Hector Garcia-Molina, who taught me everything from research, writing and philosophy to his sense of humor. I cannot possibly thank him enough for the fun times and companionship. ¡Gracias Héctor!

Princeton University and Stanford University provided excellent facilities which made my research possible. Thanks to the faculty and staff for their support.

Special thanks go to:

Dr. Anthony Tomasic, my best American friend and my teacher of American history and culture, for helping me understand many of American social customs.

Brad Adelberg, my research partner at Stanford University, for putting up with my badly documented code. Oh, also for the slang lessons.

My thesis readers, Professor Gio Wiederhold and Professor Kai Li, for their excellent comments which greatly improved this thesis.

Other members of the Stanford Database Group: Jeff Ullman for the birthday cards (which were sent three months in advance, always); Luis Gravano for his brilliant debating skill; Sudarshan Chawathe for bugging Luis; Sergey Brin for his colorful slides; Anand "The Animal" Rajaraman for teaching me how to use ketchup in food; Joachim Hammer for the answers to many of my tennis questions; Kelly Ireland, Jan Jannink, Arthur Keller, Andreas Paepcke, Yannis Papakonstantinou, Dallan Quass, Jennifer Widom, Tak Yan, and Yue Zhuge for all the fun times.

Members of the Cantonese-speaking Joyce Luck Club: Joyce, Yet-Ping, John, Keung, Terry, Wilson, Judy, Lawrence, Pok, Thomas, and James for the movies and the parties.

My family, Winnie, Sue, Nelson, Mag, Peg, Mimi, and Vicky for their continuous support and encouragement over my years of studies.

Finally, this research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI), by the Telecommunications Center at Stanford University, by Hewlett Packard, and by Philips. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U.S. Government, CNRI, Hewlett Packard, or Philips.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in computer technology enable computer users to enjoy faster and more predictable response times. Every step in the evolution, from *batch processing* to *multiprogramming* to *interactive computation* and now to *real-time processing*, opens up new classes of applications that improve our standard of living. In the old days, when computers were slow and expensive, electronic computing was only accessible by a minute group of people for scientific calculations. Job turnaround time in the old era was in the order of hours. Debugging a program, for example, was a horrendous task. With the introduction of multiprogramming techniques and faster hardware, on-line applications became possible. Users can now purchase air tickets on the fly, play interactive video games, and withdraw money from an ATM machine in 30 seconds. The next step in the evolution is real-time processing: the ability to specify and satisfy time limits (deadlines) on the response time of the jobs. Real-time applications are ones whose correctness are dependent upon when output is produced. Some examples include nuclear-reactor monitoring systems, production control in a manufacturing plant, air flight control systems, etc. For these applications, the real-time systems have to respond to an input stimulus (e.g., the temperature of the nuclear reaction chamber has reached a certain dangerous level) and *react* (e.g., lower graphite rods to slow down the reaction activity) promptly, under a specified timing constraint.

There are two types of real-time systems: hard and soft. A hard real-time system must guarantee that every single job's deadline is met. Missing a deadline is considered a system failure and must be avoided at all cost. On the other hand, for soft real-time systems, missing a deadline occasionally is allowed although it is considered undesirable. For soft systems, the designer should strive to minimize the number of missed deadlines. One popular measure of the performance of a soft real-time system is the percentage of jobs that make their deadlines.

Traditional hard real-time systems handle jobs with stringent timing constraints. Job deadlines are guaranteed by a careful but often complex system design and, in many cases, by using expensive specialized hardware (e.g., a real-time network). It is clear that systems like the space shuttle require specialized real-time hardware that can guarantee timely processing of jobs. However, there are other *soft real-time* applications for which meeting every single job deadline causes poor utilization of system resources, is unnecessary, or is even impossible. As an example of a soft deadline, consider a digitized voice packet. For good playback, it must be delivered by a certain time across a network. Yet, considering a long stream of packets, it is tolerable to miss a few packets. A second example is a banking transaction processing system. A typical requirement for such a system may be that 95% of transactions complete in less than 2 seconds [GR93]. Here again, some missed deadlines (up to 5%) may be tolerable. A real-time system that does not guarantee that every single job's deadline will be met, but is engineered to make its best effort in meeting most of the deadlines is called a *soft real-time system.*

The population of soft real-time applications has been growing at a tremendous rate. In fact, nowadays, there are many more soft real-time applications than hard real-time ones. Many applications that are not real-time in nature may also take advantage of real-time technology. Even in cases where end-users have no deadlines, it may still be highly desirable to make applications more responsive to user sensitive timing requirements. For example, in a distributed database, it may be desirable to deliver messages involved in a two-phase commit in a short time frame, to reduce the "window of vulnerability" where blocking can occur [BHG87] and to reduce the time locks are held. Packets for read-only queries would have less stringent timing requirements. Packets for remote view update and quasi-copies [LHM$^+$86] would have even less stringent requirements. Similarly, a disk I/O system should be designed in such a way that hot-spot accesses be done promptly to avoid long queues and improve response time. By allowing users to specify the timing requirements of their jobs and by having the system response to them, many applications can benefit from a well-designed soft real-time system.

Many real-time systems are also distributed systems. An example could be found in stock market analysis and program trading (see Figure 1.1). In a program trading environment, there are typically a number of financial markets (e.g., the New York Stock Exchange) in which various corporate securities and financial instruments are being traded. A trader, such as a mutual-fund company, makes money by trading these commodities. To survive, a trader needs to continuously monitor the markets' condition and react in real time to market information (e.g., price quotes). The market data is stored in various databases and is analyzed by an expert system that spots trading opportunities. The expert system may also trigger certain buy-and-sell actions to realize a profit.

There are two interesting features to observe in our program trading example.

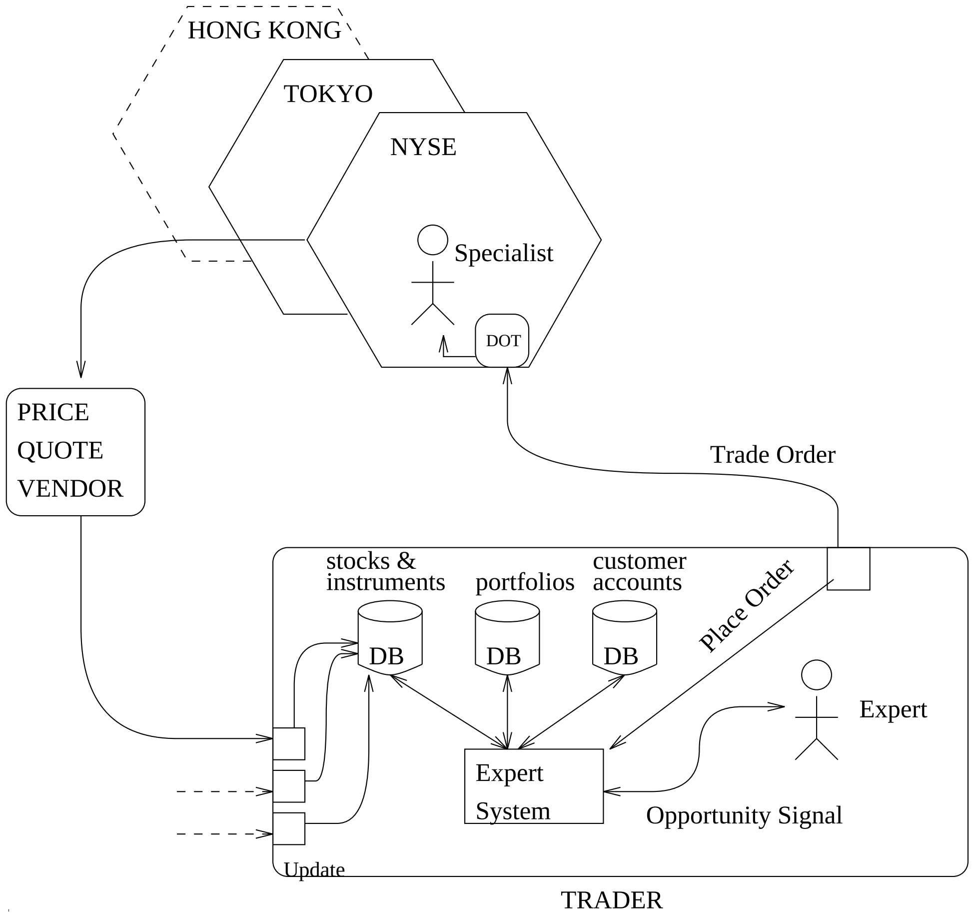


Figure 1.1: A Program Trading Environment.

First, the system involves many components of different types: multiple information sources supplying streams of financial data, a network component (represented as links in the diagram), multiple database servers (probably I/O bound), and an expert system (probably CPU bound). Second, there are global tasks that consist of multiple stages involving work at multiple components. These global tasks may also be associated with soft end-to-end deadlines. For example, a system designer may specify that no trade order issued by the system should depend on any data item in the database that is more than 5 minutes old. In that case, the following sequence of global activity: the receipt of an update, the installment of the update into the database, the expert system computation triggered by the update, and the placing of a buy-sell order, has to be completed within the 5 minutes time frame.

When designing a distributed real-time system such as the program trading example above, one approach is to build it from scratch, coding each component with algorithms and features that match the applications' needs. The main advantage of this approach is that we can program the components to speak the same language and implant hooks in them such that the scheduling of their tasks can be affected by outside control. Components can now communicate with each other easily and global cooperative scheduling is possible. (See Chapter 2 for some examples of cooperative scheduling.) However, a drawback of this approach is that the development and maintenance cost of such a system will be extremely high, and the development cycle will be very long.

Alternatively, one can build the system by piecing together well-tested, easy-to-maintain and much cheaper standard components, such as a commercial database server, a token ring network, etc. A disadvantage of this approach is that standard components are usually not designed to handle real-time tasks. As an example, commercial database systems do not compare the deadlines of conflicting transactions when granting a lock request, which may result in priority inversions[1]. Even if standard real-time components are available (e.g., commercial real-time operating systems), they may vary in their ability to meet the real-time requirements of the tasks. Also, these autonomous components may not provide any external control over their schedulers. This makes global real-time scheduling algorithms impossible because they demand cooperation and coordination among the local schedulers. Given these difficulties, is it possible to construct a distributed system out of a variety of conventional non-real-time and real-time components and still meet real-time constraints? Even if we cannot meet all real-time constraints, can we ensure that "almost all" are met? In this thesis we address these questions.

Depending on how closely the components' local schedulers are cooperating and their ability to service real-time tasks, we see a spectrum of cost-performance tradeoffs (Figure 1.2). At the expensive end of the spectrum, every component in the system understands the tasks' real-time constraints. Their schedulers are programmed to cooperate with each other to achieve some global scheduling algorithm. The execution order of tasks in this scenario is tightly controlled and scheduling is in an orchestrated fashion. High performance, in terms of meeting the timing constraints of the tasks, can be achieved.

Unfortunately, a tight control over the local schedulers often results in a *closed* system design — with a fixed set of hardware and software services. Besides the high development cost, the addition of a system component, for example, may require significant effort to restructure the scheduling routine. On the other hand, for many soft real-time applications, keeping an *open* system may also be an important design

[1]Priority inversion refers to the situation in which a more urgent task is blocked by a less urgent one.

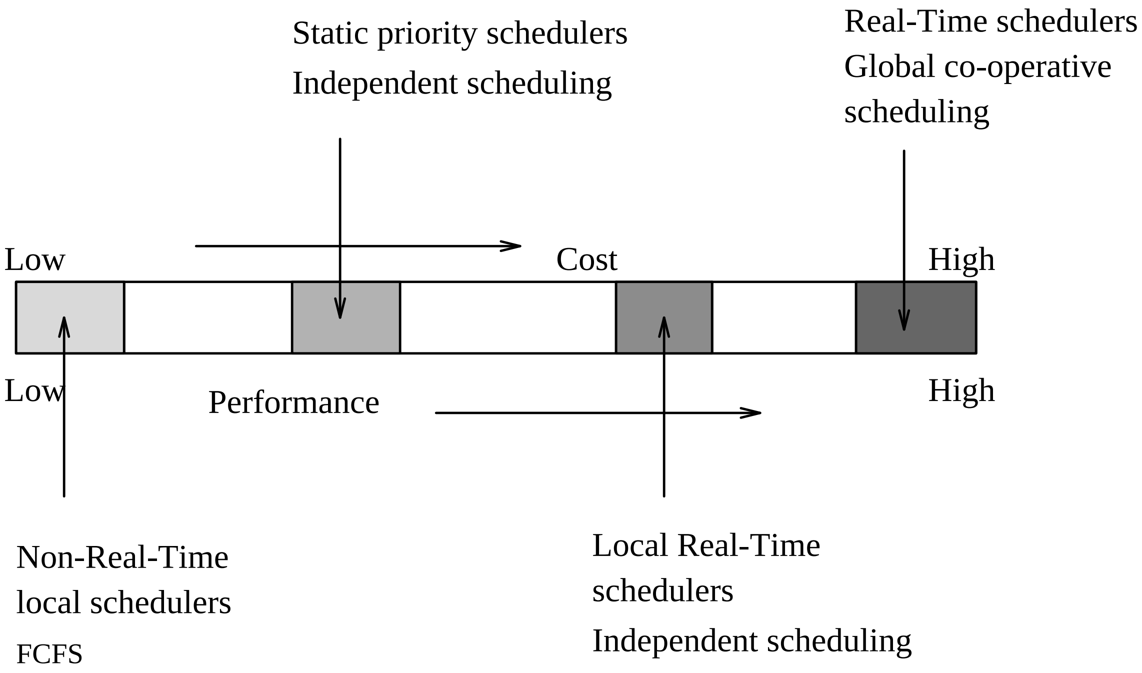


Figure 1.2: Cost-Performance tradeoff.

issue. In an open-system environment, system components are relatively independent and they communicate through simple, well-defined interfaces. *Openness* brings us the flexibility of painless upgrades of the system hardware and software. It also lets us enjoy the benefits of using standard off-the-shelf components and frees us from the restriction of a single vendor. Due to competition, standard components are advancing at a faster pace than customized ones. Also, large systems require the use of the large commercial code base already written for standard components to cut cost [Car95]. Above all, an open system design is a more economical way of system development. Thus, in the other scenarios of the spectrum, we consider the case when all the system components are autonomous commercial off-the-shelf components and each one schedules its tasks independently.

In this thesis, we consider three cases of independent schedulers corresponding to different scheduling capabilities. In one scenario, we assume that all local schedulers understand real-time constraints and they schedule their tasks accordingly (e.g., by employing the earliest-deadline-first algorithm). In other two scenarios, we consider that the schedulers are unsophisticated and are oblivious of deadlines. We will look at both first-come-first-served and static-priority local schedulers. Before we proceed, we state some premises on which our study is based:

- We focus on *soft real-time* systems. In such systems, a primary performance goal is to meet as many deadlines as possible, but unlike hard real-time systems, there is no absolute guarantee that all deadlines will be met. There are two reasons why we look at soft (instead of hard) real-time systems. First, the kinds of distributed tasks we are considering are quite complex, involving multiple stages of processing. Therefore, it is difficult to get the accurate running time estimates that are required for hard real-time scheduling. Second, in many applications it is undesirable or impossible to place an upper bound on the load. The behavior of the system is thus hard to predict. Both problems make hard real-time scheduling impossible to achieve.

- Each component's scheduler is independent. There is no global scheduler that instructs each scheduler what to do. Each scheduler makes decisions based solely on the subtasks that have been presented to it for execution, without consulting other schedulers. We believe that large systems are built out of preexisting components. Each component will have its own scheduling policy and will be unable or unwilling to coordinate or subordinate its scheduling decisions with (or to) others.

- We look at *dynamic, incremental* scheduling, as opposed to a-priori scheduling when the tasks are defined; Tasks are scheduled when they are released for execution. Dynamic scheduling is more appropriate for the type of systems we study where the types of tasks or their durations may be unknown in advance. Also, when the system provides distribution transparency (e.g., whether a piece of data item is available locally or remotely), the subtasks to be created are unknown until run time and thus off-line pre-analysis is not appropriate.

- Each system component is unique. If a task is scheduled to run at a particular component, it must run there. There is no load balancing, i.e., an overloaded component cannot ship tasks to other components.

1.1 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we survey related work concerning distributed real-time system scheduling. In Chapter 3, we describe the model on which we study the various scenarios. This includes both the system model and the task model.

In Chapter 4, we look at the scenario when all the (independent) local schedulers employ real-time scheduling algorithms such as earliest-deadline-first (EDF). We define and study the *subtask deadline assignment* (SDA) problem: For the real-time local schedulers to be effective, the tasks submitted to them must carry deadlines that truly reflect their urgency. A distributed real-time task usually consists of a number of subtasks to be executed at different system components with certain precedence constraints among the subtasks. An overall end-to-end deadline is usually specified by an application designer indicating when a distributed task is to be finished. The SDA problem concerns how an end-to-end global deadline is automatically translated to the deadline of each individual subtask. In this chapter, we examine several deadline assignment strategies. We show how one should choose an assignment strategy and how these strategies improve the real-time characteristics of the system.

When one is building a complex distributed real-time system with components of very different nature, one may not have the luxury of employing real-time schedulers at every single component (e.g., due to cost, legacy reasons). In Chapter 5, we look at the scenario when some of the local schedulers can only perform first-come-first-served (FCFS) scheduling. We study how the system performance degrades comparing with an all EDF configuration. We also discuss how performance is improved when some FCFS components can be duplicated. We introduce a load balancing technique called *server prioritization* that can significantly reduce task missed-deadline rate.

Sometimes, even though a system component is incapable of performing real-time scheduling, it may be able to do better than FCFS by allowing tasks to be assigned a static priority. Some UNIX systems, such as HP-UX, provide users with a number of real-time priority levels. Token-ring networks also schedule packets according to their priorities. In Chapter 6, we study the scenario when the local schedulers are static priority schedulers. We discuss a technique called *real-time emulation* (RTE). Our idea of RTE is to build a software layer on top of a static priority scheduler that translates task deadlines into priority levels in such a way that the behavior of the scheduler is guided by these priorities to mimic the earliest-deadline-first scheduler (see Figure 1.3).

Finally, in Chapter 7 we present a conclusion and future work to this thesis.

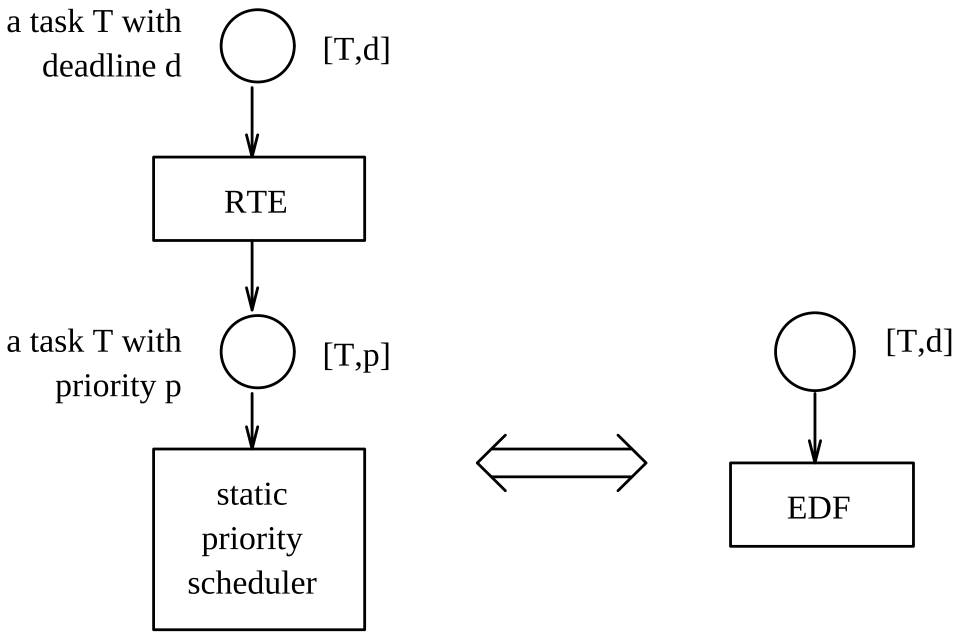


Figure 1.3: The idea of RTE.

Chapter 2

Related Work

The problem of scheduling real-time distributed tasks has been studied mostly in a hard real-time environment (e.g., on multiprocessor/distributed systems [SRC85, RSS90, WRS92], flow shop [BL90, BL92], pipeline scheduling [L^+87], and distributed tasks with precedence constraints [HS92, Ram90, CSR86, CL87]). Some of these works concentrate on the task-allocation/load-balancing strategies, and on the schedulability of tasks.

For example, in [CSR86], the authors studied the problem of dynamic scheduling and allocation of groups of tasks with precedence constraints in distributed hard real-time systems. They consider a homogeneous system (i.e., tasks can execute at any node in the system) on which "task groups" are executed. A task group consists of a number of subtasks with precedence constraints among them. Each task group must finish by a specified (end-to-end) deadline but there is no explicit timing constraint on each individual subtask. Tasks are nonpreemptible in this model. The goal is to schedule and allocate the subtasks to nodes in the system in such a way that the end-to-end deadlines are met.

Their scheduling algorithm consists of a number of components. For instance, there is a *group scheduler module* which, when given a task group, determines whether the group can be scheduled at the local node. A group is schedulable at a node if there exist an execution order of all the tasks of that node with all the deadline constraints satisfied. If the group cannot be scheduled locally, a *preprocessor module* will be invoked to break up the task group into smaller *clusters*. Each cluster is then assigned a *pseudo window* of time in which the cluster should be executed. These clusters will then be distributed to other nodes in the system. Each node will evaluate the schedulability of the clusters submitted to them. Finally an execution plan for the task group is constructed. In case a cluster is not schedulable, its pseudo window is adjusted appropriately, or an attempt will be made to offload the cluster to another node in the system.

Besides deadlines, the scheduling algorithm also considers the communication overhead between subtasks when a task allocation decision is made. For example, subtasks that communicate a lot are usually "clustered" and scheduled to be executed at the same node. The algorithm is also efficient with linear time complexity.

In the above studies, the focus is on coordinating the schedulers at the multiple components closely to ensure job deadlines are always met. System components are assumed to have similar capability and they share the load. Scheduling is done in an orchestrated fashion. On the other hand, our study focuses on *open systems* which are built out of pre-existing components of different nature. A database server, for example, will not give up its cycles to help out an image processing node. We also assume a much looser control over the local schedulers. In particular, each component employs its own local scheduling routine. Cooperately global scheduling and load balancing are therefore not appropriate in this environment. Also, information about tasks, such as their (worst case) execution time, is assumed not known in advance. Off-line exhaustive planning of task schedule is therefore not considered.

We have stated that the problem of deriving sub-deadlines for subtasks from an end-to-end global deadline (SDA) is an important one in a heterogeneous autonomous system. Chapter 4 is devoted to a study on SDA. There are relatively few studies on the SDA problem [KGM92]. However, we would like to mention three studies that are closely related to our approach and SDA. Bettati and Liu [BL90, BL92] discuss the problem of scheduling subtasks in a *hard* real-time distributed environment. In their study, nodes in the system represent different processing element responsible for different functions. Their work focuses on those systems for which global tasks can be characterized as "flow shops." In their base model, all global tasks consist of the same set of subtasks to be executed on nodes in the same order. Each global task is associated with a deadline. The goal is to devise efficient *off-line* algorithms (assuming all the information about global tasks, such as their arrival times, amount of execution time etc., is known ahead) for computing a schedule of the subtasks such that all deadlines are met (if such a schedule exists). This end-to-end scheduling problem is called the flow-shop problem.

The general flow-shop problem is $\mathcal{NP}$-hard. In [BL92], two special cases of the flow-shop problem are identified and it is shown that the earliest-effective-deadline-first (EEDF) algorithm is optimal for these cases. (By optimal, they mean that the EEDF algorithm can find a schedule of the tasks which meet all the deadlines if such a schedule exists.) These two cases are: (1) when all the subtasks have identical execution time, and (2) when all the subtasks submitted to a common node have identical execution time but subtasks that are submitted to different nodes may have different execution time. For the general ($\mathcal{NP}$-hard) case, the authors suggest and evaluate an efficient heuristic algorithm. Simulation results show that the heuristic algorithm is very effective (has a high chance of finding a feasible schedule) when tasks are given a reasonable amount of slack or when tasks are of similar length. Other variations of the base model, like periodic tasks, are also discussed in the paper.

Our model differs from the flow-shop model in many ways. First of all, we assume that the task sets are unknown in advance. Off-line scheduling is thus impossible in our model. Also, nodes in our model may use very different local scheduling algorithms, and are not under the control of any external force. Finally, global tasks are not required to have the same set of subtasks and the subtasks are not required to execute in the same order.

Another interesting paper [PLC92] by Pang, Livny, and Carey investigates the problem of "bias" against longer transactions under "earliest-deadline-based" scheduling policies in *real-time database systems*. The study shows that long transactions miss more deadlines compared to short ones. This is not because of tighter timing constraints (the phenomenon occurs even when long transactions have larger amount of slack), but because of their bigger size, which causes them to have "further-in-the-future" deadlines. Since the earliest-deadline-first algorithm schedules tasks according to their deadlines, long transactions are competing unfavorably against short transactions in accessing system resources.

Their approach to the problem is to assign *virtual deadlines* to all transactions. A transaction with an earlier virtual deadline is served before one with a later virtual deadline. The virtual deadline of a transaction is adjusted dynamically as the transaction progresses, and is computed as a function related to the size of the transaction. In some sense, their approach is to introduce another bias factor counter-attacking the intrinsically biased behaviour of earliest-deadline-based scheduling policies. By monitoring the system and gathering runtime statistics, a parameter in computing virtual deadlines is carefully adjusted in such a way that the linear correlation between miss ratio and transaction size is minimized.

Our work is similar to [PLC92] in that we both try to assign earlier deadlines to transactions (or tasks). However, in their case there is a single scheduler for a single database system. For our problem, there are multiple "resources" handled by independent schedulers. Furthermore, distributed tasks have natural breaks (subtasks) that make the assignment of deadline more natural.

In [PSS$^+$94], the problem of assigning deadlines to *triggered transactions* in a real-time active database system is studied. Triggered transactions are of three types: immediate, deferred, and independent. The authors suggest a number of policies for assigning priorities (deadlines) to immediate and deferred transactions from information such as the deadline of the triggering transaction, the number of triggered transactions and their duration etc. The study shows that different policies excel in different situations. One important factor to consider when choosing a deadline assignment strategy is whether immediate or deferred transactions are dominating in the system. The study also considers the effect of data contention. The result shows that the relative performance of the different policies are not affected to a large extent by the presence of data conflict.

In Chapter 5, we discuss the case when some components in the system can only perform FCFS scheduling. One way to improve the performance of a real-time system with FCFS schedulers is to duplicate the FCFS components and perform intelligent load sharing among them. Here, we sample some of the related work done on the load sharing problem.

Systems using dual (or multiple) servers processing a single job stream have been analyzed extensively by queueing theorists [NSM89, SM89, KMST86, LK84, IL88]. These studies usually consider non-real-time systems and focus on the queueing delay of jobs. For example, in [KMST86], a dual-queue system with heterogeneous servers (i.e., servers with different servicing capacity) in which a newly arrived job always joins the shorter queue is analyzed. Extension of the study can be found in [IL88, NSM89] in which a threshold-type scheduling policy is evaluated. In their model, a job is always serviced by the faster server unless the faster-server queue is longer than the slower-server queue by a certain threshold. This scheduling policy is called a "threshold-type" policy. The study proves that with an appropriate threshold value, the "threshold-type" policy is optimal in minimizing the expected queueing delay of jobs. Others studies that consider queueing systems employing real-time scheduling algorithms like earliest-deadline-first can be found in, e.g., [Tay92, HTT89]. However, these studies are usually confined to single-queue systems.

There is also work done on real-time communication using multiple paths [ML90, SSS92]. In [ML90], a medium access protocol for integrated voice, video and data traffic, operating on a dual-ring LAN is discussed. The main idea is to use one LAN for packet transmission and the other for scheduling using circulating tokens. Another idea on sending *duplicate* packets over redundant paths to enhance communication timeliness is studied in [CL88, OR88, RS91, GMKB91]. Finally, the use of disk shadowing and disk striping to improve fault-tolerance and I/O system performance is considered in [BG88, GHW90].

Chapter 3

The Model

In this chapter we describe the model we use to study the performance of a distributed soft real-time system under various scheduling scenarios. It consists of two parts: the task model and the system model. In order to study and contrast system behavior under different local scheduler algorithms, we developed and implemented a simulation and performed extensive experiments. The structure of our simulation model follows the conceptual model described in this chapter. We will present the details of the simulation model in the following chapters for each individual study.

3.1 The Task Model

We consider two types of tasks in our system: locals and globals. A local task is one that is executed at one and only one component (e.g., a database server, a network component etc.) A global task, on the other hand, can be quite complex and may involve work at multiple components in the system. In this thesis, we only consider global tasks that are serial-parallel as defined below. As shorthand, we use the notation $T = [T_1 T_2 ... T_n]$ to represent a ***global*** task T that consists of n subtasks, T_1, T_2, ..., T_n, to be executed in *series*. A subtask T_i ($i > 1$) cannot execute before subtask T_{i-1} finishes. We also use the notation $T = [T_1 || T_2 || ... || T_n]$ to represent a global task T consisting of n subtasks T_1, T_2, ..., T_n to be executed in *parallel*. The n subtasks arrive at the same time and task T is considered finished only if all n subtasks finish. Composition of these notations is possible. For example, $[T_1[T_{21}||T_{22}]T_3]$ represents a global task that has three subtasks to be run in series, and the second subtask is itself a global task of two parallel subtasks (T_{21} and T_{22}).

Formally, we define the class of (serial-parallel) global task by the following recursive rules:

GT1: A single task which is part of a larger task and which is to be executed at one and only one component is called a *simple subtask*.

GT2: If $T = [T_1 T_2 ... T_n]$ where T_i's are either simple subtasks or global tasks, then T is a global task.

GT3: If $T = [T_1 || T_2 || ... || T_n]$ where T_i's are either simple subtasks or global tasks, then T is a global task.

For rules GT2 and GT3, if T_i is itself a global task, we say that T_i is a *complex subtask* of task T. Figure 3.1 illustrates the various terms and concepts.

A task X (whether it is a local task, a simple subtask, or a global task) has the following attributes:

$$\begin{aligned} ar(X) &= \text{arrival (or submission) time of } X, \\ dl(X) &= \text{deadline of } X, \end{aligned}$$

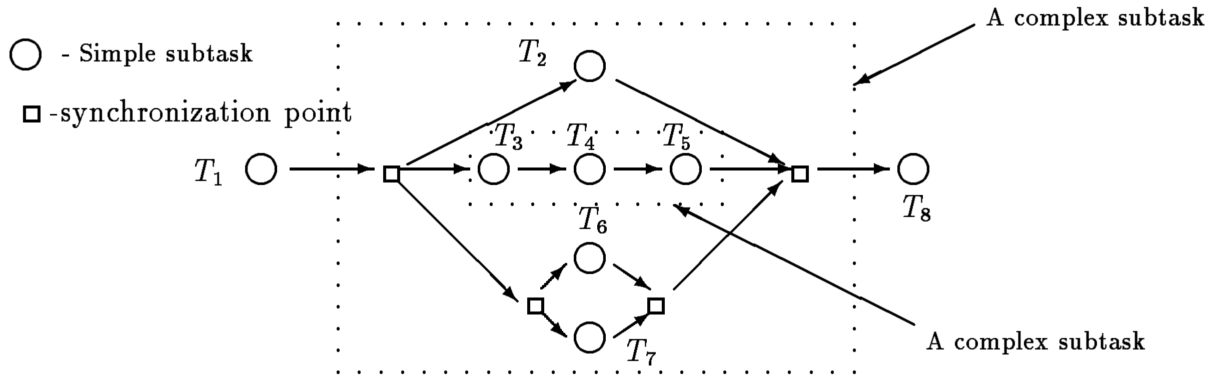


Figure 3.1: A global task example: $X = [T_1[T_2||[T_3T_4T_5]||[T_6||T_7]]T_8]$.

$$
\begin{aligned}
sl(X) &= \text{slack of } X,\\
ex(X) &= \textit{real} \text{ execution time of } X,\\
pex(X) &= \textit{predicted} \text{ execution time of } X.
\end{aligned}
$$

We do not assume the value of $ex(X)$ be available, but some of our algorithms presented later do take advantage of an estimate, $pex(X)$, which is an approximation to $ex(X)$. The slack of a task is the maximum amount of time that the task can be delayed and yet it is possible to meet its deadline. Alternatively, slack can be defined by the following formula:

$$sl(X) = dl(X) - ar(X) - ex(X).$$

We also define flexibility (denoted by $fl(.)$) of a task X to be the ratio of the amount of X's slack to the execution time of X. That is,

$$fl(X) = sl(X)/ex(X).$$

Intuitively, the more flexible a task is (higher $fl(.)$), the less stringent is its timing constraint.

An additional attribute for a subtask is its *stage*, which is defined by:

$$stage(X) = \begin{cases} i & \text{if } \exists T \text{ s.t. } T = [T_1, ..., T_{i-1}, X, ..] \\ 1 & \text{if } \exists T \text{ s.t. } T = [X, ..] \end{cases}$$

We say that a subtask X is an *earlier-stage* than another subtask Y if they are of the same global task and $stage(X) < stage(Y)$.

Furthermore, if X is a simple subtask, we denote $component(X)$ as the execution component of X. That is, X is destined to be executed at $component(X)$.

Finally, there is the issue of tardy tasks, or *overload management policy.* Say a task X has already missed its deadline, but has not completed execution. One option is to abort X as soon as it misses its deadline, under the assumption that whatever it was doing is now useless. (Example: after analyzing the current state of the stock market, a decision is made to sell certain stock. A task X is issued to sell by a given time. If the time is exceeded, it is best to abort X, as the market conditions may have changed.) A second option is to continue to process X, under the assumption "better late than never." (Example: at a bank, customers are "guaranteed" a two second response time. However, if the guarantee is not met, it is still desirable to complete the task.) In this thesis, we only focus on the no-abort case (no specific action is taken when a deadline expires).

3.2 The System Model

Our model of a distributed real-time system consists of a number of *components* (Figure 3.2). These components manage different resources like a database, an expert system, or a compute engine. Even the communication network is considered

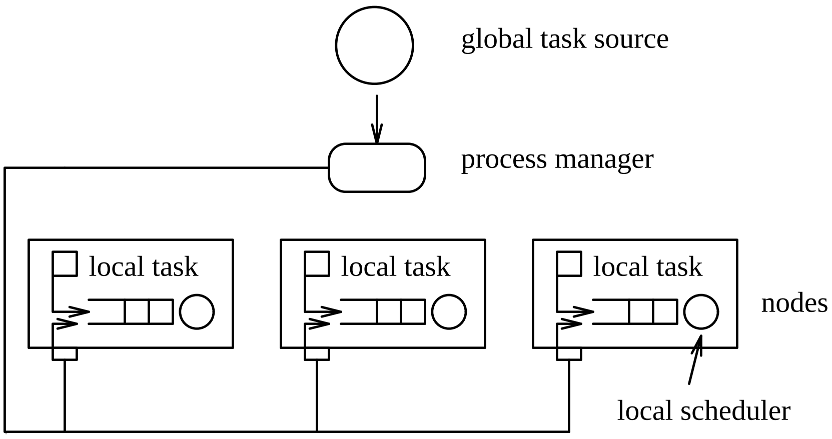


Figure 3.2: The System Model.

a resource and is subsumed as one or more processing components. For example, a direct link between two sites is considered as one resource, while a LAN is considered another. Each component services both local tasks (which are generated at each component) as well as simple subtasks of global tasks. Task service order is scheduled by a local scheduler residing at each component. These schedulers are all independent and they do not collaborate. The only things that *may* influence scheduling decision are the real-time attributes (as listed in Section 3.1) associated with each task.

Newly created global tasks are first processed by a process manager. Figure 3.2 shows a single process manager, but in reality there can be many. Each manager controls one or more independent global tasks. We assume that certain control information of a global task, such as the precedence relationship among the subtasks, and the end-to-end deadline is available to the process manager. The major functions of the process manager are to assign deadlines to simple subtasks, submit the simple subtasks to the appropriate components for execution, and enforce the precedence constraints among the subtasks of a global task. In some cases, the process manager has to generate scheduling information for a subtask before submitting it. For example, if a component A accepts tasks with priority (and schedules them according to the priorities) and another component B schedules tasks according to their deadlines, the process manager has to make sure that the subtasks submitted to the components have the right attributes and the *appropriate* values for these attributes. Part of our study concerns how these values are assigned and what the performance implication of a good assignment strategy is. In reality, the process manager consumes system resources (e.g., communication overhead between the manager and the components,) but this consumption can be considered as additional subtasks, and be handled similarly. Therefore, we do not model the resource requirement of the process manager explicitly.

Chapter 4

Earliest-Deadline-First Schedulers

In this chapter we consider a distributed soft real-time system that is built out of a collection of independent real-time components. In such an environment, global tasks often have processing demands at multiple different components. A distributed global task is usually divided into several subtasks, each to be executed in order at some component. An overall end-to-end deadline is usually specified by an application designer indicating when a distributed task is to be finished. Before a subtask is submitted to a component for execution, however, we must assign a deadline to the subtask since the local scheduler at the component schedules tasks according to their deadlines. We therefore needs a way to translate the end-to-end global deadline into sub-deadlines for each individual subtask. In this chapter, we examine (through simulations) several strategies for subtask deadline assignment in a distributed soft real-time environment. We show how a user should choose an appropriate assignment strategy and the significant improvement in reducing missed deadlines made possible by these strategies.

4.1 The Subtask Deadline Assignment Problem

In traditional soft real-time applications, a task is considered a single unit of work with a given deadline — the time by which the task should be completed. The system usually schedules tasks according to their deadlines, with more urgent ones running at higher priorities. Over the years, researchers have developed real-time scheduling algorithms for different real-time system components, including the communication network [KSY84, ZR87, KSF91], database [AGM88a], disk I/O [AGM90], and processor [LL73]. One common tacit assumption made by these algorithms is that the deadline of a task truly reflects the urgency of completing the task. As real-time systems evolve, however, "tasks" become "bigger", more complicated, and more frequently possess subtasks to be executed on various system nodes or components. In a distributed environment, local schedulers find themselves scheduling subtasks, or "segments" of global tasks instead of complete, integrated tasks. In most situations, a single value of an end-to-end global deadline fails to capture the sense of urgency of each individual subtask. This severely hampers the efficacy of real-time scheduling algorithms.

As an example of a complex distributed task, let us re-consider the stock market analysis and program trading example shown in Chapter 1 (Figure 1.1). In this application, information on stock prices is gathered through multiple sources and is piped through a series of filters for refinement. The information is then used by an expert system that spots trading opportunities. This latter stage may involve extensive database operations and knowledge rules processing. A profit may then be realized by the appropriate buy and sell actions. While the deadlines for high-level tasks are usually given as a part of the system specification (e.g., a buy-sell action should be implemented within 2 minutes from the time when the information is gathered), we lack a methodical way of assigning deadlines to the individual subtasks (e.g., how much time should we give a database search? a disk access? a network transmission?). In this chapter, we study the subtask deadline assignment problem (SDA), and suggest guidelines for deriving subtask deadlines from a global task's end-to-end deadline.

To study the SDA problem, we need to understand the structure of global tasks. A global task can be very complex, with arbitrary precedence relationships among its subtasks. Many global tasks, however, fall into the category of serial-parallel tasks, which have a simpler structure (see Section 3.1 for the definition of serial-parallel tasks). For this type of tasks, we can generally reduce the SDA problem into two simpler sub-problems: one deals with serial subtasks, and another one with parallel subtasks. To illustrate the concept, let us consider the following example:

$$T : \left\{ \begin{array}{c} T_{11} \\ .. \\ T_{15} \end{array} \right\} T_2 \qquad \text{deadline of } T = 10.$$

Here, we have a global task T which consists of two serial stages: $\{T_{11}, .., T_{15}\}$, and T_2. The first stage is a parallel task with five subtasks. Each of the six subtasks is to be scheduled at a different system component (e.g., disks, networks, processors). Assuming task T arrives at time 0 and has a deadline at time 10, what deadlines should we assign to the six subtasks? If we use the end-to-end deadline (time 10) as the subtasks' deadlines, a scheduler will be fooled to believe that it has 10 time units to complete the first stage $\{T_{11}, ..., T_{15}\}$. This leaves no *slack* for subtask T_2, and a missed deadline may ensue. An earlier deadline for the first stage should therefore be used instead, but how early should it be? What happens if we have three serial stages instead of two? We refer the problem of breaking up an end-to-end deadline into earlier ones for serial subtasks as the *serial subtask problem* (SSP).

Now suppose we figure out that we should reserve at least 4 seconds for T_2 in order to have a good chance of making the deadline. In this case, we would like the first stage be done by time 6. It seems natural to assign 6 as the deadline for each of T_{11}, ..., T_{15}. However, this turns out to be inadequate. As an analogy, consider a group of friends agreed on meeting at a theatre for a movie. If that is a small group of one or two people, they will probably all arrive before the show. However, if the group consists of 10 members, chances are that someone will show up late and the whole group misses the movie. A similar problem occurs in our example: when a real-time task is being divided up into a number of subtasks for parallel processing, it is very likely that at least one of the subtasks run into a busy component and become tardy. This will cause the whole global task (in our example, the first stage) to miss its deadline (time 6). The *parallel subtask problem* (PSP) deals with assigning deadline to parallel subtasks.

We believe that the SDA problem is a central one in distributed soft real-time systems. Notice that it even encompasses the communication subsystem. That is, sending messages can be viewed as another type of subtask. Also note that the SDA problem may arise *within* a single site that contains multiple subsystems such as the CPU, the disk, and memory. For example, even within a single database system, sub-deadlines may have to be used when requesting a disk block read, a buffer page allocation, or a processor for computation.

As we mentioned earlier, the SDA problem can be divided into two subproblems: SSP and PSP for the class of serial-parallel tasks. Our approach to SDA is to study the two subproblems individually and suggest some possible solutions. These solutions are evaluated using the model described in Chapter 3. We will then combine the techniques to devise an integrated SDA strategy. We start with SSP in next section.

4.2 The Serial Subtask Problem (SSP)

To study SSP, we consider global tasks of the form: $T = [T_1 T_2 ... T_m]$ where the T_i's $(1 < i < m)$ are all *simple subtask*. That is, a global task only consists of a sequence of subtasks to be executed *in order*. (Complex subtasks are considered in Section 4.4.) A SSP strategy is one that determines the values of a *virtual deadline* $dl(T_i)$ $(1 \leq i \leq m)$ at the time when T_i is submitted.

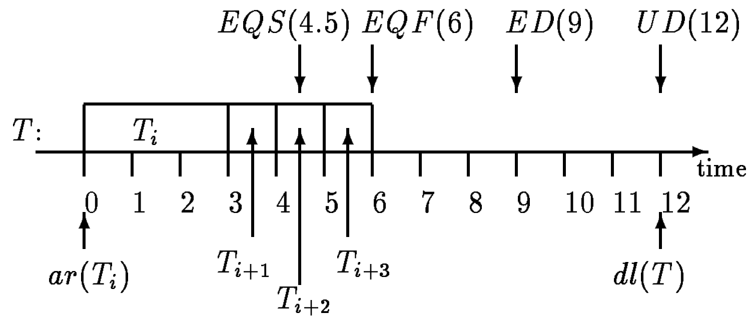
We demonstrate the various SSP strategies by an example shown in Figure 4.1. In the example, the global task T has 4 subtasks T_i, T_{i+1}, T_{i+2}, T_{i+3} remaining. These subtasks have execution times of 3, 1, 1, and 1 units respectively. (We do not assume that the algorithms know these values. They might use an estimate though). Moreover, T_i arrives at time 0 (i.e., T_{i-1} finished at time 0) and the ultimate deadline of T is time 12. The amount of slack remaining for T is 6 units.

Without any knowledge on the execution times of the subtasks, the only available measure of their timing requirement is the deadline of their global task T. A simple SSP strategy would be to set the deadline of a subtask to be equal to the deadline of its global task (12 in Figure 4.1). We call this strategy *Ultimate Deadline* (UD).

(1) Ultimate Deadline (UD):

$$dl(T_i) = dl(T).$$

A problem with UD is that the time for the execution of a later-stage subtask (e.g. T_{i+1} in Figure 4.1) is considered slack to an earlier stage (T_i). This gives the



key: $X(y) \Leftrightarrow dl(T_i) = y$
under SDA strategy X.

Figure 4.1: Example on the various SSP strategies.

schedulers incorrect information about how much time a subtask can be delayed in its execution without causing a missed deadline.

If an estimate of the subtask execution time is available, we can compute the effective deadline of a subtask. The effective deadline of a subtask T_i is equal to the deadline of its global task T (time 12 in the example) minus the total expected execution time of the subtasks of T following T_i (3 units)[1]. Formally, we have,

(2) Effective Deadline (ED):

$$dl(T_i) = dl(T) - \sum_{j=i+1}^{m} pex(T_j).$$

A problem common to both UD and ED is that all the remaining slack of T (6 units in the example) is allocated to the currently active subtask (T_i). This subtask thus has a low priority compared to other tasks of the system. A big portion of this slack may be consumed while the subtask is waiting for its turn in the scheduler queue. Subtasks that represent early stages of global tasks thus consume most of the slack of their global tasks. This leaves little slack for the subtasks to follow. Global

[1]We assume the estimates are perfect in our example.

tasks may therefore have a low probability of meeting their deadlines.

The problem of "little slack for final-stage subtasks" gets worse when there are many local tasks in the system. This is because local tasks have only one, probably short stage. If local tasks have similar flexibility ($sl(.)/ex(.)$) as compared to global tasks, then on average they have smaller total amount of slack than global ones do. When competing for system resources, early-stage subtasks of global tasks will be discriminated against (i.e., scheduled later than other tasks), because of their much larger slack. The scheduler is therefore biased in favor of local tasks at the expense of global ones.

To avoid discrimination and to allow enough slack for final-stage subtasks, each subtask should have its fair share of its global task's slack. One way of doing it is to divide the total remaining slack (6 units in Figure 4.1) equally among the *remaining* subtasks (4 of them). This gives us the *Equal Slack (EQS)* strategy:

(3) Equal Slack (EQS):

$$dl(T_i) = ar(T_i) + pex(T_i) + [dl(T) - ar(T_i) - \sum_{j=i}^{m} pex(T_j)]/(m - i + 1).$$

The term in brackets represents the remaining slack. In our example, EQS would assign $dl(T_i)$ as $0 + 3 + 6/4$ or 4.5. This is shown at the top of Figure 4.1.

A fourth strategy is to divide the total remaining slack among the subtasks in proportion to their execution times. In this way, subtasks of the same global task do not have equal slack, but equal flexibility. We call this strategy *Equal Flexibility (EQF)*.

(4) Equal Flexibility (EQF):

$$dl(T_i) = ar(T_i) + pex(T_i) + [(dl(T) - ar(T_i) - \sum_{j=i}^{m} pex(T_j))] * [pex(T_i)/ \sum_{j=i}^{m} pex(T_j)].$$

For our example, under EQF, $dl(T_i)$ is $0 + 3 + 6 * (3/6)$ or 6.

4.2.1 Simulation Model

In order to study and contrast system behavior under these SSP strategies, we developed a simulation model and performed extensive experiments. In this section we describe the model.

In practice, a distributed soft real-time system can be very complex. For instance, each component may have a different scheduler type, and different execution characteristics. For example, the communications subsystem may schedule outgoing messages using a simple earliest-deadline-first strategy, and its subtasks may take on the order of microseconds to execute. A database server component, on the other hand, may take into account lock contention in its scheduling decisions and subtasks may take on the order of seconds. Similarly, global tasks can be extremely varied, ranging from ones with a few short subtasks, to ones lasting days and spanning many components.

If we model all this diversity and complexity at once, our results would be obscured by many irrelevant parameters. Instead, we *start* with a very simple model that captures the essential features that impact SSP. To pick a particular example, in a given experiment we assume that all global tasks have a fixed number of subtasks, m. Clearly this is not "realistic;" as we have stated, tasks will have a variety of subtasks. If we were trying to predict performance of a particular distributed application, this assumption would not be acceptable. But if we are trying to understand how the number of subtasks affects the choice of SSP algorithm, this is the best assumption. By running different experiments with different values of m, we may be able to see that for short global tasks maybe algorithm X is better than Y. For long tasks, the reverse may be true. The same argument can be made for all the modeling assumptions we are about to make. Our point is that only by selecting a small number of simple, key parameters will we be able to understand the fundamental interactions. Later on, we will relax some of our simplifying assumption (making the model more realistic) and study the impact on our initial conclusions.

Our simulator is written in the simulation language **DeNet** [Liv90]. Each simulation experiment (generating one data point) consists of two simulation runs, each lasting one million time units (at least 100,000 tasks are generated per run, many more for high load experiments). The 95% confidence interval is $\pm$ 0.35 percentage point (much smaller for high load experiments) for the missed deadlines figures shown in later sections.

The structure of our simulation model follows the conceptual model described in Chapter 3 with the following characteristics.

Nodes

There are k components in the system. Each component services their tasks according to some real-time scheduling algorithm with *no preemption*. In this chapter, we use *earliest-deadline-first*[2] as the scheduling algorithm for most of our experiments, mainly due to its popularity. We will also discuss briefly in Section 4.2.3 the case when *minimum-laxity-first*[3] is used as the scheduler algorithm.

Local Tasks

[2] Tasks in a scheduler queue are ordered by increasing deadline; the task with the earliest deadline is served first.

[3] Tasks in a scheduler queue are ordered by increasing slack; the task with the least slack is served first.

Local tasks are being generated *at each component* according to a Poisson distribution with mean interarrival time $1/\lambda_{local}$ time units. Task arrivals are independent. Since there are k components, the total average arrival rate is $k\lambda_{local}$ per unit time. Execution times of local tasks are exponentially distributed with mean $1/\mu_{local}$ time units. The rate of work due to local tasks is thus $k\lambda_{local}/\mu_{local}$. In the experiment , we set $\mu_{local} = 1$, other time measures are thus relativized to the average execution time of a local task. Slack of local tasks is uniformly distributed in the range $[S_{min}, S_{max}]$.

Global Tasks

Similar to local tasks, global tasks are being generated as a *single stream* of Poisson processes with mean interarrival time $1/\lambda_{global}$. In order to simplify our discussion, we hold a simple view that global tasks are homogeneous. In particular, we assume that all global tasks consist of m subtasks and the execution times of the subtasks all follow the same exponential distribution with mean equal to $1/\mu_{subtask}$ time units. The total execution times of global tasks thus follow an m-stage Erlang distribution with mean $m/\mu_{subtask}$. The rate of work due to global tasks is therefore $m\lambda_{global}/\mu_{subtask}$. The execution component of a subtask is picked randomly (and uniformly) from the k components.

We define the term *rel_flex* to be the relative flexibility of global tasks with respect to local tasks[4]. So if the slack distribution of global tasks is uniform in the range $[a,b]$, we can determine a and b by the following equations[5]:

$$\left(\frac{a}{m/\mu_{subtask}}\right) / \left(\frac{S_{min}}{1/\mu_{local}}\right) = rel\_flex, \qquad \left(\frac{b}{m/\mu_{subtask}}\right) / \left(\frac{S_{max}}{1/\mu_{local}}\right) = rel\_flex.$$

That is,

$$a = m \cdot rel\_flex \cdot S_{min} \cdot \frac{\mu_{local}}{\mu_{subtask}}, \qquad b = m \cdot rel\_flex \cdot S_{max} \cdot \frac{\mu_{local}}{\mu_{subtask}}.$$

[4]Recall that the flexibility of a task is defined to be the ratio of its slack over its execution time.

[5]Recall that the slack distribution of local tasks is uniform in the range $[S_{min}, S_{max}]$.

A rel_flex of 1 means that global tasks and local tasks have the same flexibility distribution. A greater (smaller) than 1 rel_flex means in general, global tasks are more (less) flexible than local ones.

System Load

We define the *normalized load* (or *load* for short) to be the ratio of the rate of work generated to the total processing capacity of the system. That is,

$$load = \frac{\frac{m \cdot \lambda_{global}}{\mu_{subtask}} + \frac{k \cdot \lambda_{local}}{\mu_{local}}}{k}.$$

For a stable system, we have $0 \leq load < 1$.

We also define $frac\_local$ to be the fraction of $load$ that is contributed by local tasks. That is,

$$frac\_local = \frac{k \cdot \frac{\lambda_{local}}{\mu_{local}}}{m \cdot \frac{\lambda_{global}}{\mu_{subtask}} + k \frac{\lambda_{local}}{\mu_{local}}}.$$

If the system does not have any local tasks, $frac\_local = 0$; a $frac\_local$ of 1 means no global tasks. As another example, if subtasks and local tasks had similar execution time, then a $frac\_local$ of $1/2$ would mean that there are the same *number* of local tasks and global *subtasks*. However, since it takes m subtasks to make a global task, when $frac\_local$ is $1/2$, there are m times more *local tasks* generated than *global tasks*.

Table 4.1 shows the parameter setting of our baseline experiment. In particular, we assume that the prediction on execution time is perfect, i.e., $pex(.) = ex(.)$ (In Section 4.2.3, the effect of error in the execution time predictions is discussed.) To study the effect of these parameters on system performance, we will vary the parameters from their base settings. This is discussed in the following section.

Overload Management Policy	No Abortion
Local Scheduling Algorithm	Earliest Deadline First
$\mu_{subtask}$	1.0
μ_{local}	1.0
k (# of components)	6
m (# of subtasks of a global task)	4
$load$	0.5
$frac\_local$	0.75
$[S_{min}, S_{max}]$	[0.25,2.5]
rel_flex	1.0
$pex(X)/ex(X)$	1.0

Table 4.1: Baseline setting for SDA experiments

4.2.2 Results

In this section, we summarize the results of our simulation experiments on SSP. The primary performance measure we use to evaluate our algorithms is the percentage of missed deadline (or miss ratio).[6] In particular, we look at the probability of a task missing its deadline conditional on its task class (i.e. global or local). We adopt the notation MD_A^B where MD stands for fraction of missed deadlines, and $A \in$ {local, global}, $B \in \{UD, ED, EQS, EQF\}$ are optional modifiers describing the task class and SSP strategy used. For example, MD_{global}^{UD} denotes the probability that a global task misses its deadline under the UD strategy.

Baseline Experiment

As a starting point, let us look at how the various strategies do relative to each other in our baseline experiment. Figures 4.2(a) and 4.2(b) show MD_{local} and MD_{global}

[6]Result on other measures like average tardiness and waiting time will also be shown to assist our analysis.

of the various SSP strategies as *load* varies from 0.1 to 0.5 (We use dotted lines for MD_{local} and solid lines for MD_{global}. We also associate symbols like $\diamond$ and $+$ to different SSP strategies.)

Comparing Figures 4.2(a) and 4.2(b), we see that even though global tasks and local tasks have the same average flexibility ($rel\_flex = 1$ for the baseline setting), there is a significant difference in their ability to meet deadlines. For example, at load $= 0.5$, $MD^{UD}_{global} = 40\%$ (point A), while $MD^{UD}_{local} = 24\%$ (point B). Also, different SSP strategies miss different numbers of global task deadlines, unless the load is too light (Figure 4.2(b)).

We should point out that traditionally soft real-time systems are studied under high load situations. Most of the time, the system will hopefully operate under low load; no deadlines will be missed regardless of what scheduling policy is used. However, once in a while the system will be overloaded, and it is precisely at those times when we need a scheduling policy that can miss the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Figure 4.2(b) are important.

In order to understand these differences, let us consider the types of resource competition among tasks. Since there are 2 task classes, there are 3 types of contention: local-local, local-global and global-global. A local scheduler resolves contention by comparing the deadlines of tasks. Since a SSP strategy only affects subtask deadlines, it only impacts local-global and global-global contention. The reason why local tasks are not affected by the SSP strategy (Figure 4.2(a)) is that in our baseline experiment, 75% of the load is contributed by local tasks. Thus, much of the contention faced by local tasks is local-local, and unaffected by the SSP strategy. On the other hand, global tasks face local-global and global-global contention, so the choice of a

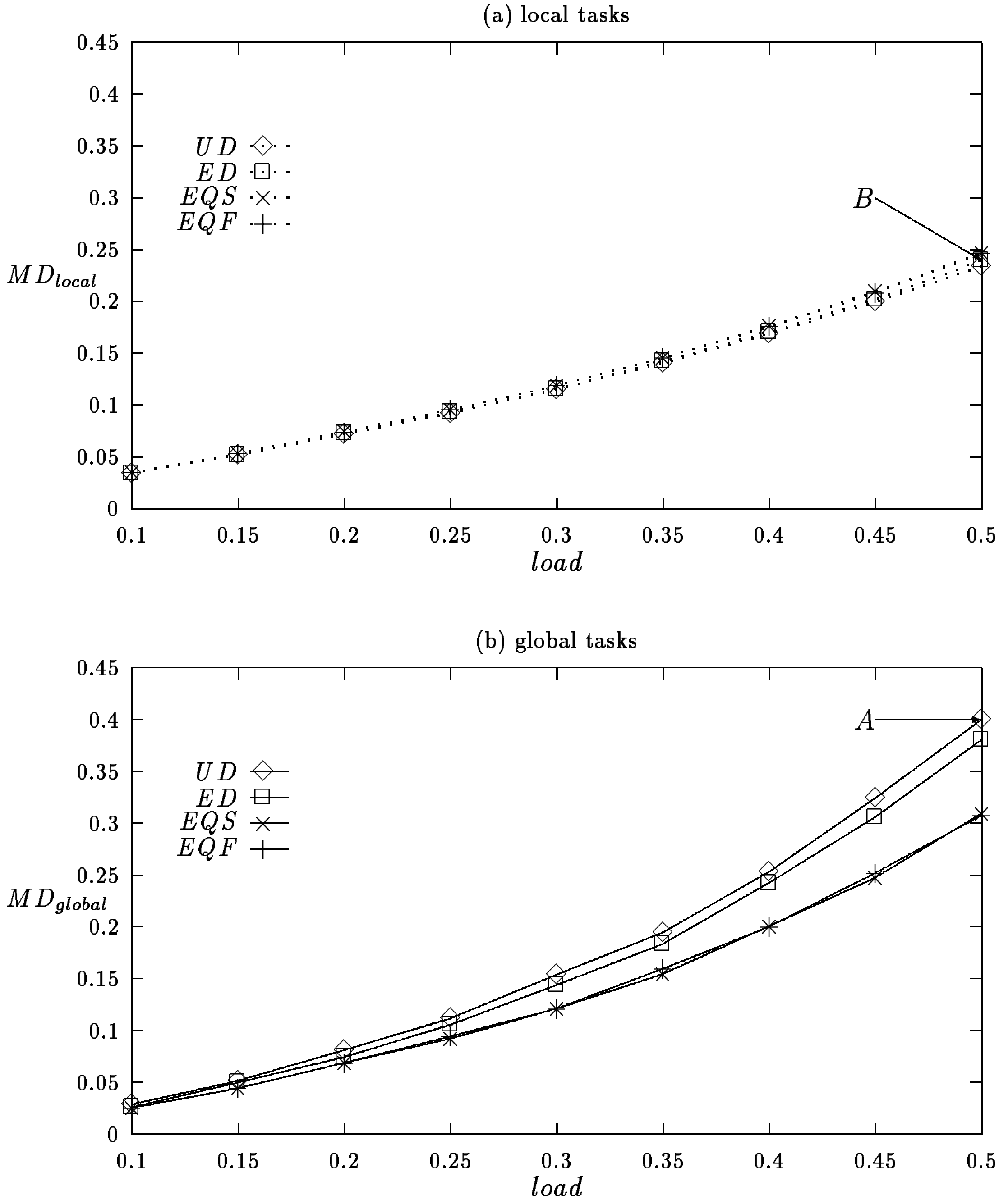


Figure 4.2: Performance of various SSP strategies in the baseline experiment.

SSP strategy affects them significantly (Figure 4.2(b)).

Through extensive simulation experiments (not shown here), we observed (and this is confirmed in Figure 4.2) that the performance of ED lies between that of UD and EQF. We also observed that EQS's performance is very close to that of EQF over a wide range of parameter settings. In cases when they differ, EQF usually is superior. Thus, in order to simplify our presentation, we will exclusively focus on UD and EQF in the rest of this Section.

UD Vs EQF: Their different treatment of global tasks

To further compare UD and EQF, we overlap Figures 4.2(a) with 4.2(b), and remove the curves for ED and EQS. The result is plotted in Figure 4.3. From this figure, we make the following three observations:

(I) Under UD and high loads, global tasks miss many more deadlines than local tasks.

(II) Strategy EQF significantly improves the performance of global tasks (high load), but still local tasks have a better chance of meeting their deadlines.

(III) Comparing the local tasks under the two strategies (dotted lines,) UD gives a lower local task missed deadlines figure than EQF does. The difference, however, is very small.

Observation (I) is not surprising. We had hypothesized that UD would assign too much slack to early subtasks, resulting in a scheduling discrimination when they competed against local tasks. However, observation (II) is surprising because by assigning global tasks flexibility that was comparable to that of local tasks, we expected

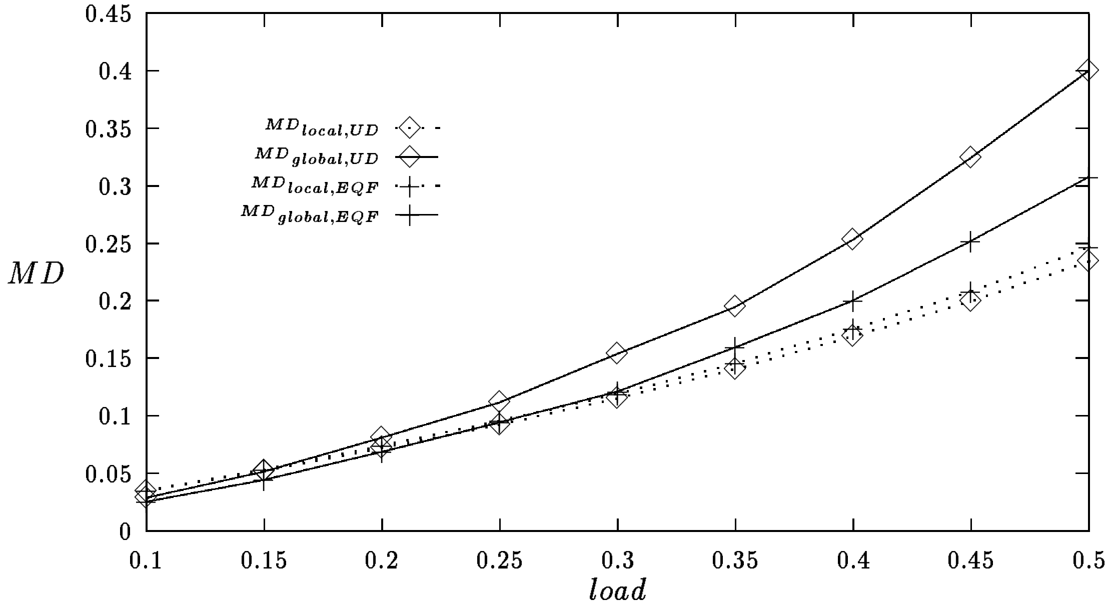


Figure 4.3: Comparing UD and EQF in baseline experiment.

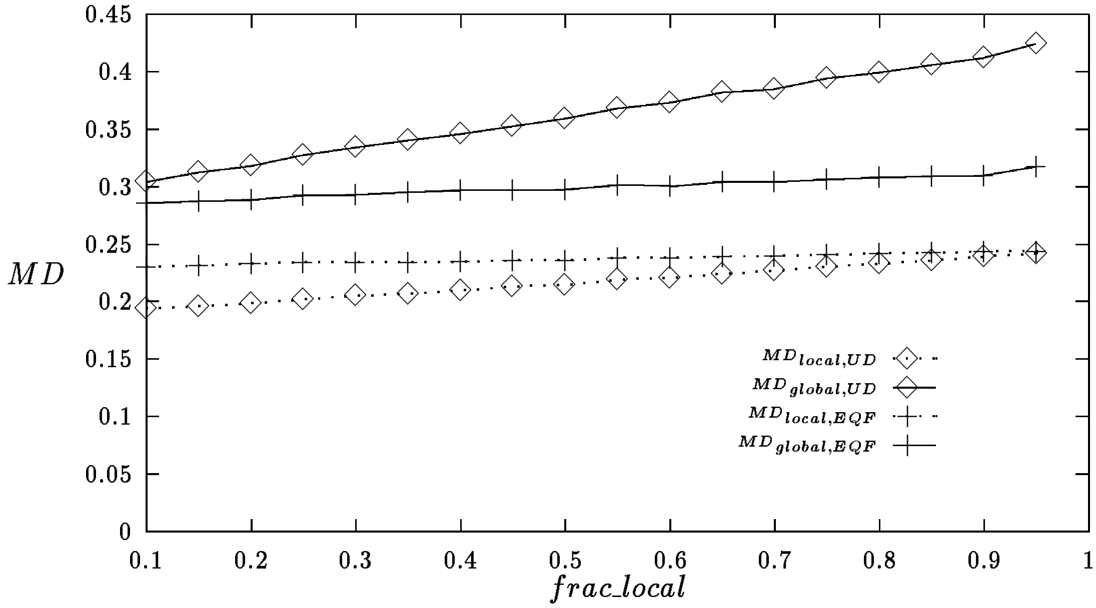


Figure 4.4: Effect of varying the fraction of local tasks.

a "fair playing field." Unfortunately, in spite of our efforts, global tasks somehow are still hurt.

Before studying the causes of observation (II), let us confirm the hypothesis behind observation (I). Let us call this hypothesis the *discrimination hypothesis*: by giving early subtasks of global tasks too much slack, UD makes global tasks "second class citizens" as they compete with local tasks. To confirm the hypothesis, in Figure 4.4 we vary the relative proportion of the two task classes, i.e., we vary *frac_local* from 0.1 to 0.95[7].

[7]Recall that *frac_local* is the fraction of *load* contributed by local tasks (see definition on page 34).

In the figure we see that as $frac\_local$ increases (fewer global tasks), indeed MD^{UD}_{global} increases. This is because global tasks face more and more conflicts with local tasks, and are discriminated against more and more. Notice that MD^{UD}_{local} also increases (although to a smaller extent): this is because local tasks are also facing more and more conflicts with "first class" tasks. On the other hand, observe that the MD^{EQF}_{local} and MD^{EQF}_{global} values hardly change as $frac\_local$ varies. This is because EQF does not discriminate against global tasks.

As another observation on Figure 4.4, notice that when $frac\_local$ is close to zero (almost all tasks are global), UD and EQF miss similar number of global task deadlines (the two solid lines merge). Strategy UD is still assigning too much slack to global tasks, but since everybody is a "second class citizen," it does not really matter. Hence, in an application where most tasks are global, UD may be an adequate (and simpler) strategy.

Observation (III), namely, $MD^{UD}_{local} < MD^{EQF}_{local}$, can be explained by the fact that EQF assigns earlier deadlines to subtasks than UD does. The subtasks can therefore compete better with local tasks. Local tasks thus miss more deadlines under EQF than under UD. In Figure 4.4, the gap between MD^{UD}_{local} and MD^{EQF}_{local} (the two dotted lines) widens as the fraction of local work *decreases*, for more subtasks enter the arena fighting for resource with local tasks.

Figures 4.5 and 4.6 show the average task waiting time (ω) and the average tardiness (τ) of *tardy* tasks respectively in our baseline experiment. The tardiness of a tardy task is the amount of time by which the task misses its deadline. The waiting time of a task is the amount of time it spends waiting in queue. With discrimination in effect, global tasks spend more time waiting in queues under UD (e.g. 25% more at load = 0.5), causing more conflicts and longer response time. As a result, tardy

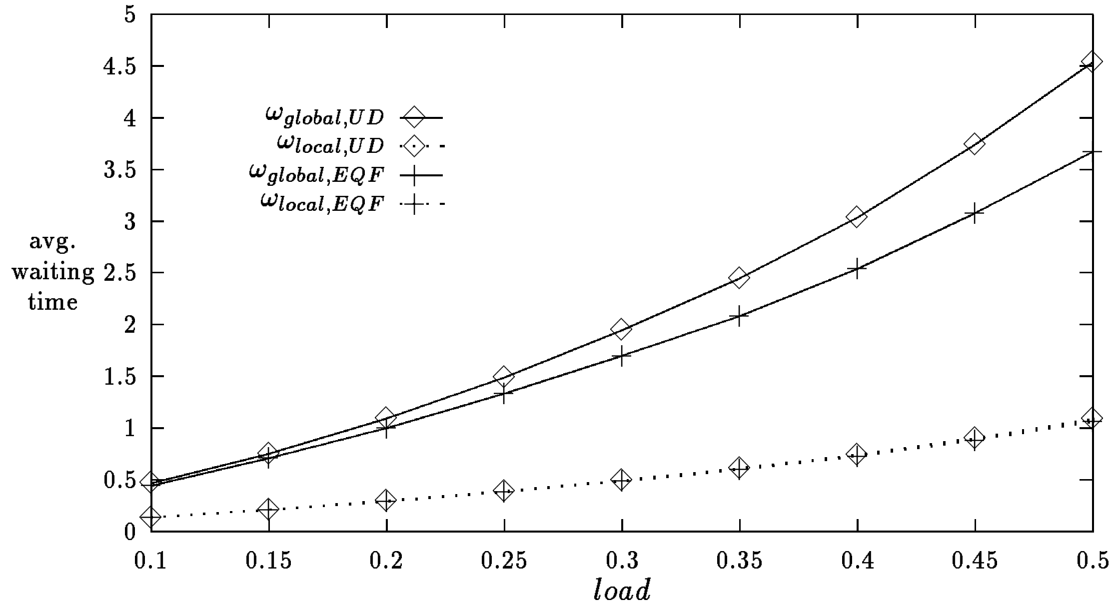


Figure 4.5: Task waiting time in baseline experiment.

global tasks tend to miss their deadlines by a wider margin under the UD policy than under EQF (Figure 4.6). EQF thus performs better than UD in both reducing the miss ratio and the tardiness of global tasks.

Why $MD^{EQF}_{global} \neq MD^{EQF}_{local}$?

If there is no discrimination under EQF, then why are global tasks still performing poorly (observation (II))? The reason is that global tasks consist of a *series* of subtasks and there are two phenomena affecting the series. One phenomenon is beneficial to global tasks: if one subtask finishes early, its leftover slack is inherited by the subtasks that follow. Thus, later subtasks will tend to have even more slack. The second

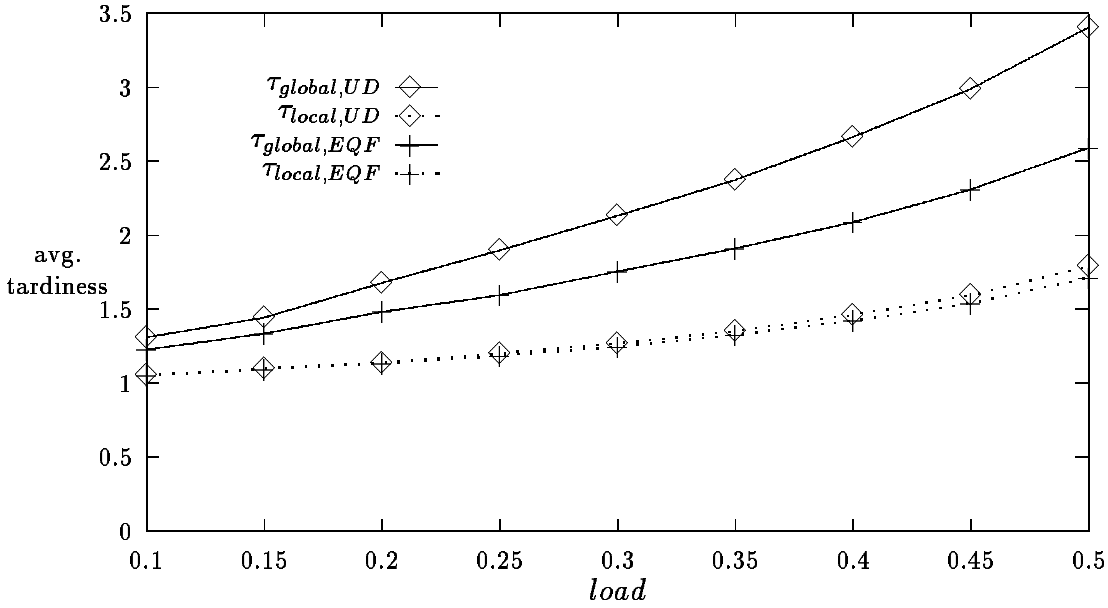


Figure 4.6: Task tardiness in baseline experiment.

phenomenon is detrimental to tight tasks. If a global task has little slack to begin with, it may miss an early sub-deadline, robbing slack from the following subtasks, and making things even worse for them. Essentially, "the poor get poorer while the rich get richer."

To observe these phenomena, we ran an experiment with the same parameter values as our baseline experiment except that the number of subtasks of a global task (m) is now 14 instead of 4. In Figure 4.7, we plot the average slack of a subtask versus the stage of the subtask (the '$\Box$' line). For example, point A shows that the average slack of a 10th stage subtask is 2.3 units in our experiment. Also shown in the figure is the range of the slack data points with the largest 2.5 percentiles and the smallest 2.5 percentiles removed. The graph thus shows, for example, that the "middle" 95% of all 13th-stage subtasks have slacks in the range [-5.7, 19.7].

We can see that the average slack of a subtask increases as it advances. This is due to the beneficial phenomenon. However, the variability in slack increases, due to the detrimental phenomenon. Even though there is more slack on the average, there is a group of tasks that are suffering (the "poor") and are more likely to miss their ultimate deadlines. This is suggested by the fact that part of the 95% range bars lie below the x-axis. (As an analogy, even if the standard of living is increasing, more people may be under the poverty line.)

Depending on the system parameters, the beneficial and detrimental phenomena may have more or less force. In Figure 4.3, as the load increases, there are more global tasks that are "poor" to begin with (their initial stage subtasks miss their deadlines or have little amount of slack to be inherited by its followers). Thus, the detrimental effect causes more global tasks to miss their deadlines. As the load decreases, however, there are fewer poor tasks, and global tasks miss roughly the same number of deadlines

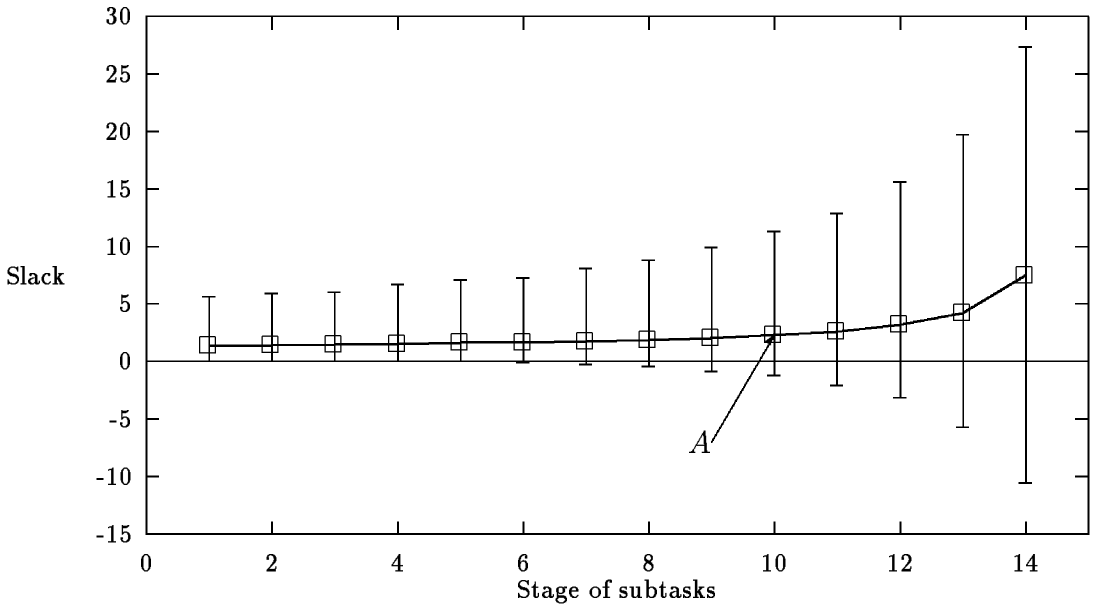


Figure 4.7: Average and the mid-95 percentile range of slack of subtasks versus stage of subtasks.

as local tasks. As the load decreases further, the beneficial effect may actually cause global tasks to do *better* than local tasks. This effect can barely be seen in Figure 4.3, but will be seen clearly later in other graphs.

The effect of slack and the number of subtasks of a global task (m)

To evaluate the gains of EQF over UD, we varied over wide ranges all of our model parameters (Table 1). Here we report on two of the more interesting experiments, involving the slack and the number of subtasks of a global task (m).

Figure 4.8 shows the result of varying S_{max} from 1.25 to 11.25 for the baseline setting. To improve legibility, we divide the graph into two parts: 4.8(a) and 4.8(b), showing tight and loose slack conditions respectively. Note that these subfigures use a different MD scale.

As expected, Figure 4.8 shows that as slack (S_{max}) increases, tasks of all types enjoy a lower miss ratio. As in Figure 4.3, MD_{local} is not affected significantly by the subtask deadline assignment (SSP) policy (75% of the load is local tasks, and their local-local conflicts are not affected by SSP).

The choice of a SSP strategy, however, strongly affects global tasks (solid lines). When slack is large (Figure 4.8(b)), UD consistently misses about 50% more global task deadlines than EQF does. As slack tightens (Figure 4.8(a)), global tasks miss more deadlines, and the gap between EQF and UD narrows, eventually closing when there is effectively no slack and most of the deadlines are missed.

Comparing the MD^{EQF}_{local} and MD^{EQF}_{global} curves ("+"), we observe the same effect of Section 4.2.2. With tight slack, there are more "poor" tasks that end up missing their deadlines because of the detrimental effect of serial subtasks. However, with more

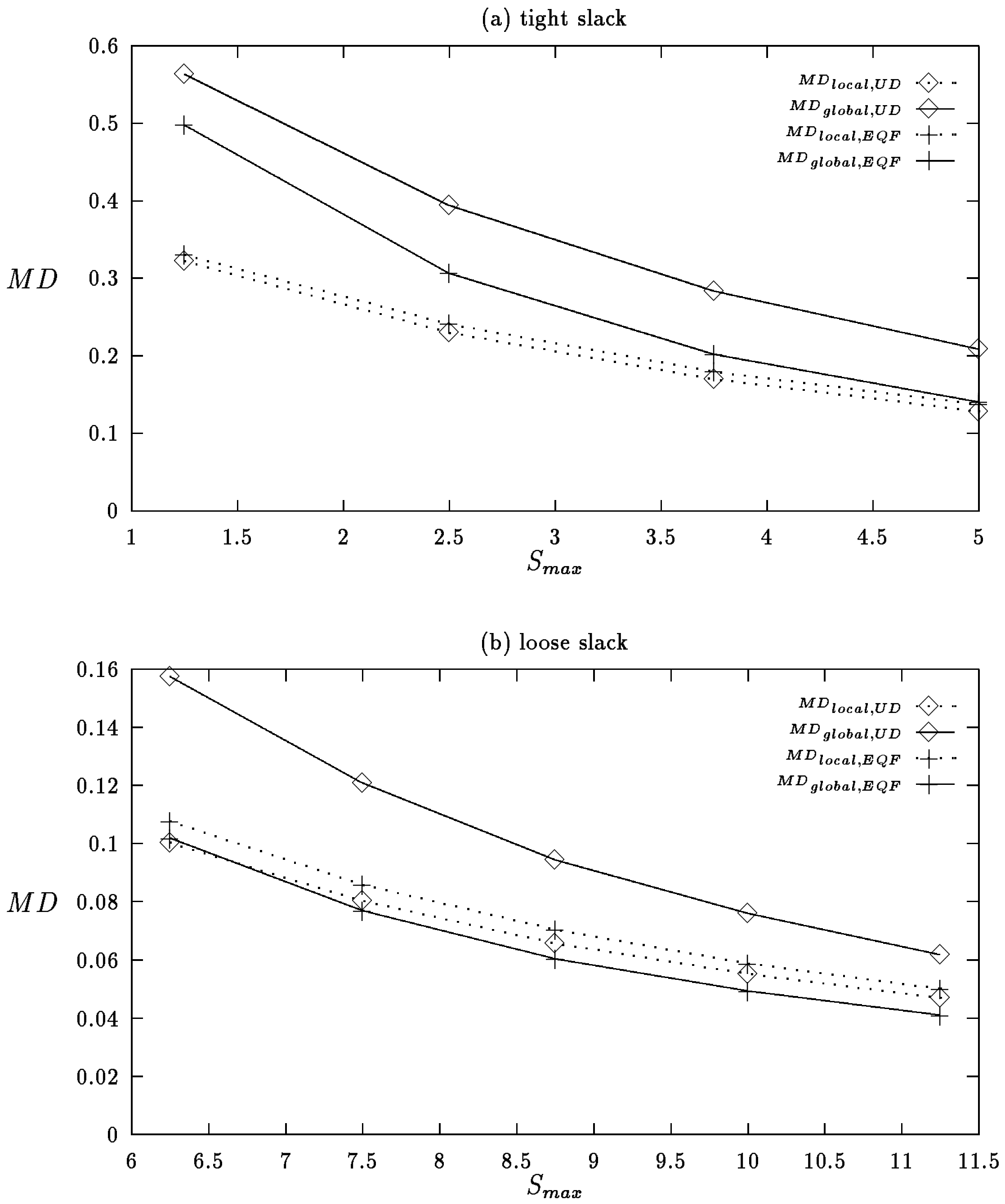


Figure 4.8: Effect of varying S_{max}.

slack, the beneficial effect is stronger and we see that global tasks actually miss fewer deadlines as compared to local tasks. The crossover point occurs when $S_{max} \simeq 5$ units.

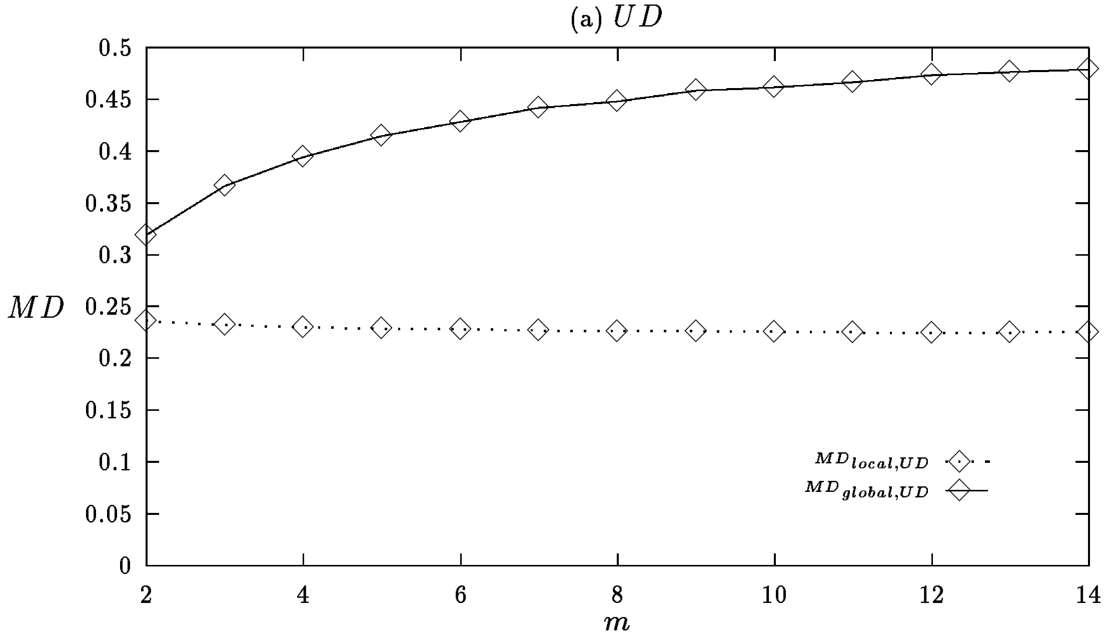
Keeping all other baseline parameters fixed, Figures 4.9(a) and 4.9(b) show the results of varying the number of subtasks, m, from 2 to 14 for UD and EQF respectively. From Figure 4.9(a), we see that MD^{UD}_{global} increases as m increases while MD^{UD}_{local} decreases. This shows that the discrimination problem is intensified by the length (number of subtasks) of global tasks. As a global task grows, UD incorrectly assigns more and more slack to early subtasks.

For EQF (Figure 4.9(b)), increasing m increases both the beneficial and the detrimental phenomena. For our parameter settings, the effects cancel each other and the result is a rather flat curve of MD^{EQF}_{global}.

To sum up, EQF always performs better than, or at least as well as, UD. The EQF gains are more significant when there is "moderate" slack and load. That is, if slack is too tight or the load too high, no matter what SSP policy we use, many deadlines will be missed. If slack is too loose or load too light, then all tasks will make their deadlines, no matter how we schedule. But in the intermediate range a smart SSP policy can make a difference and this is where EQF wins big. The EQF strategy is also superior when global tasks have many subtasks.

4.2.3 Variations of the Baseline Model

In our simulation model, several assumptions have been made. In particular, we have assumed that tardy tasks are not aborted, that the local scheduling algorithm is earliest-deadline-first, that execution time estimates are perfect, and that information



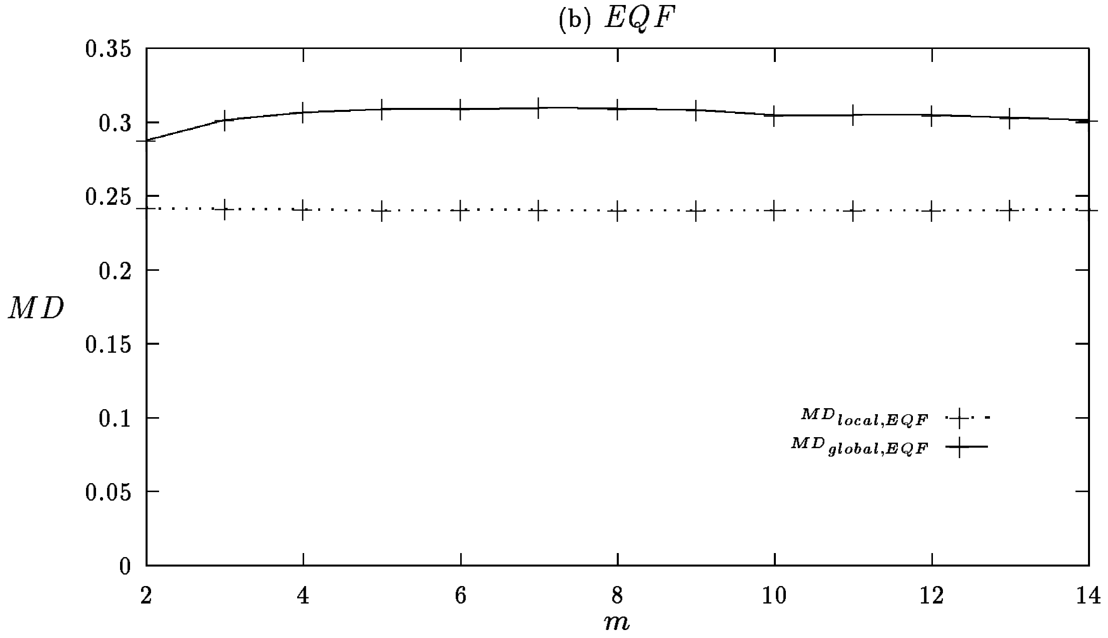


Figure 4.9: Effect of varying m.

on the number of subtasks is available. In this section we relax these assumptions and describe the performance implications. (We also experimented with other variations, that are not repeated here. For instance, we studied a scenario when global tasks can have different number of subtasks. We also studied a scenario where some of the nodes had higher local task loads than others. In all these experiments, the results do not change our basic conclusions, so the results are omitted.)

Inaccurate Information On Tasks

Among the assumptions listed above, the most crucial one is perfect prediction of task "duration". There are two components to consider — (1) task execution time, and (2) the number of subtasks of a global task. We first look at the effect of error in execution time estimates.

In order to study the sensitivity of performance to execution time estimate errors, we introduce "random noise" to the estimates. Specifically, for a task X we have: [8]

$$pex(X) = \begin{cases} ex(X) * Uniform([1 - ErrorFactor, 1 + ErrorFactor]), & ErrorFactor < 1, \\ ex(X) * Uniform([0, 1 + ErrorFactor]), & ErrorFactor \geq 1, \end{cases}$$

where $Uniform([x_1, x_2])$ returns a random number uniformly distributed in the range $[x_1, x_2]$, and $ErrorFactor$ gives the *maximum* relative error in the estimation. Notice that an estimate $pex(X)$ can be either larger or smaller than the true execution time. For $ErrorFactor > 1$, the *average* estimates are biased towards overestimating the true values.

Figure 4.10 shows MD^{EQF}_{global} and MD^{EQF}_{local} as $ErrorFactor$ changes from 0 (no error) to 2.0. Other parameter values are kept the same as our baseline experiment.

[8] Recall that $pex()$ is the *predicted* execution time, and $ex()$ is the *real* execution time.

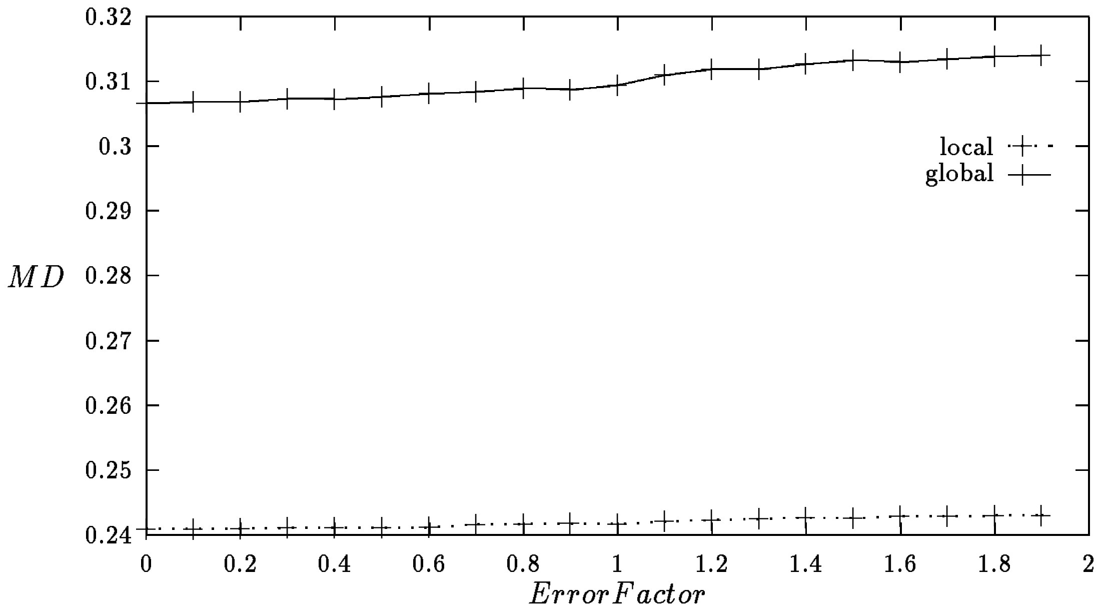


Figure 4.10: Sensitivity of EQF performance to $ErrorFactor$.

In particular, the number of subtasks of global tasks (m) is assumed known (a constant 4). Since UD does not require the knowledge on execution time, its performance is *not* affected by $ErrorFactor$ and is omitted in the graph. (The value of MD^{UD}_{global} is a constant 0.4, significantly above MD^{EQF}_{global} for the entire experiment.)

Surprisingly, the performance of EQF is not very sensitive to errors in the execution time estimate. This is illustrated by the rather flat curves shown in Figure 4.10. Even if estimates are off by up to a factor of two, EQF continues to perform well. Further experimentation shows that this sensitivity is increased by a small amount in a high load, tight slack environment, and when *minimum-laxity-first* (see below) is used as the local scheduling algorithm.

The low sensitivity to errors may appear counter-intuitive at first, but it can be explained by three observations: (1) $ErrorFactor$ gives the maximum error, but in many cases the errors are not that far off from the true values. In all these cases, EQF makes good deadline assignments. (2) In the cases where the estimates are significantly off, it is still beneficial to split up the available slack among subtasks, even if the split is rough. Incidentally, in a study of soft real-time database scheduling and concurrency control algorithms [AGM88b], it was also observed that errors in execution time estimates do not significantly affect performance. (3) Error is introduced in *both directions.* If the execution time of a global task is underestimated, the process manager will assume that there is more slack for the task than it really has. Subtasks of the global task are thus run in lower priority, with an increased probability of missing the deadline. Conversely, if the execution time is overestimated, the global task will have a higher probability of meeting its deadline. "On average", the two effects roughly cancel out. The net result is a mild increase in global task miss ratio. This phenomenon will be discussed further shortly.

Inaccurate information about global tasks can appear in another form, namely, error in estimating the number of subtasks of a global task (m). This kind of error usually occur less frequently than, and not as far off as, that of task execution time. However, it has a bigger impact on performance. For example, in an extreme case when the estimate is always one, EQF is reduced to UD.

To study the effect of an incorrect m estimate, we modify our baseline experiment such that each global task now has 15 subtasks (i.e., $m = 15$). Instead of knowing this number, each global task is associated with an estimate (Est_m). A random error ($0 < StepError < 1$) is then introduced. Specifically,

$$Est\_m = 15 * Uniform([1 - StepError, 1 + StepError]).$$

The EQF algorithm is also slightly modified. Subtasks deadlines are calculated

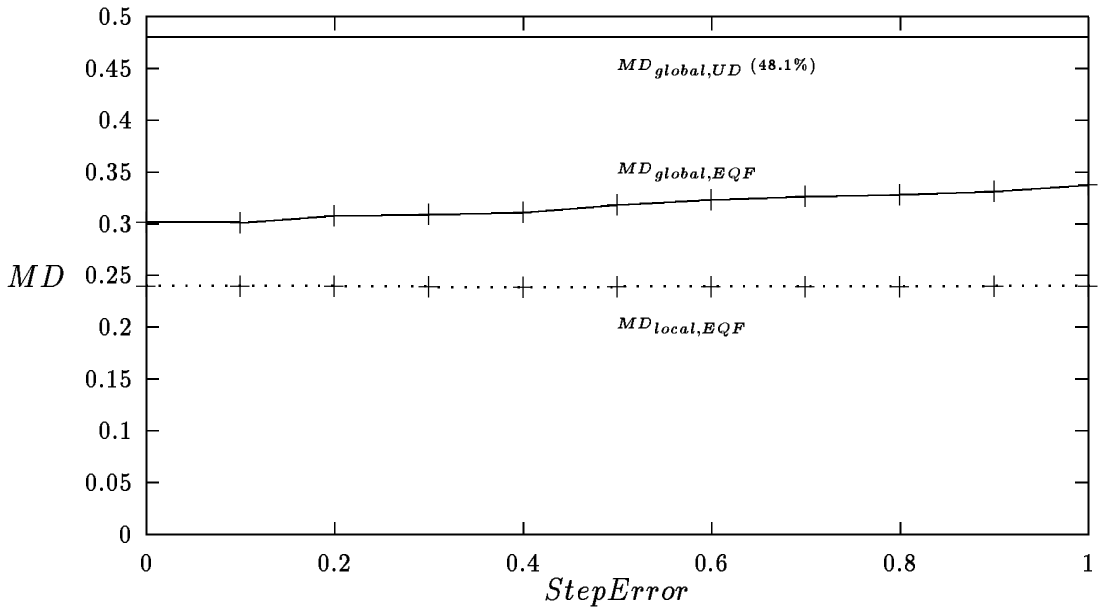


Figure 4.11: Sensitivity of EQF performance to $StepError$.

according to the following formula:

$$dl(T_i) = \begin{cases} ar(T_i) + pex(T_i) + [(dl(T) - ar(T_i) - \sum_{j=i}^{Est\_m} pex(T_j))] & \\ \qquad *[pex(T_i)/\sum_{j=i}^{Est\_m} pex(T_j)], & i \leq Est\_m, \\ dl(T), & i > Est\_m. \end{cases}$$

The difference between the above formula and the one shown on Section 4.2, page 31 is that if the current subtask is an unexpected one ($i > Est\_m$), we assume no information about the subtask. We switch to UD in that case; otherwise, the number of subtasks in the old formula (m) is replaced by the estimate (Est_m).

The result of the experiment is plotted in Figure 4.11. As a reference point, MD^{UD}_{global}, which is 48.1%, is also shown in the figure. We see that the performance of EQF is affected negatively in our experiment by an inaccurate m estimate. However, the penalty is modest. Even when $StepError$ is large, EQF still outperforms UD by a wide margin. The situation is similar to the case of inaccurate execution time estimate, and similar explanation applies.

In our previous two experiments, we assumed errors can occur in both directions. Our next experiment studies the case when errors are always in one direction. Figures 4.12(a) ((b)) shows the result of always underestimating (overestimating) the number of global subtasks. That is, $Est\_m = 15 * Uniform([1 - StepError, 1])$ in 4.12(a), and $Est\_m = 15 * Uniform([1, 1 + StepError])$ in 4.12(b). Again, underestimating m causes more slack to be assigned to the initial stages of global tasks, lowering their priority, and jeopardizing their chances of meeting deadlines. Contrarily, overestimating m brings about the opposite effect.

Tasks With Firm Deadlines

Another assumption we made in the base model is that all tardy tasks are valuable and no particular action is taken when a task misses its deadline. (See Section 3.1) . In case of firm real-time system, tardy tasks carry no value and should be aborted as soon as possible to conserve system resource. Although not reported here in detail, we modified our simulator to abort tardy tasks. The results show that abortion helps lower the percentage of missed deadlines in general (for both local and global tasks, and regardless of the SDA policy). This is because by aborting tasks that have already missed their deadlines, we effectively reduce the workload of the system. More processing power is allocated to non-tardy tasks and this results in fewer missed deadlines.

Aborting tardy tasks does not significantly change our conclusions regarding the UD and EQF policies. However, abortion does help UD (in reducing the number

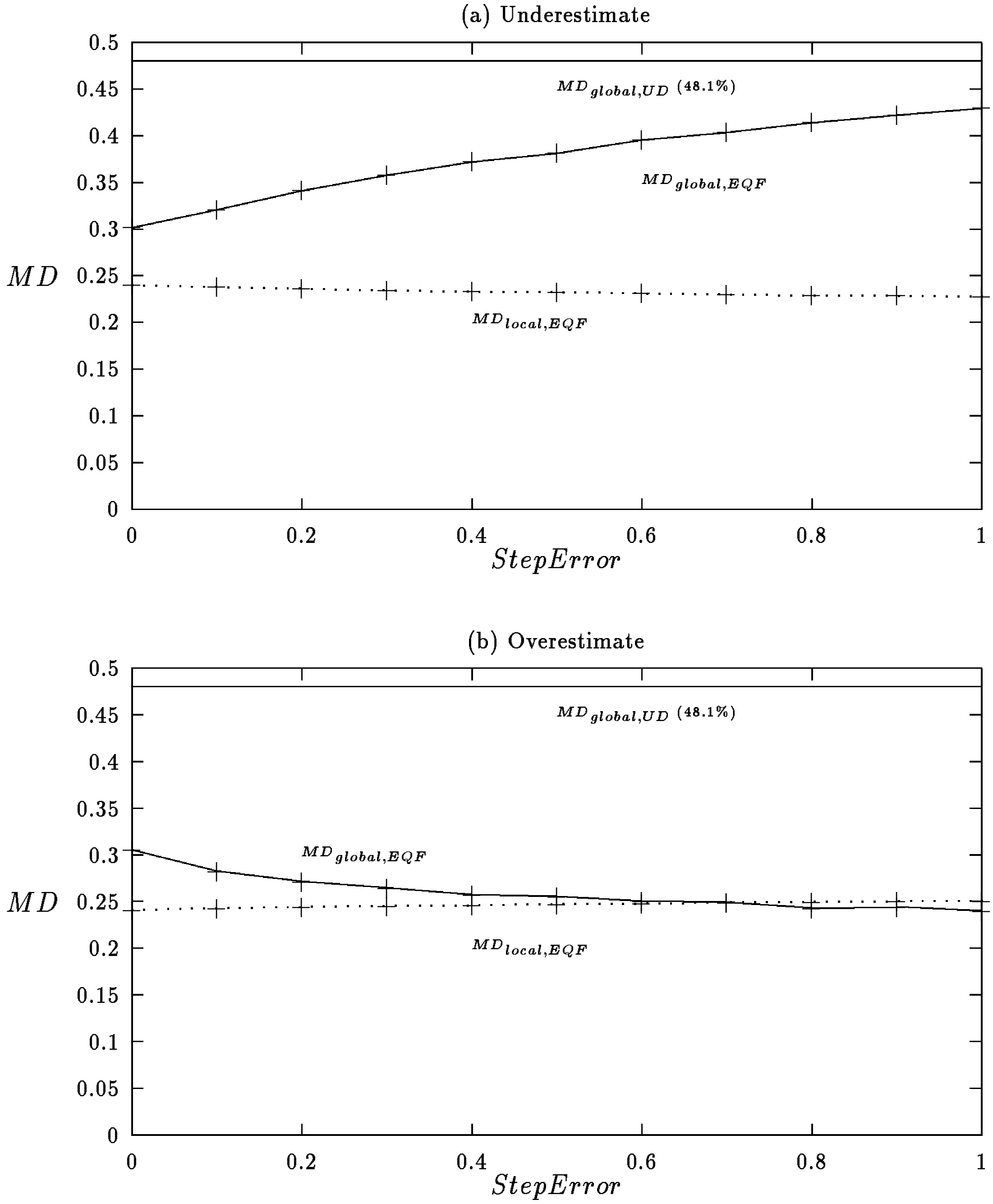


Figure 4.12: Lopsided error in estimating m.

of local and global task missed deadlines) more than it helps EQF. The reason is, for EQF, early-stage subtasks have less slack than later-stage subtasks (due to slack accumulation, see Figure 4.7). Early-stage subtasks are thus served more promptly than later-stage subtasks. A global task therefore spends relatively more time in its later stages (due to longer wait in the scheduler queues). The converse is true for UD, i.e., a global task spends more time in its early stages. Consequently, under EQF, abortion of global tasks usually happens at a later stage as compared to UD. The effective reduction of workload due to abortion is thus more significant in UD than in EQF.

Minimum-Laxity-First Schedulers

Finally, we repeated the set of experiments presented in Section 4.2.2 with the earliest-deadline-first schedulers replaced by minimum-laxity-first schedulers. The results of such experiments show similar trends and relative performance between UD and EQF. As an example, Figure 4.13 compares the performance of UD and EQF with minimum-laxity-first schedulers and the following parameter setting: No abortion, $frac\_local = 0.75$, $rel\_flex = 1.0$, $S_{min} = 0.25$, and $S_{max} = 5.0$.

From Figure 4.13, we observe similar results as for the case of earliest-deadline-first schedulers. For example, at high (low) load, global tasks miss many more (fewer) deadlines than local tasks. Also, EQF significantly reduces the gap between MD_{local} and MD_{global} when the load is high.

4.3 The Parallel Subtask Problem (PSP)

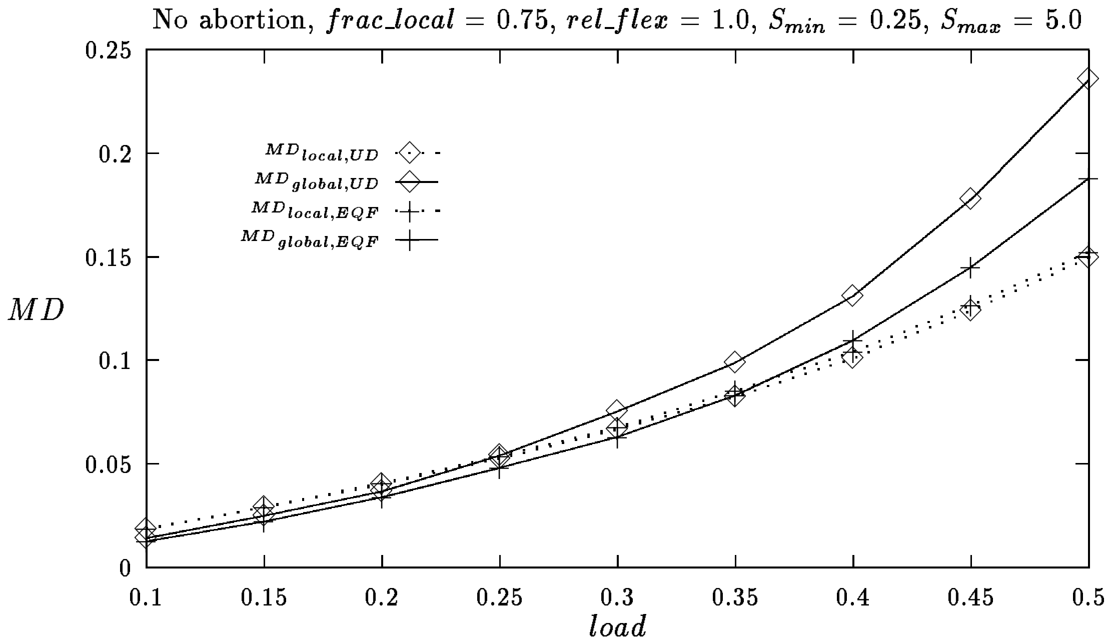


Figure 4.13: Performance comparison between UD and EQF with minimum-laxity-first schedulers.

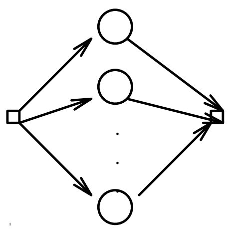


Figure 4.14: A global task that consists only of simple subtasks executing in parallel.

In this section, we discuss the second part of the SDA problem: PSP. To study PSP, we only consider global tasks of the form: $T = [T_1||T_2||...||T_m]$ where the T_i's $(1 < i < m)$ are all *simple subtask* (See Figure 4.14). For the global task T to meet its deadline $(dl(T))$, all T_i's have to be finished before $dl(T)$, their *natural* deadline.

In a soft real-time environment, when a task is submitted to a node for execution, there is no guarantee that the task will be completed before its deadline. Missing a deadline occasionally is, although undesirable, possible. There is thus a probability that a task becomes tardy due to a transient overload at its execution node. This "missed deadline" probability gets amplified in the case of global tasks with parallel subtasks. As a quick example, if an average node in the system misses 5% of the tasks' deadlines, then the probability that a global task of 6 parallel subtasks misses his would be $1-(1-0.05)^6 = 26.5\%$.[9] Comparing to a local task, or one that involves execution at only one node, a global task consumes more system "effort" to complete and yet it faces a much higher miss rate.[10] In firm real-time systems where tardy tasks are of no value and the resource invested in them wasted, the problem of missing global tasks simply because "they are big" is particularly costly. We will shortly look at several heuristics with varying degree of aggressiveness in helping global tasks to meet their deadlines.

4.3.1 Heuristics for PSP

To give global tasks a better chance of completing, we can again assign their subtasks *virtual* deadlines before they are submitted to their execution nodes.

[9] Here, we assume that all the nodes in the system have similar miss rate, tasks have similar real-time constraints, and the queueing time at the nodes are independent.

[10] In this thesis, we use the terms "miss rate" and "missed deadline probability" interchangeably.

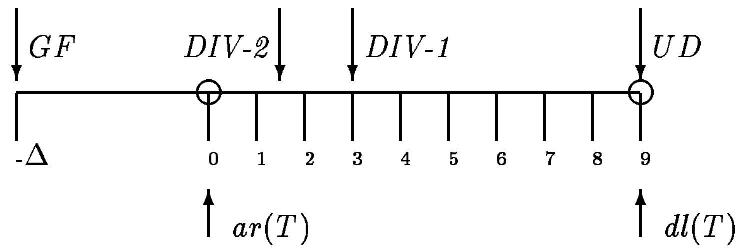


Figure 4.15: Example on the various PSP strategies applied to $T = [T_1||T_2||T_3]$.

With our example global task $T = [T_1||T_2||...||T_n]$, our goal is to set a virtual $dl(T_i)$ from $dl(T)$. Figure 4.15 illustrates the various strategies listed below. In the example, time is relative to the arrival of T. The deadline of T is time 9.

As a base strategy for comparison, we set $dl(T_i) = dl(T)$. That is, the subtasks inherit the deadline of their global task. We call this the *Ultimate Deadline* strategy (UD):

$$\textbf{UD:} \quad dl(T_i) = dl(T).$$

To make the simple subtasks of global tasks more competitive, we need to set their virtual deadlines earlier. Here, we look at a class of strategies called *DIV-x*:

$$\textbf{DIV-x:} \quad dl(T_i) = [dl(T) - ar(T)]/(n * x) + ar(T). \tag{4.1}$$

Here, x is a parameter we can adjust. The *DIV-x* strategy simply divides the amount of time that a global task has by x times its number of subtasks. The larger the value of x is, the earlier are the virtual deadlines assigned to the subtasks, and thus the higher the priority of the subtasks. As shown in Figure 4.15, *DIV-1* assigns a deadline of (9-0)/(3*1) + 0 = 3 to $dl(T_i)$, while for *DIV-2*, it is 9/6+0 = 1.5.

One issue with the *DIV-x* strategy concerns task abortion. If the local schedulers abort tasks whose (virtual) deadlines have already passed, the idea of pushing the virtual deadlines of subtasks early runs the risk of having the subtasks aborted by the schedulers. Going back to the example shown in Figure 4.15, suppose the execution time of the subtasks are all 4 time units. If *DIV-1* is used, the virtual deadlines of the subtasks will be set to time 3 (see Figure 4.15). In this case, all three subtasks will be aborted at time 3 when the scheduler figures out that their (virtual) deadlines have already been missed. In this chapter, we focus on non-aborting systems. Task abortion and its impact on the strategy performance is further discussed in Section 4.3.4.

One may notice that with the *DIV-x* strategy, the virtual deadlines assigned to the subtasks are, however big x is, later than the tasks' arrival time. A subtask therefore, may still have a lower priority than a local task if the local task has an early enough deadline. A strategy that is even more aggressive than *DIV-x* would always serve subtasks before locals. We call this strategy *Globals First* (*GF*). To implement *GF* on a pure earliest-deadline-first scheduler, we subtract a big number Δ from the deadline of a global task and take the result as the subtasks' virtual deadline:

$$\textbf{GF:} \quad dl(T_i) = dl(T) - \triangle.$$

With *GF*, the earliest-deadline-first servicing order is preserved individually within the classes of globals and locals. However, global subtasks are always scheduled before local tasks.

4.3.2 Simulation Model

We use the same simulation model as in the SSP case except that global tasks now consist of purely parallel subtasks. Specifically, a global task T consists of m subtasks $T_1, T_2, ..., T_m$ to be executed in parallel at m *different* nodes (we use the same value

m for all global tasks). The deadline of a global task is set by the following formula:

$$dl(T) = \max_i\{ex(T_i)\} + slack + ar(T). \tag{4.2}$$

where $\max_i\{ex(T_i)\}$ is the execution time of the longest subtask among the T_i's, and *slack* is the the slack chosen (from the uniform distribution) for this particular global task. We note that even though the slack of global tasks and local tasks is generated from the same slack distribution, on average, a *subtask* of a global task has more slack than a local. This is because, for a subtask T_j, we have,

$$sl(T_j) = dl(T) - ex(T_j) - ar(T) = \max_i\{ex(T_i)\} + slack + ar(T) - ex(T_j) - ar(T) \geq slack. \tag{4.3}$$

We also use the same baseline setting (see Table 4.1) except that the slack distribution is now [1.25,5.0].

4.3.3 Results

UD. Figure 4.16 shows the performance of UD in our baseline experiment. The x-axis is the normalized load to the system while the y-axis shows the fraction of missed deadlines of the various task types. The MD of three task types are shown for local tasks (dotted □), simple subtasks of global tasks (+), and global tasks (solid □) (Recall that a global task consists of 4 simple subtasks to be executed in parallel). As the load increases, the waiting time of tasks increases and more tasks (of all kind) miss their deadlines.

Comparing local tasks and simple subtasks, we notice that simple subtasks have a slightly higher chance of *meeting* their deadlines (the + line is lower than the dotted □ line). This is due to the way we generate global task deadlines. As discussed in

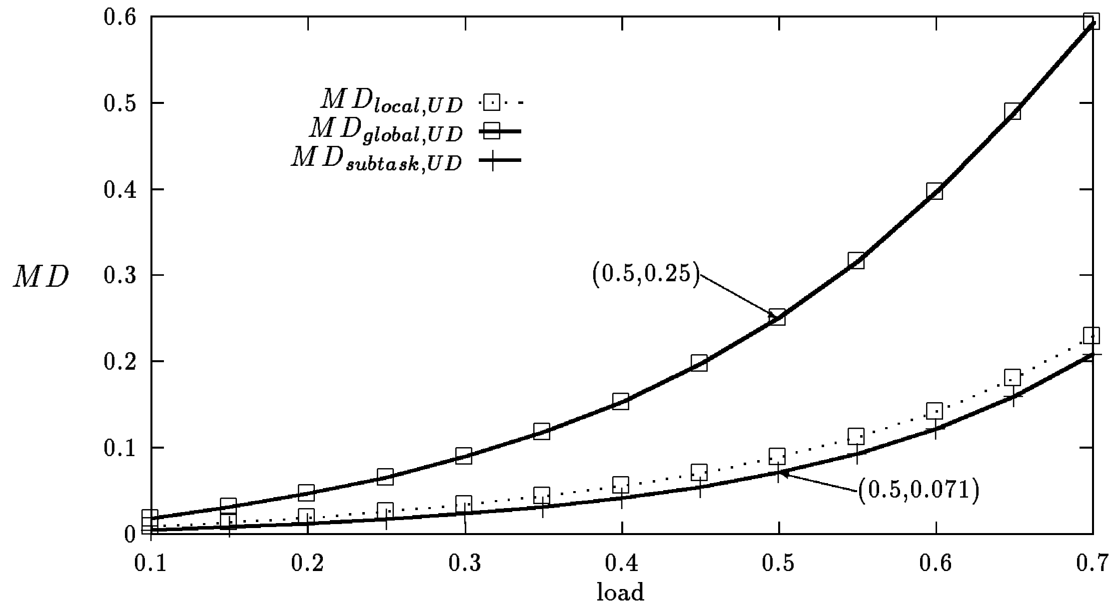


Figure 4.16: Performance of UD in baseline experiment.

Section 4.3.2, even though global tasks have the same average slack as local ones, their subtasks, on average, have slightly more slack than locals' (see Equation 4.3 on page 61). This accounts for the difference between $MD^{UD}_{subtask}$ and MD^{UD}_{local}.

For a global task to meet its deadline, all four of its subtasks must finish by the deadline. From Figure 4.16, if the load of the system is 0.5, the probability that a subtask misses its deadline is about 7.1%. If subtasks miss their deadlines independently,[11] we would expect about $1 - (1 - 7.1\%)^4 \simeq 25.5\%$ of the global tasks miss their deadlines. This number is not far from what we obtained from our experiment (25%), and is about 3 times as much as MD^{UD}_{local} (8.9%). In general, it is inadequate to assign the deadline of a global task to its subtasks and let them

[11]They don't. For example, if every global task generates subtasks for every node in the system, and there are no local tasks, then the subtask missed deadline probability will be highly co-related. We make the independent assumption here just to illustrate the problem.

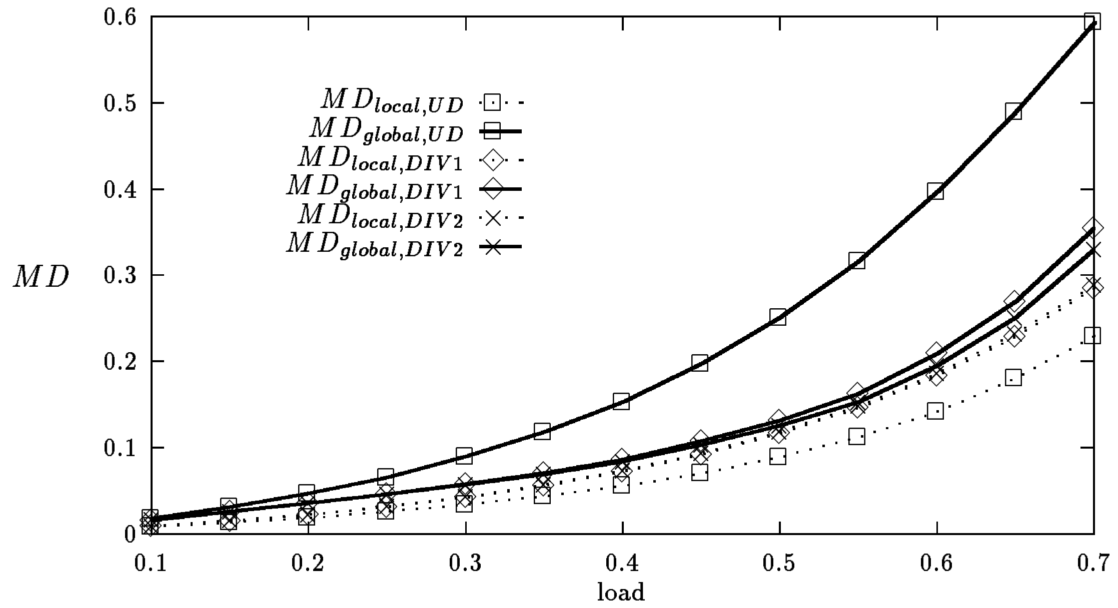


Figure 4.17: Performance of *UD* and *DIV-x* in baseline experiment.

compete fairly with local tasks.

DIV-x. From our previous discussion, we can deduce that the more subtasks a global task has, the poorer is its chance of meeting its deadline. By dividing up the amount of time that a global task is allowed to finish (see Equation 4.1), *DIV-x* effectively promotes the priority of the subtasks and thus reduces global task miss rate. One nice property of *DIV-x* is that the amount of priority promotion grows with the number of subtasks of the global task. It therefore, adjusts automatically to the need.

Figure 4.17 compares the performance of *UD* ($\Box$) and *DIV-x* for $x = 1$ ($\Diamond$) and $x = 2$ ($\times$). Let us first focus on *UD* and *DIV-1*. By giving subtasks higher average priority, *DIV-1* manages to keep the miss rate of both locals and globals at similar level (the two $\Diamond$ lines are close to each other). Since only the subtasks are given earlier virtual deadlines for a raise in their priority, local tasks suffer from this unfairness with a higher miss rate than under *UD*. However, under our baseline setting, this increment is marginal compared with the improvement achieved on global tasks. For example, at load 0.5, MD_{local} goes up from 9% to 11.7% while MD_{global} is reduced almost by half from 25% to 13%.

The relatively small increment on MD_{local} resulted from a switch from *UD* to *DIV-1* may be accounted for by the fact that only 25% of the workload is due to global tasks ($frac\_local = 0.75$). The disturbance made to local tasks by subtasks with heightened priority is therefore mild. One may argue that since the majority of the work is due to local tasks, and the miss rate of locals is increased using *DIV-1*, will we end up missing more local tasks than the number of globals saved? The answer is yes, i.e., the overall *number* of missed deadlines is actually higher for *DIV-1* than *UD*. However, if we count the work done on tardy tasks as the amount of missed work, *DIV-1* does have an advantage in this category. In fact, at a load of 0.5, our experiment shows that the fraction of missed work is reduced from $0.75 \times 0.09 + 0.25 \times 0.25 = 0.13$ to $0.75 \times 0.117 + 0.25 \times 0.13 = 0.12$. Recall that our main goal is to lower the unacceptably high global task miss ratio. In this respect, as Figure 4.17 shows, *DIV-1* is quite an effective strategy for the baseline setting. We will look at the effect of having a different job mix on the PSP strategies later.

Figure 4.17 also shows MD_{local} and MD_{global} for *DIV-2* ($\times$). By pushing the virtual deadlines of subtasks further earlier, *DIV-2* raises the priority of subtasks even higher than does *DIV-1*. The difference between their performance, however, is hardly noticeable except at very high load. Setting $x > 1$ in our baseline experiment is therefore not necessary to provide a low level of missed deadlines for global tasks. We will further discuss the question of how to set the value of x for the *DIV-x* strategy in a later section.

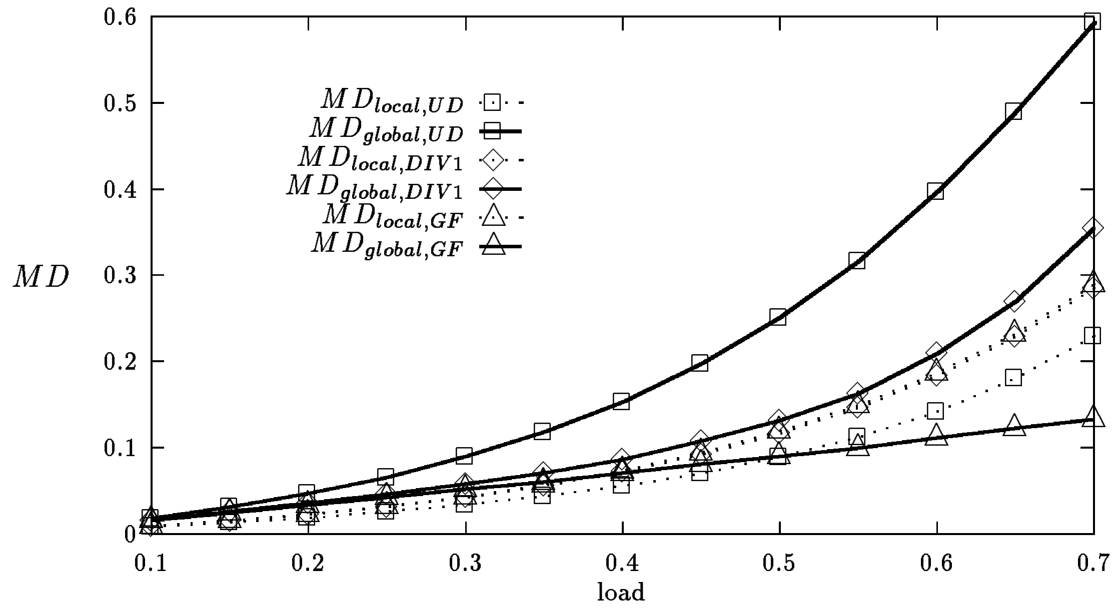


Figure 4.18: Performance of *UD*, *DIV-1*, and *GF* in baseline experiment.

GF. The minute difference between *DIV-1* and *DIV-2* as shown in Figure 4.17 may suggest that one should not look further for even more aggressive strategies. *GF*, which represents the ultimate one in raising subtask priority, may not be expected to provide any significant improvement over *DIV-x* in reducing global task miss rate. Surprisingly, our experiment shows that *GF* *does* further reduce MD_{global} by a significant amount. This is shown in Figure 4.18. Figure 4.18 is basically the same as Figure 4.17 except that we replace the curves for *DIV-2* ($\times$) by those of *GF* ($\triangle$).

Comparing *DIV-1* with *GF* in Figure 4.18, we notice that both of them miss approximately the same number of local tasks while *GF* misses significantly fewer number of global tasks than *DIV-1* does, particularly under high load situation. To understand how *GF* can meet more global task deadlines without losing on the local side, let us situate ourselves as a particular subtask, T_s, just submitted to a node,

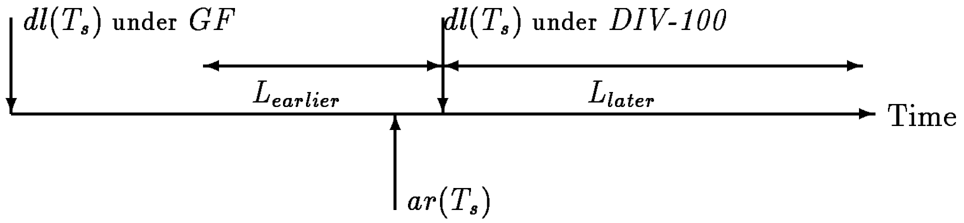


Figure 4.19: Queueing position of T_s under *DIV-100* and GF

i.e., $ar(T_s)$ = current time. We are interested in comparing the queueing position of T_s under *DIV-x* and GF strategies. From Figure 4.17, we see that the performance difference between *DIV-1* and *DIV-2* is quite small (and as will be shown later, the performance of *DIV-x* flattens out as x increases). To make the contrast more dramatic, we will consider *DIV-100* (or any x that is excessively big). Now, under *DIV-100*, T_s will see two types of local tasks in the waiting queue of the node: those with deadlines later than $dl(T_s)$ and those with earlier deadlines.[12] We call these two sets of locals L_{later} and $L_{earlier}$ respectively. Figure 4.19 illustrates the queueing position of T_s. Loosely speaking, since we use a very big x (100) in *DIV-x*, $dl(T_s)$ is set very close to $ar(T_s)$, which is also the current time. Any local tasks in the set $L_{earlier}$ would thus have deadlines that are either very close to the current time, or which have already been past. In other words, these are the tasks that are very likely to miss the deadlines. Now, by switching from *DIV-x* to GF, we are setting the priority of T_s higher than any locals. T_s is therefore cutting the line and gets itself ahead of any locals in $L_{earlier}$. We make three observations over this switch of priority:

1. Queueing time of locals in L_{later} is not affected.

2. Queueing time of locals in $L_{earlier}$ is lengthened.

3. Queueing time of T_s is shortened.

[12]Here, $dl(T_s)$ is the virtual deadline assigned to T_s.

Since the locals in $L_{earlier}$ are going to miss their deadlines no matter whether *DIV-100* or *GF* is used, prolonging their queueing time will not affect the miss rate of local tasks. On the other hand, shortening the queueing time of subtasks (such as T_s's) does significantly reduce the miss rate of global tasks. Furthermore, the higher the system load is, the more local tasks will there be in $L_{earlier}$, and the queueing time for the subtasks will be reduced further. This explains why MD_{global}^{GF} is much smaller than MD_{global}^{DIV1} especially under high load.

As a conclusion, our baseline experiment shows that the PSP problem can be corrected at the expense of losing some local tasks. Two simple strategies *DIV-x* and *GF* are shown to be effective under our baseline setting.

4.3.4 More on *DIV-x* and *GF*

Now that we have shown that *DIV-x* and *GF* are two viable solutions for PSP, in this section, we study when they are most effective. In particular, we will study the impact of the following factors on their performance: (1) The value of x for *DIV-x*. (2) The relative proportion of local tasks in the system. (3) Abortion on tardy tasks. (4) Global tasks with different number of subtasks.

Choosing x

In last section, we compared *DIV-1* and *DIV-2* and showed that the performance difference is not significant. So, is *DIV-x* sensitive to the value of x? Referring to the *DIV-x* formula:

$$dl(T_i) = [dl(T) - ar(T)]/(n * x) + ar(T),$$

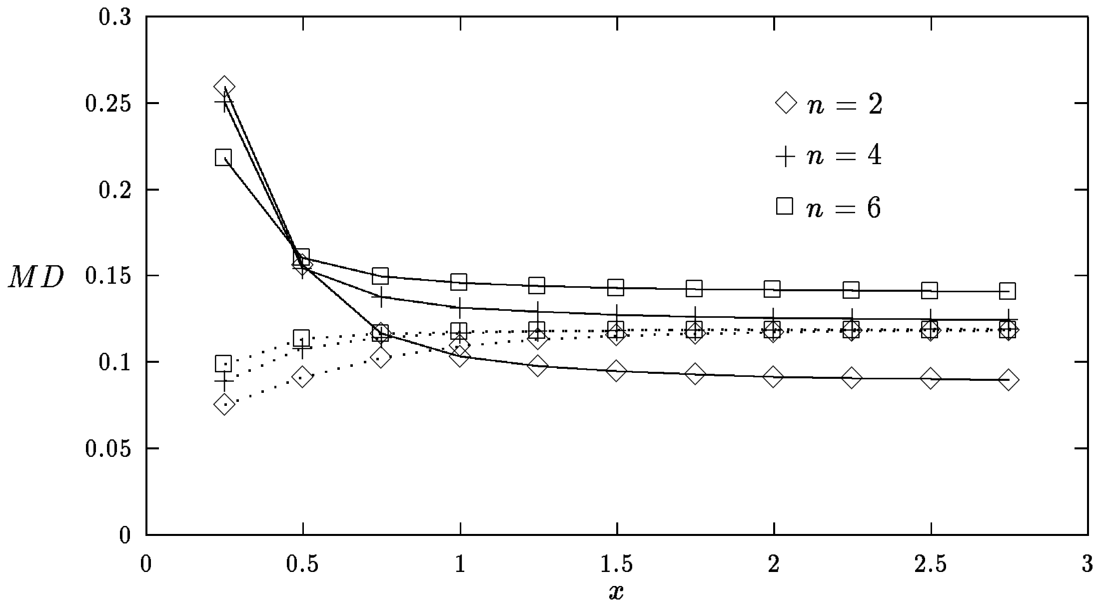


Figure 4.20: MD^{DIVx} as functions of x under different value of n. (Dotted lines for MD_{local}; Solid lines for MD_{global}.)

we see that two important factors are affecting the value of the virtual deadlines set to the subtasks: (1) the number of subtasks (n) of a global task, and (2) the value of x. The larger the product $n * x$ is, the smaller is the virtual deadline and thus the higher is the subtask priority. This, in turn, results in a lower MD_{global} and a higher MD_{local}. Now, if n is small (e.g., 2), a global task has only a small number of subtasks and PSP is not a serious problem. Therefore, we may not need a very large x or a large boost in subtask's priority to keep MD_{global} low. On the other hand, if n is large, even though PSP is serious, the $n * x$ product is already large, and again, we do not need a really big x. In fact, through extensive experiment, we find out that setting $x = 1$ is usually adequate. As an example, Figure 4.20 shows MD_{global}^{DIVx} and MD_{local}^{DIVx} as a function of x for the cases where the number of subtasks of a global task (n) is 2, 4, and 6.

From Figure 4.20, we see that all MD curves flatten out as x increases. The values of MD_{global} and MD_{local} stabilize at a smaller value of x as n becomes larger. Since 2 is the smallest number of subtasks that a global task can ever have, and the curves for $n = 2$ ($\diamond$) almost reach their stablized points when $x = 1$, setting $x = 1$ is therefore sufficient.[13]

Local Task Population

In our baseline experiment, we set $frac\_local$, the fraction of work in the system that is due to local tasks to 75%. The performance of GF and $DIV\text{-}x$ depends on this parameter. For example, if there are no local tasks in the system (i.e., $frac\_local = 0$), GF will perform exactly the same as UD because the deadlines of all subtasks are reduced by exactly the same amount (Δ) by GF. Figures 4.21(a) and 4.21(b) show MD^{DIV1} and MD^{GF} as functions of $frac\_local$ respectively. MD^{UD} is also shown for comparison.

From the figures, we see that as $frac\_local$ increases, both MD^{UD}_{local} and MD^{UD}_{global} increase slightly. This is due to the way we generate global task deadlines. Again, a global task deadline is generated as the arrival time plus a random slack plus the execution time of the *longest* subtask (see Equations 4.2 and 4.3 in Section 4.3.2). In other words, on average, the deadline of a global task is slightly larger than that of a local that is generated at the same time. Therefore, global tasks have slightly lower priority than locals and are thus slightly less competitive. As a result, as $frac\_local$ increases, more *competitive* local tasks are introduced to the system replacing some *less competitive* global tasks, and higher MD values are observed for both classes of tasks.

[13]There is no harm for setting x larger for the no-abortion case. However, if components abort subtasks whose virtual deadlines expire, setting a large x will cause many unnecessary abortions.

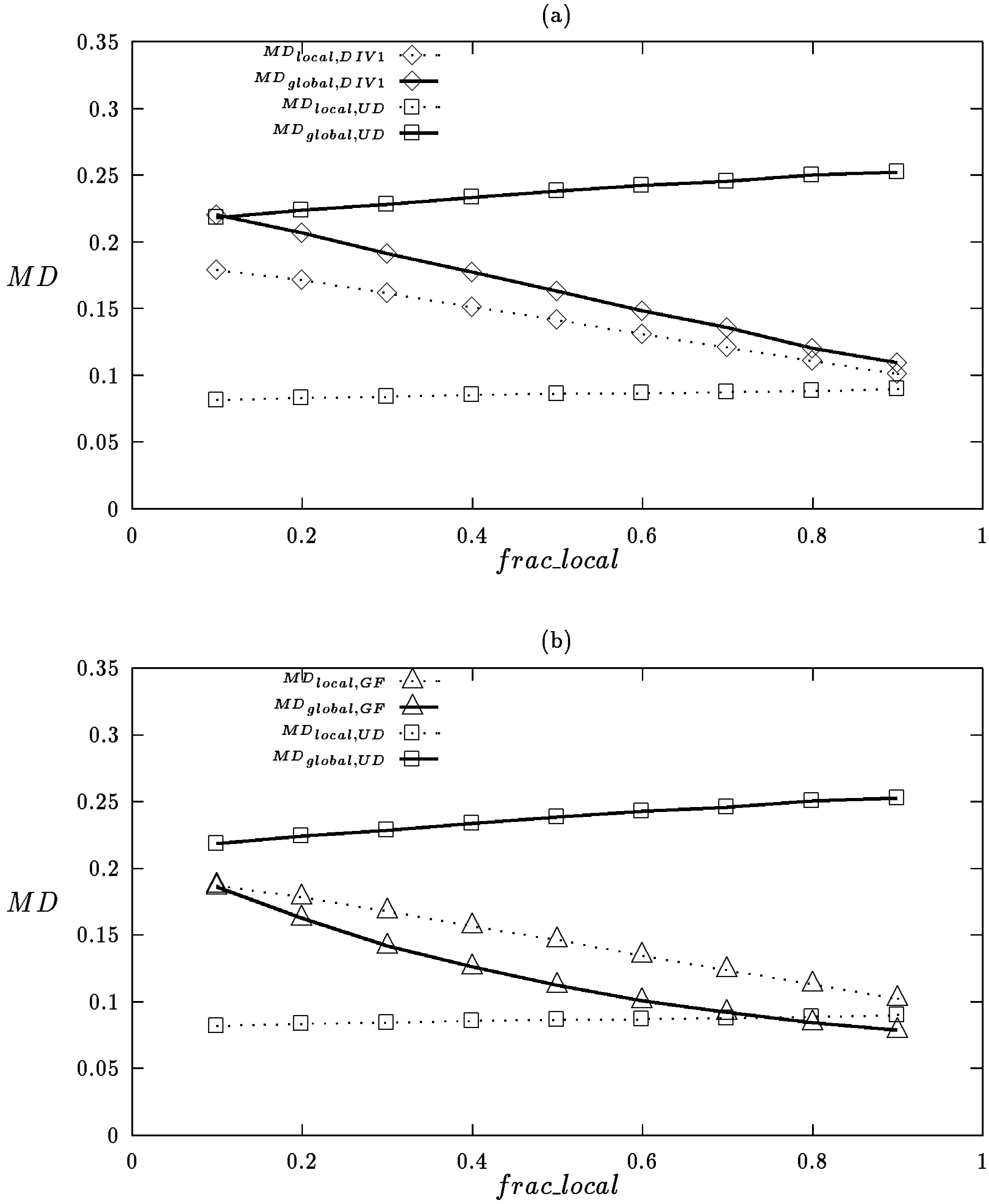


Figure 4.21: Performance of (a) *DIV-1* and (b) *GF* as functions of $frac\_local$.

On the other hand, when either *DIV-x* or *GF* is used, the subtasks are given early virtual deadlines. This raises their priority and makes them *more competitive* than local tasks. MD^{DIVx} and MD^{GF} thus drop as $frac\_local$ increases. As shown in Figure 4.21, *DIV-x* and *GF* are most effective when the system has a relatively large population of local tasks.

Abortion

We consider two different ways of aborting tardy tasks: (1) abortion by process manager and (2) abortion by local scheduler. For case (1), we assume that the local schedulers do not initiate any abortion. They will keep working on a task even though its (virtual) deadline has already expired. We implement this type of abortion using timers. When a task (global or local) is generated, an associated timer is set to run off at the task's *real* deadline. If the task is not finished before the timer expires, it is aborted[14]. Figure 4.22 shows the performance of *UD* and *DIV-1* when process manager abortion is implemented.

Comparing Figure 4.18 and Figure 4.22, we see that abortion helps reduce all miss rates by not wasting resources on tardy tasks. *DIV-1* again is very effective in keeping MD_{global} low (e.g., at $load = 0.5$, $MD^{UD}_{global} = 15.0\%$, while $MD^{DIV1}_{global} = 7.8\%$). The performance of *GF* in this case is very similar to *DIV-1* and whose curves are omitted from the graph for legibility.

For the second type of abortion, namely, abortion by local schedulers, a subtask is aborted if its virtual deadline is missed. A local scheduler thus takes an active role and aborts tasks based on the deadlines presented to it. This causes a serious problem in the implementation of *GF* and *DIV-x*. Recall that under *GF*, the virtual

[14]For a global task, all its subtasks are aborted.

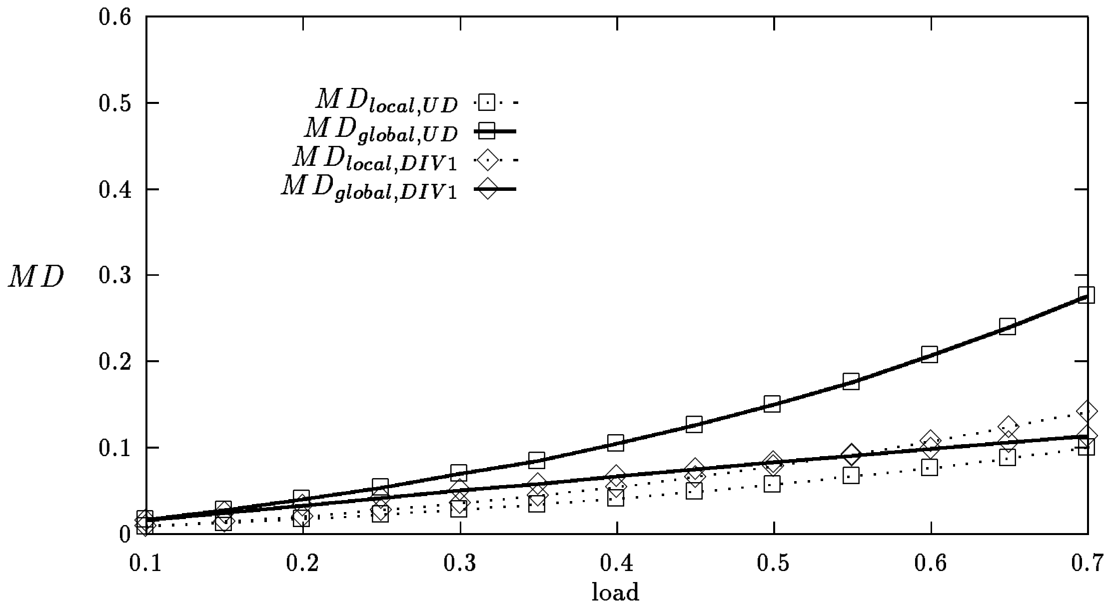


Figure 4.22: Performance of *UD* and *DIV-1* in baseline experiment with abortion.

deadline of a subtask is set by deducting a big number from its global task's deadline. The virtual deadline is thus always less than the arrival time of the subtask, and the subtask will be aborted right away. *GF* is therefore inapplicable in this situation. Similarly, *DIV-x* also pushes the virtual deadlines early. If x is set too big, the virtual deadlines will be too close to the arrival time and subtasks will be aborted before they finish even though their real deadlines are not expired.

Unnecessary abortions by local schedulers bring about two adverse effects. First, resource invested in aborted subtasks is wasted. Second, an aborted subtask has its slack consumed mostly by its former unsuccessful trial. The subtask thus has little slack left when it is resubmitted and will very likely miss its deadline. Results of experiments not shown here indicate that *DIV-x* performs poorly with local scheduler aborts. This is particularly so in a moderate to tight environment (i.e., high load,

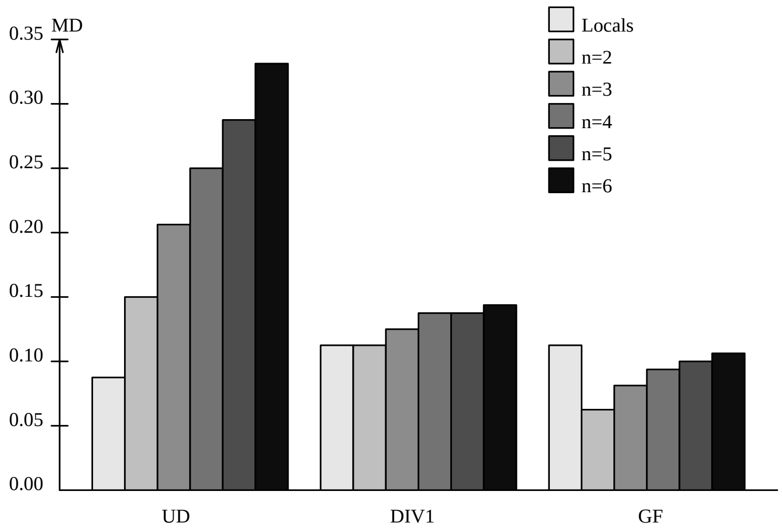


Figure 4.23: MD of different task classes under the various PSP strategies.

small slack). In this situation, special directives should be given to nodes specifying that subtasks are non-abortable locally.

Non-homogeneous Global Tasks

In our baseline experiment, we assume that all global tasks consist of exactly four parallel subtasks, and the execution time of the subtasks are all generated from the same exponential distribution. In this subsection, we generalize our base model to cover a non-homogeneous system. Due to space limitation, here we only consider systems in which global tasks may have different number of subtasks.

We modified our baseline experiment such that the number of subtasks of a global task is chosen uniformly from the range [2..6]. There are thus six classes of tasks (locals + 5 classes of globals). Figure 4.23 shows the fraction of missed deadlines of each class of task under different PSP strategies.

As expected, UD has trouble meeting the deadlines of global tasks, especially those of the *large* ones. For example, a global task with 6 subtasks has an one-third chance of missing its deadline, and which is about 4 times as likely as that of local tasks. On the other hand, at the expense of missing some more local tasks, *DIV-1* manages to keep the MD of all task classes at roughly the same level. Finally, by hoisting the priority of global tasks, GF further reduces global tasks miss rates to even lower values.

This concludes our discussion on the parallel subtask problem. We will look at the integrated solution to the subtask deadline assignment problem next.

4.4 SSP + PSP

The SSP and PSP strategies discussed in this chapter can be integrated nicely and be applied to serial-parallel tasks: A global deadline is broken down into virtual deadlines using either the SSP or the PSP strategies depending on whether the global task is serial or parallel. If a subtask is itself a complex serial-parallel task, the virtual deadline assigned to it is further decomposed. Figure 4.24 shows an algorithm which breaks down an end-to-end deadline (D) of a global task (X) into sub-deadlines for the *executable* simple subtasks. (A simple subtask is executable if all of its preceding subtasks have completed.)

To study the relative importance of the SSP and PSP strategies and how they affect complex distributed tasks, we ran an experiment with different SSP/PSP strategies applied to a specific class of serial-parallel global tasks. Table 4.2 shows the deadline assignment combinations, and Figure 4.25 shows the task graph of the global tasks generated in this experiment. Each global task has 5 serial stages. Stages 2 and

```
FUNCTION SDA(X:task; D:deadline)
BEGIN
        IF (X is a simple subtask) then
            dl(X) := D
        ELSE IF X = [X_j X_{j+1}...X_n] THEN
            assign virtual deadline to task X_j according to the SSP strategy.
            SDA(X_j, dl(X_j));
        ELSE IF X = [X_1||X_2||...||X_n] THEN
            PAR i = 1 FOR n
                  assign virtual deadline to X_i according to the PSP strategy.
                  SDA(X_i,dl(X_i));
        ENDIF
END
```

Figure 4.24: SDA algorithm for integrated SSP and PSP.

SDA	SSP	PSP
UD-UD	UD	UD
UD-DIV1	UD	DIV1
EQF-UD	EQF	UD
EQF-DIV1	EQF	DIV1

Table 4.2: Possible pairwise combination of SSP/PSP strategies.

4 are complex subtasks each with 4 parallel simple subtasks. Using our stock trading application example, the 5 stages can represent (1) initialization, (2) distributed information gathering, (3) analysis, (4) action implementation, and (5) conclusion of the trading task.

The experiment follows the same baseline setting as shown in Table 4.1 except that global tasks are replaced by those shown in Figure 4.25 and the slack of global tasks is chosen uniformly from the range [6.25,25]. This slack distribution is equal to that of local task ([1.25,5]) scaled up by 5 (because our globals have 5 stages). Figure

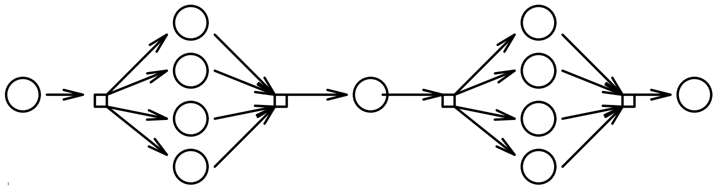


Figure 4.25: Task graph

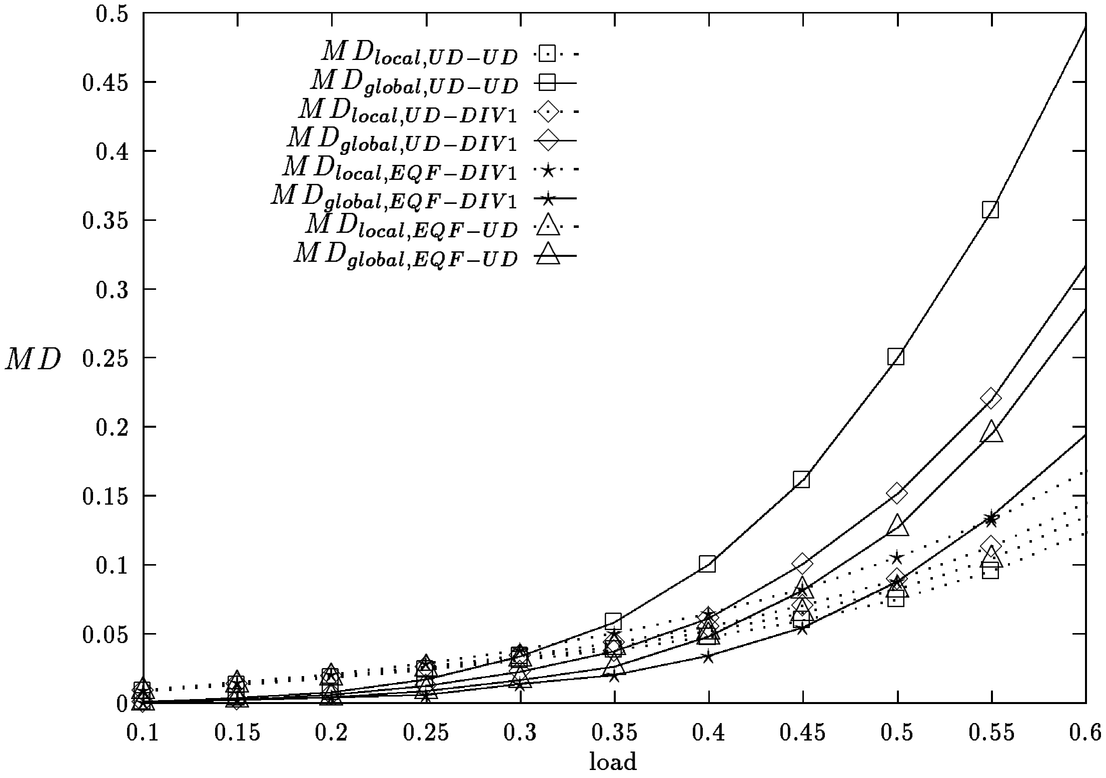


Figure 4.26: Performance of different SDA strategies.

4.26 shows the MD of locals and globals with different SDA strategies applied.

From the figure, we see that under low load, global tasks have a slightly lower miss rate. This is because of their much larger slack. However, as load increases, the *UD-UD* strategy ($\Box$) misses vastly more global deadlines than it misses local ones. The application of either EQF ($\triangle$) or *DIV-1* ($\diamond$) significantly reduces MD_{global} with a mild increment in MD_{local}. Nonetheless, they are not quite adequate if applied singularly, particularly under high load situation. The two strategies compliment each other and together ($\star$) they are able to keep MD_{global} close to MD_{local}, even up to a load of 0.6 in our experiment.

Although the results we presented in this section is limited to one particular type of serial-parallel task, they do clearly suggest that the SSP and the PSP policies can be combined, and that their benefits are "additive." As soft real-time applications get larger and more complex, our results show that a good SDA strategy becomes a very crucial part of the system design.

4.5 Conclusion

This chapter studied the problem of assigning deadlines to subtasks of complex distributed global tasks in a soft real-time environment. We showed that a single end-to-end global deadline often fails to represent the right priority of individual subtasks. This results in a very large global task miss rate. As curative measures, different algorithms for assigning virtual deadlines to subtasks were presented and evaluated. These virtual deadlines are earlier than the end-to-end global deadlines so as to speed-up the progress of global tasks. Our results showed that a good subtask deadline assignment strategy can significantly reduce global task miss rate without severely jeopardizing local tasks.

For the class of serial-parallel tasks, the SDA problem can be divided into two sub-problems: SSP and PSP. For SSP, Our performance study revealed that even though EQF significantly reduces the difference between the miss ratios of local and global tasks, global tasks still miss (in many cases) more deadlines than local ones. As we pointed out, this phenomenon is due to the "multiple stages" of global tasks which induces variation in the slack distribution. An interesting modification to EQF would control the extent of slack variability, perhaps by giving subtasks of tight global tasks *less* slack than EQF would give. One trick would be adding artificial stages. We intend to study this option in future research.

For PSP, *DIV-x* and *GF* are two effective strategies. Our study shows that they are most outstanding under high load situation and when there is a non-trivial population of local tasks in the system. Between *DIV-x* and *GF*, *GF* usually holds an edge if tardy task abortion is not supported by the system. Otherwise, *DIV-x* is a better choice because it evens up the miss rate of global tasks with different number of subtasks.

Finally, we would like to remark that the SDA problem is an important one in the design of *open systems*. An open system is usually built with existing standard components, often developed by different vendors. It is thus hard (or impossible) to orchestrate the independent schedulers, that are built into each individual component, to carry out a global scheduling policy. A good way of automatically assigning deadlines to subtasks which truly reflects the urgency of each unit of work is thus vital in an open system environment.

Chapter 5

First-Come-First-Served Schedulers

To build a soft real-time system, one may or may not be able to use specialized real-time components, such as a real-time network or a real-time database system that schedules I/O requests according to deadlines. Real-time components would obviously make it easier to meets system deadlines. Unfortunately, they are more expensive and are not readily available. For instance, standard networks such as Ethernet or token ring, and their corresponding drivers do not support real-time traffic, and they tend to deliver messages in a FCFS order. As another example, *sophisticated* disk controllers that employ algorithms such as the Elevator Algorithm [PS85] schedule disk requests according to disk block position instead of request deadlines. Conventional real-time scheduling algorithms like earliest-deadline-first do not perform well either because they cause long average seek time and poor throughput [CJL89, AGM90]. If one cannot afford the use of real-time components, how will that hamper the efficacy and performance of the real-time system? Is it crucial that every single component in the system understands deadlines? Will a non-real-time component causes a "deadline bottleneck"? If an urgent task experiences a long delay at a non-real-time component, can it catch up while visiting real-time components? In this chapter we look at how the performance of a distributed soft real-time system degrades when some of the components perform FCFS scheduling instead of EDF. We also study a technique called *server prioritization* that can improve system performance when redundant components are available.

5.1 Performance Degradation With FCFS Schedulers

To understand how tasks are affected by the presence of FCFS schedulers, we carried out an extensive set of simulation experiments based on the conceptual model in Chapter 3 and the simulation model in Section 4.2.1. Since a number of studies have been done on comparing the behavior of EDF and FCFS on a *uniprocessor* enironment, in our study, we only focus on global tasks and ignore locals. We start with serial global tasks first and look at parallel global tasks later. All simulator parameters have the same setting as in Table 4.1 except that *frac_local* is now set to 0 (i.e., no local tasks are generated).

In our first experiment, we randomly pick N_{fcfs} components to use FCFS local schedulers, while other components use EDF. Figure 5.1 shows the result of the experiment. Three curves are shown corresponding to $N_{fcfs} = 0$ (all local schedulers

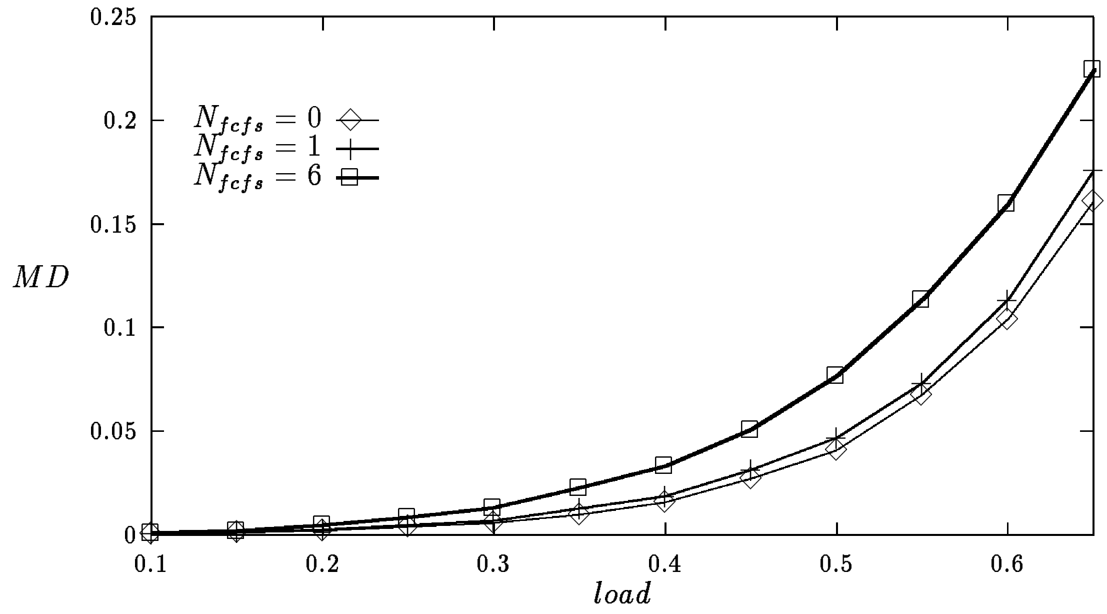


Figure 5.1: Fraction of missed deadlines versus system load for various values of N_{fcfs}.

are EDF), 1, and 6 (all local schedulers are FCFS).

From the figure we see that global task miss rate increases as the system load increases, as expected. At low load, the system has enough capacity to handle tasks so whether the local schedulers use FCFS or EDF is inconsequential. However, as the load increases and the system starts to miss more deadlines, the difference between the two schedulers becomes more significant. At high load ($load = 0.65$), a system with 6 FCFS local schedulers misses 22.4% of the deadlines — 40% more than a system with 6 EDF schedulers, which misses about 16.1% of the deadlines. We would like to remark that under normal condition, a soft real-time system should be operating with low load and few deadlines missed. However, occasionally the system will be overloaded, and it is precisely at those times when we need a good scheduler that misses the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Figure 5.1 are important.

While it is not surprising to see that an all-FCFS system performs poorly compare with an all-EDF system, it is interesting to see that switching *one* component from EDF to FCFS only causes a mild performance loss. In fact, in the baseline model, the performance of the system appears to degrade linearly with respect to the number of FCFS components. This is shown in Figure 5.2 which plots MD against N_{fcfs} for various values of system load. The fact that performance degrades linearly is somewhat counterintuitive. For instance, one would expect that with a single FCFS server, the other 5 could "make up" with their good scheduling. For example, if a task is delayed at the FCFS server, perhaps its could "make up for the lost time" when it visited the remaining EDF components. If this argument held, then the degradation for adding one FCFS server would be less than 1/6th of the total degradation when all 6 servers are FCFS. Unfortunately, Figure 5.2 shows that this argument does not hold.

The almost linear degradation of performance is due to the fact that all of the components in the baseline model handle a similar amount of load and that they work for a similar number of global tasks. To see how the system behaves under different loading condition, we modify our baseline experiment so that all the stage 1 subtasks go to component 1 exclusively while components 2 through 6 handle the rest of the subtasks. (For example, every global task in a distributed soft real-time system needs to use the network component but not all of them query the database server.) Since each global task consists of four subtasks (Table 4.1), the load is biased with component 1 handling 25% of the system load and the other 5 components handling 75% of the load (or 15% each). In this experiment, we switch the local schedulers from EDF to FCFS one by one. In one case, we start the switching from component 1; In another case, component 1 is the last one to be switched. Figure 5.3 shows the

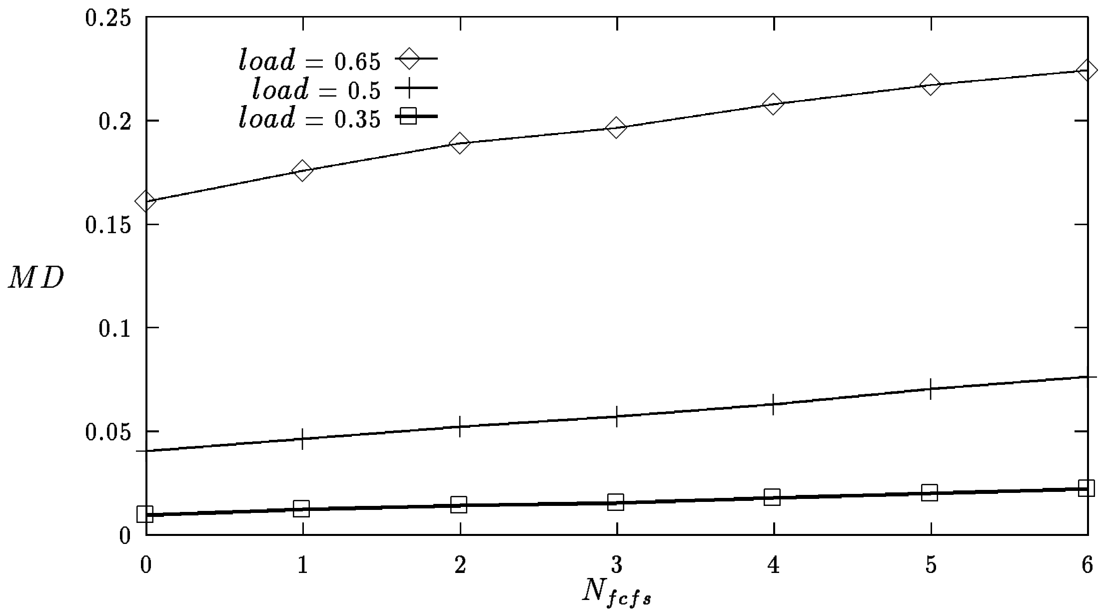


Figure 5.2: Fraction of missed deadlines versus number of FCFS local schedulers (balanced load).

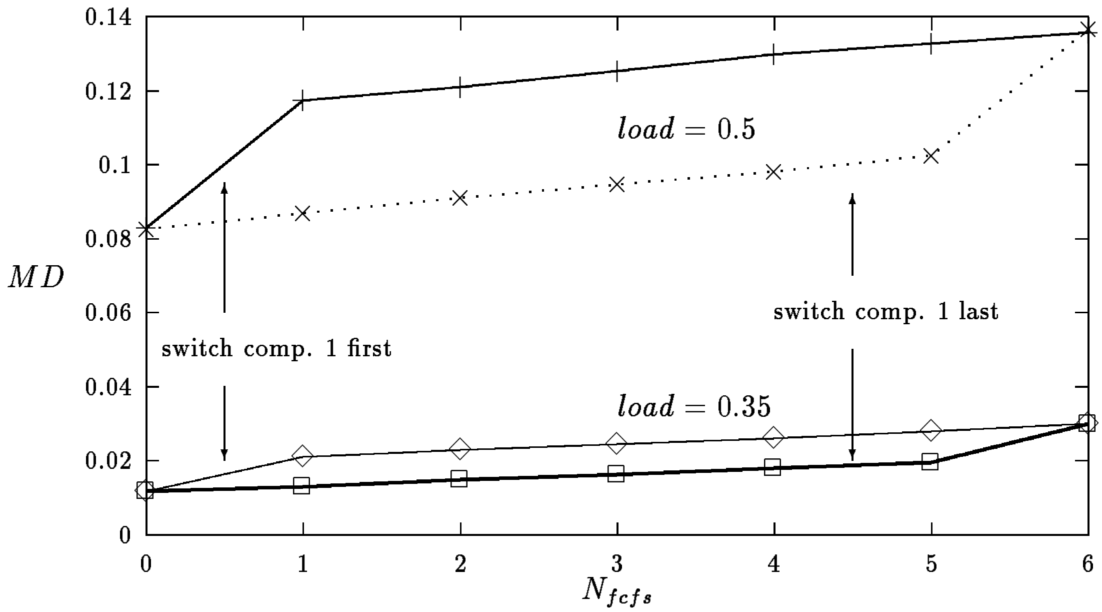


Figure 5.3: Fraction of missed deadlines versus number of FCFS local schedulers (biased load).

results of the experiment.

In Figure 5.3 two pairs of curves are shown ($\{+, \times\}$ and $\{\diamond, \Box\}$) corresponding to two levels of system load (*load* $= 0.5$ and *load* $= 0.35$). For each load, we show the MD for two cases — switching component 1 first or last. Let us focus our attention on the top most curve (line "+"). We will not discuss the other three curves since similar conclusions can be drawn from them. From the "+" line we see that switching the more heavily loaded component (component 1) from EDF to FCFS causes the miss rate to jump from 8.2% to about 11.7% (or by 3.5%) while switching the other five causes a gradual linear increase in MD to 13.5% (or by 0.36% stepwise). Even though components 2 to 6 together are responsible for 75% of the total system load, Figure 5.3 shows that it is actually *less* beneficial to run five EDF schedulers at these components than it is to run one EDF scheduler at component 1.

We have tried other system parameter settings, such as changing the slack range, directing all the *stage 4* subtasks to component 1 instead of stage 1 subtasks, etc. In all these cases we obtain similar results. This lets us conclude that with serial global tasks, the performance loss of having a FCFS component instead of an EDF one is directly related to the load of that component. When all the components in the system are loaded similarly and are used by relatively the same number of tasks, this performance loss is almost linear with respect to the the number of FCFS components. In this case, it is not absolutely critical to have every single component in the system use deadline scheduling. It may be more economical to use a couple of FCFS components if a mild performance loss is tolerable. In case the loading is biased, the designer should concentrate his effort in the more heavily loaded components. To ensure better performance, these components should use real-time schedulers or should be duplicated to increase their capacity. In some cases, the performance gain by converting a heavily used component to EDF scheduling can be more significant than converting many other less heavily used ones.

5.2 Server Prioritization

With an understanding on how FCFS schedulers affect the system's ability of meeting deadlines, we now have a better idea of which system components deserve more attention and be better equipped to handle real-time tasks. For example, a distributed system usually utilizes its network extensively. The network should therefore be appropriately upgraded with a real-time scheduler or with enough capacity to handle heavy-load situation. However, given the lack of real-time support by (low cost) standard hardware, how can we improve the timeliness of all those soft real-time applications that are running on non-real-time hardware? If we only have a single non-real-time standard hardware component, where applications have no control over the low level scheduling mechanism, there is not much that we can do. On the other hand, if redundant components are available, it may be possible to control how jobs are dispatched to replicated components to achieve real-time goals.

Fortunately, redundant components are not uncommon. Redundancy is used to provide higher reliability and scalability. For example, redundant networks are used within a local area (e.g., a Tandem Nonstop system [tan91]), and in a wide-area environment by using different carriers or lines. A stock market trader may have multiple machines analyzing the activities of a trading floor. Also, mirrored disks are used for stable storage and for crash recovery in database systems [GR93]. Data stored on both disks can also be accessed in parallel to improve performance. Special drivers can be designed to intelligently dispatch jobs across the multiple identical components. In this section we study how these redundant components can be exploited by a technique called *server prioritization* to meet system's real-time constraints.

Note that since real-time hardware is not common, its cost will be higher than that of standard hardware, where economies of scale and simpler designs lower costs. Thus, it may be more cost effective to use redundant standard hardware as opposed to a single specialized real-time one, achieving at the same time higher reliability and real-time responsiveness.

For our evaluation of how redundant components can be used, we will first consider a dual-server system as shown in Figure 5.4. (We will extend our model later and consider an n-server system in Section 5.8.) This system processes jobs with soft real-time constraints. Servers are assumed to be identical in their processing ability.

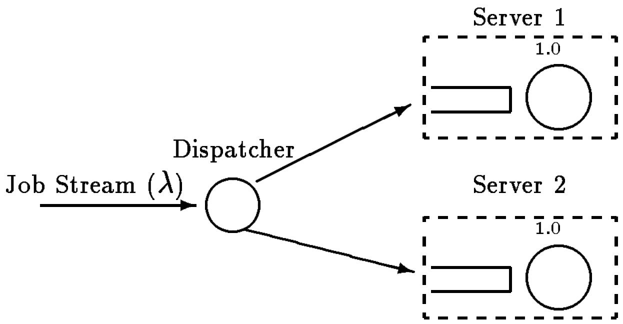


Figure 5.4: A Queueing Model

Each server has its own queue where jobs are spooled. Conceptually, the servers can be of any type (e.g. network, or disk). When a job arrives, an application entity, which we will call a *dispatcher*, will choose a server to which the job is dispatched.

We study and compare two schemes for dispatching jobs over the two servers. The first scheme is to evenly distribute jobs to the two servers. The second scheme is to prioritize jobs into two classes depending on their time criticality. Jobs with more stringent time constraints are routed to one of the servers, while the other server handles the less time-critical jobs. The server prioritization scheme is analogous to a four lane highway. The slower lanes on the outside carry more traffic (higher load); passengers on those lanes are in less of a hurry. The fast central lanes usually have fewer cars, traveling at a higher speed. In our servers, our goal will be the same as on the highway: the load on the server carrying the high priority jobs will be kept lighter so they may meet their time constraints.

To analyze the relative performance of these dispatcher algorithms, we follow two strategies: First we construct a *very simple* queuing model that lets us study the major tradeoffs involved. Second, we construct a variety of more detailed and realistic simulation models that let us verify the trends observed for the analytic model. The simulations include specifics of Ethernet and disk I/O subsystems, as well as relaxation of the simplifying assumption (e.g., periodic tasks and more than 2 servers are considered.) We stress that the analytical model is very simple. Its goal is not to predict performance of some real system; rather, its goal is to let us understand the key issues. The fact that the simulations, with very different assumptions, confirm the trends of the analysis, gives us reason to believe that the analytical model is indeed capturing the key factors.

5.3 The Analytical Model

Our analytical model is shown in Figure 5.4. We make the following assumptions:

1. Jobs are generated according to a Poisson distribution with mean interarrival time $1/\lambda$.
2. As a job is generated, it is dispatched, according to the dispatcher strategy adopted, to one of the servers.
3. Job execution times are exponentially distributed with mean equal to 1 time unit. System load is therefore normalized against server capacity.
4. Associated with each job is its slack. Here, we define slack to be equal to $t_s - t_g$ where t_s is the time at which the server must start servicing the job (to meet its deadline) and t_g is the generation time of the job.
5. Job slacks are assigned according to a slack density function $S(x)$. For illustrative purpose, we consider two slack distributions in this chapter: (1) A uniform distribution over the range $[S_{min}, S_{max}]$ (Figure 5.5(a)). That is, slacks that

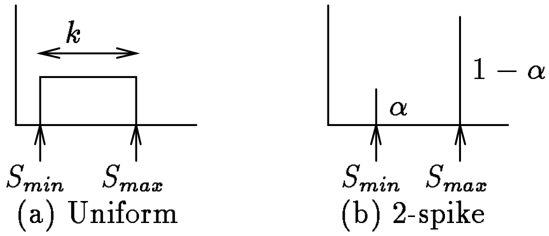


Figure 5.5: Slack Distributions.

fall in between the minimum value (S_{min}) and the maximum value (S_{max}) are equally likely to be picked. If we define $k = S_{max} - S_{min}$, then

$$S(x) = \begin{cases} 0, & x < S_{min}, \\ \frac{1}{k}, & S_{min} \le x \le S_{max}, \\ 0, & x > S_{max}. \end{cases}$$

(2) A discrete two-spike distribution (Figure 5.5(b)). In this case, a job's slack is either S_{min} or S_{max}; it takes on no other value. We will use α to represent the fraction of jobs in the system that belong to the tight-slack (S_{min}) class. The uniform distribution is suitable for modeling systems with many different real-time applications, each with different time-constraints. Systems that handle both real-time and non-real-time jobs may be modeled by the two-spike distribution.

6. The queueing discipline at each server is FCFS (M/M/1 queue).

7. Jobs are independent of each other.

As discussed in Section 5.2, this is a simple model and is used to study how different factors affect the *relative performance* of the dispatcher strategies. Real systems have their idiosyncratic properties which are not captured by our general model. For example, disk controller may employ the Elevator Algorithm in scheduling disk head movement. Similarly, networks may not process packets in a FIFO fashion and packets may be queued at each interface (as opposed to a global queue we have assumed) [1]. Again, in our simulations (Section 5.6) we model the Ethernet and the Elevator protocols (individually), and show that the results are not that different from what the FIFO model predicts. Other variations to the base model, such as earliest-deadline-first queues, and different execution time distributions, will be discussed in Section 5.8.

5.4 Analysis of a Single Server

In this section, we compute the percentage of missed deadlines for a *single server* system. That is, we assume that server 2 of Figure 5.4, is shut down and all jobs are handled by server 1, i.e., by a single $M/M/1$ queue. The expressions derived in this section will be used to study the more general case in Section 5.5.

Consider a job with slack y. The probability that it misses its deadline is equal to the probability that it must wait in the server queue more than y time units. Given an $M/M/1$ system with mean inter-arrival time $1/\lambda$ and mean service time 1.0, the waiting time distribution $T(y)$ is given by [Kle76]:

$$\begin{aligned} T(y) &= \Pr[\text{time spent in queue} \leq y] \\ &= 1 - \lambda e^{-(1-\lambda)y}. \end{aligned}$$

Thus, the probability that the job waits more than y time units and misses its deadline is $1 - T(y)$. To compute the fraction of missed deadlines MD (i.e., the expected

[1] Studies have shown that standard network protocols like Ethernet and token ring tend to queue packets in approximate FIFO order. Some access protocols (e.g. in Distributed Queue Dual Bus [NH86], and a variation of the Virtual Time CSMA protocol [ML85]) approximate a FCFS order of packet transmissions on a bus.

probability of missing a deadline), we must consider the distribution of y, i.e., $S(y)$, giving:

$$MD = \int_0^\infty S(y)[1 - T(y)]\, dy. \tag{5.1}$$

Using the uniform slack distribution of Section 5.3 (Figure 5.5(a)) and defining $MD_{1s,uniform}$ to be the fraction of missed deadlines for the single server case with the uniform slack distribution, we obtain:

$$\begin{aligned} MD_{1s,uniform} &= \int_{S_{min}}^{S_{max}} \frac{1}{k}[\lambda e^{-(1-\lambda)y}]\, dy \\ &= \frac{\lambda}{k(1-\lambda)}[e^{-(1-\lambda)S_{min}} - e^{-(1-\lambda)S_{max}}]. \end{aligned} \tag{5.2}$$

To illustrate how $MD_{1s,uniform}$ is affected by system load and job slackness, we select $S_{min} = 0.5$ time unit and plot $MD_{1s,uniform}$ as a function of λ. Three curves are shown in Figure 5.6 for $S_{max} = 10$, 30, and 100 time units[2]. We observe from the figure that the percentage of missed deadlines increases rapidly as the load increases. Moreover, when jobs are given more slack (as S_{max} increases), $MD_{1s,uniform}$ drops.

Next we consider the 2-spike slack distribution (see Section 5.3, Figure 5.5(b)). Recall that α is the fraction of jobs with slack $= S_{min}$. Substituting this distribution into Equation 5.1 we obtain:

$$\begin{aligned} MD_{1s,2-spike} &= \alpha[1 - T(S_{min})] + (1-\alpha)[1 - T(S_{max})] \\ &= \alpha\lambda e^{-(1-\lambda)S_{min}} + (1-\alpha)\lambda e^{-(1-\lambda)S_{max}}. \end{aligned} \tag{5.3}$$

To illustrate the behavior of this expression, let us take $\alpha = 0.2$ (i.e. 20% of the jobs have tight time constraint), $S_{min} = 0.5$, and $S_{max} = 19$. Figure 5.7 shows $MD_{1s,2-spike}$ (solid line) as a function of λ. (The dotted lines in Figure 5.7 will be explained later.) Note that the value of S_{max} was chosen so that the 2-spike distribution would have the same mean as the uniform distribution with $S_{max} = 30$. Thus, the average slackness

[2]Recall that the average job service time is 1 time unit.

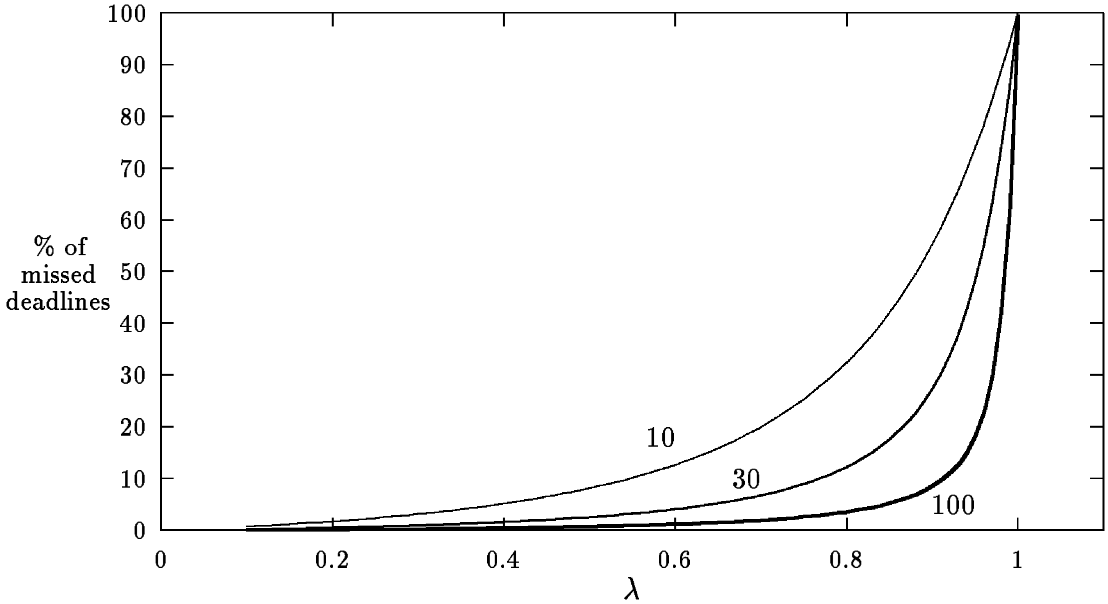


Figure 5.6: $MD_{1s,uniform}$ as a function of λ. $S_{min} = 0.5$. (Curves are labeled with corresponding S_{max} values.)

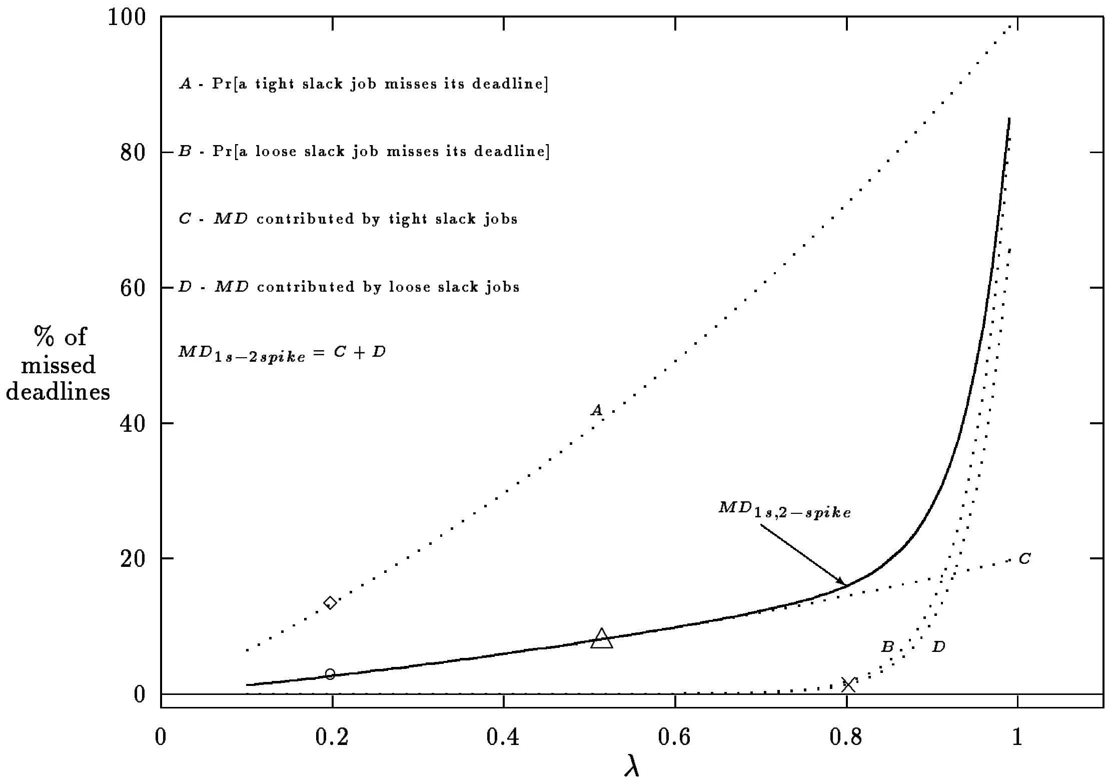


Figure 5.7: $MD_{1s,2-spike}$ as a function of λ, two job classes with slacks equal 0.5, and 19, $\alpha = 0.2$.

is the same in Figure 5.6 (curve for $S_{max} = 30$) and Figure 5.7 and it is relevant to compare. We observe that the 2-spike slack distribution gives a higher MD than the uniform distribution. For example, when $\lambda = 0.5$, $MD_{1s,2-spike} = 7.8\%$ while $MD_{1s,uniform} = 2.64\%$. Intuitively, the 2-spike distribution has relatively more jobs with very tight time constraints. These tight slack jobs contribute most to the missed deadlines. We will take a closer look at this issue shortly.

Also shown in Figure 5.7 are the components of $MD_{1s,2-spike}$ (labeled A, B, C, D). Curve A represents the probability that a tight slack job (with slack $= S_{min}$) misses its deadline, i.e., A is $1 - T(S_{min})$ (or $\lambda e^{(-(1-\lambda)S_{min})}$). Similarly, B is the probability that a loose slack job (with slack = S_{max}) misses its deadline, i.e., B is $1 - T(S_{max})$ (or $\lambda e^{(-(1-\lambda)S_{max})}$). Note that these curves are independent of α. Curves C and D represent the summands of Equation 5.3, that is, $C = \alpha A$ and $D = (1-\alpha)B$. Curve C (D) shows the portion of MD attributable to tight (loose) slack jobs, and $MD_{1s,2-spike} = C + D$.

From Figure 5.7, we observe that even though tight slack jobs represent only 20% of the population, they contribute much more to $MD_{1s,2-spike}$ than loose slack jobs do (Curve $C \gg D$), for a wide range of λ (< 0.8). Intuitively, they are the major cause of missed deadlines and hence should be given preferential treatment. This is precisely what the dispatcher attempts to do.

5.5 Dispatcher Strategies

In this section, we study a dual-server system with two possible dispatcher strategies:

Strategy 1 (Balance): Dispatch a job to either server with equal probability.

Strategy 2 (Chop): One server, S_T, is reserved for tight slack jobs. It handles jobs whose slacks are at the lower p quantiles of the slack distribution. Server S_L handles the rest of the load, i.e., jobs with slack in the upper $(1-p)$ quantile of the distribution (looser slack) (See Figure 5.8 for the uniform slack distribution case).

For the *Chop* strategy, we only consider p in the range [0,0.5]. Since the slack and execution time distributions in our baseline model are independent, if p were greater than 0.5, we would be routing more than 50% of the jobs to S_T, which is the opposite of what we want to do.[3] Instead, the goal is to have a reduced load on S_T. To see the

[3] If the slack and execution time distributions are not independent (for example when lengthier

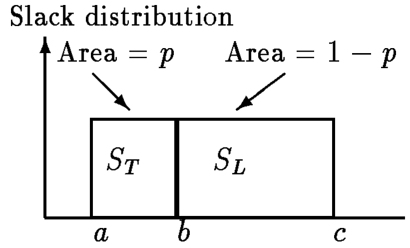


Figure 5.8: *Chop* Strategy. Tasks with slack in the range [a,b] ([b,c]) are serviced by S_T (S_L).

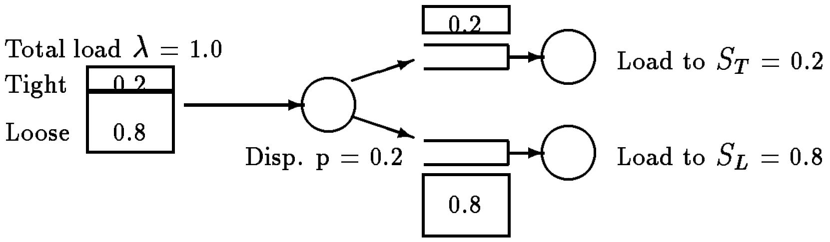


Figure 5.9: Example of *Chop* with 2-spike slack distribution.

motivation for this, say the total load is $\lambda = 1$. With the *Balance* strategy, 0.5 of this load would go to each server. If we assume a two-spike slack distribution, then the percentage of deadlines that would be missed by *each* server is given by Figure 5.7. (The slack distribution on each server is the same as $S(x)$ because jobs are randomly routed.) Thus, for a load of $\lambda = 0.5$, each server loses 7.8% of its job deadlines (point labeled $\triangle$ in the figure). The overall MD for the system would still be 7.8%, since each network has half the load.

If we now consider the *Chop* strategy with $p = 0.2$ (Figure 5.9), we see that the tightest 20% of the jobs will go to S_T. Since in our example, 20% of the jobs have slackness S_{min}, it so happens that all tight slack jobs go to S_T. The remaining 80% loose slack jobs go to S_L. The load on S_T is 0.2. The probability that a job on S_T misses its deadline is obtained from curve A in Figure 5.7. (This is the percentage of

jobs tend to have longer slack), then we may have to consider p bigger than 0.5. In any case, the main point here is to keep the load of the tight-slack server below 0.5.

missed deadlines curve for tight slack jobs given that they see a server load λ.) On this server, the probability of missing a deadline is 0.134 (point $\diamond$ on curve A), and the contribution to the total MD is $0.2 \times 0.134 = 2.7\%$ (point $\circ$ on curve C). On the other hand, on S_L, we have a load of 0.8 loose slack jobs. In this case, the contribution to MD is 1.43% (point $\times$ on curve D). Hence, the total MD is $2.7\% + 1.43\% = 4.13\%$, almost half of the MD when the $Balance$ strategy was used. Thus, by "robbing Peter to pay Paul" we have managed to make each server perform better.

We now proceed to derive expressions for the percentage of missed deadlines under each scenario. We use the notation $MD_{a,b}$ to refer to the percentage of missed deadlines under dispatcher strategy a when the overall slack distribution is b. Optional annotation will appear as superscript. The analysis for the $Balance$ dispatcher is simple since each server in the dual-server system behaves as in the single server case, except that the load is cut in half. (The Poisson arrival distribution is preserved at each server under random selection of jobs.) Hence, we can use Equations 5.2 and 5.3 on page 91, with λ replaced by $\lambda/2$:

$$MD_{Balance,uniform} = \frac{\lambda/2}{k(1-\lambda/2)}\left[e^{-(1-\lambda/2)S_{min}} - e^{-(1-\lambda/2)S_{max}}\right], \text{ and}$$

$$MD_{Balance,2-spike} = \alpha(\lambda/2)e^{-(1-\lambda/2)S_{min}} + (1-\alpha)(\lambda/2)e^{-(1-\lambda/2)S_{max}}.$$

For the $Chop$ strategy, we consider each slack distribution in turn. For the uniform slack distribution, we note that the load on S_T is Poisson with parameter $p\lambda$ and its slack distribution is uniformly distributed in the range $[S_{min}, S_{min} + kp]$. (Recall that $k = S_{max} - S_{min}$.) By Equation 5.2, the fraction of missed deadlines for S_T is

$$MD_{Chop,uniform}(S_T) = \frac{\lambda}{k(1-\lambda p)}\left[e^{-(1-\lambda p)S_{min}} - e^{-(1-\lambda p)(S_{min}+kp)}\right].$$

By similar argument,

$$MD_{Chop,uniform}(S_L) = \frac{\lambda}{k[1-(1-p)\lambda]}\left[e^{-[1-(1-p)\lambda](S_{min}+kp)} - e^{-[1-(1-p)\lambda]S_{max}}\right].$$

Finally,

$$MD_{Chop,uniform} = p \cdot MD_{Chop,uniform}(S_T) + (1-p) \cdot MD_{Chop,uniform}(S_L). \quad (5.4)$$

To analyze the two spike distribution, we consider two subcases:

Case 1 : ($\alpha \leq p$). In this case, the fraction of system load that is handled by S_T is more than the fraction of jobs that have tight slack. S_T thus has a mixture of jobs. It has a total load of $p\lambda$. A fraction α/p of this load consists of tight slack jobs. Loose slack jobs represent a fraction of $(p-\alpha)/p$ of the S_T load. Thus, by Equation 5.3 we have:

$$\begin{aligned} MD_{Chop,2-spike}(S_T) &= (\alpha/p)[p\lambda e^{-(1-\lambda p)S_{min}}] + \frac{p-\alpha}{p}\lambda p e^{-(1-\lambda p)S_{max}} \\ &= \alpha\lambda e^{-(1-\lambda p)S_{min}} + (p-\alpha)\lambda e^{-(1-\lambda p)S_{max}}. \end{aligned}$$

Server S_L is responsible for loose slack jobs only. Its load is $(1-p)\lambda$. Therefore,

$$MD_{Chop,2-spike}(S_L) = (1-p)\lambda e^{-[1-(1-p)\lambda]S_{max}}.$$

Case 2 : ($\alpha > p$). In this case S_T has exclusively tight slack jobs with a load $p\lambda$. Server S_L receives a load of $(1-p)\lambda$, out of which a fraction of $(\alpha-p)/(1-p)$ is tight slack and $(1-\alpha)/(1-p)$ is loose slack. Again, by Equation 5.3, we have,

$$MD_{Chop,2-spike}(S_T) = p\lambda e^{-(1-\lambda p)S_{min}}.$$

$$MD_{Chop,2-spike}(S_L) = (\alpha-p)\lambda e^{-[1-(1-p)\lambda]S_{min}} + (1-\alpha)\lambda e^{-[1-(1-p)\lambda]S_{max}}.$$

In either case 1 or 2,

$$MD_{Chop,2-spike} = p \cdot MD_{Chop,2-spike}(S_T) + (1-p) \cdot MD_{Chop,2-spike}(S_L).$$

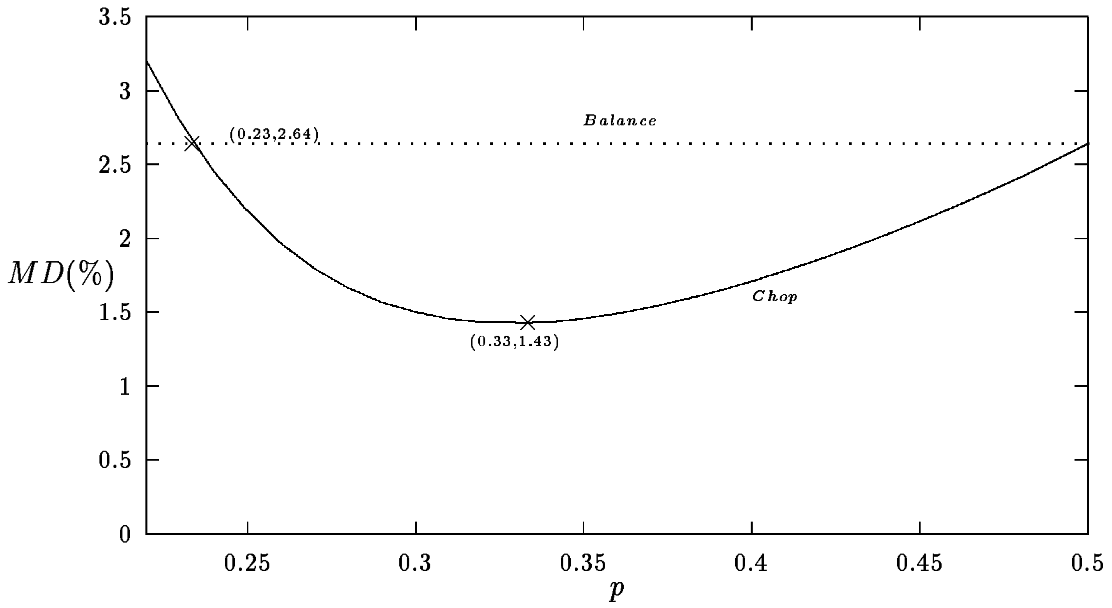


Figure 5.10: $MD_{Chop,uniform}$ as a function of p. $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 30$. Numbers in parentheses give the co-ordinates of marked point.

To study the expressions we have derived, we select a set of representative base parameter settings. We are not attempting to model any particular system; rather we select a base setting that illustrates the main tradeoffs involved. For the base setting we then vary one or two parameters at a time, showing their impact on performance. Here, we have selected to present the results we consider the most illustrative.

For our base setting we choose $\lambda = 1.0$,[4] the uniform slack distribution with $S_{min} = 0.5$, $S_{max} = 30$, and the two spike distribution with $S_{min} = 0.5$, $S_{max} = 19$, and $\alpha = 0.2$. Note that the two slack distributions have similar mean value.

Figure 5.10 shows $MD_{Chop,uniform}$ as a function of p. Also shown is $MD_{Balance,uniform}$

[4]Since we have two servers, the highest load for a stable dual-server system is 2.0.

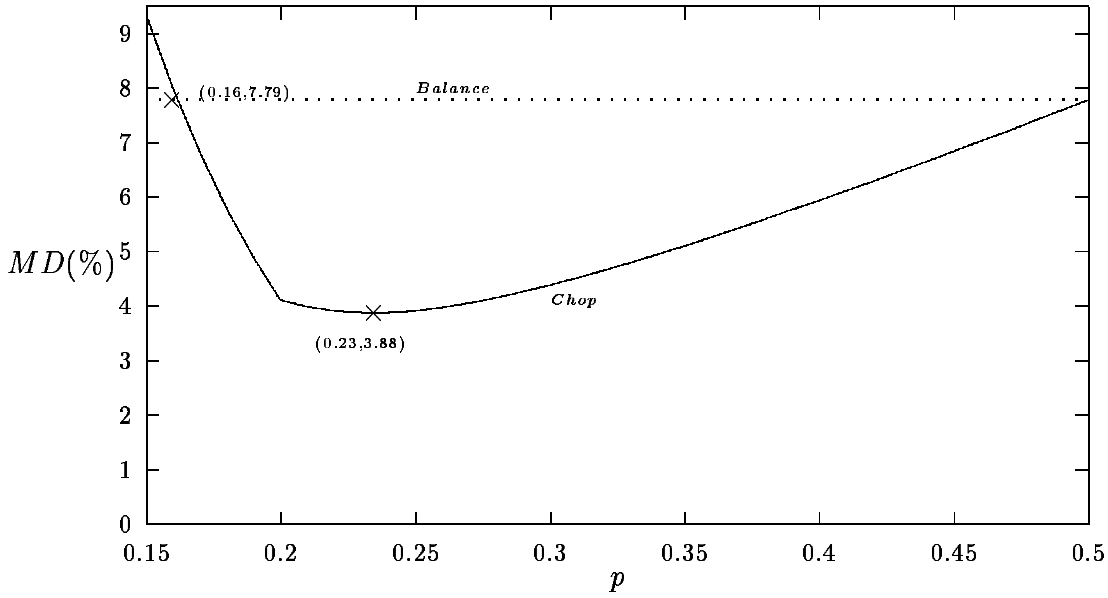


Figure 5.11: $MD_{Chop,2-spike}$ as a function of p. $\alpha = 0.2$, $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 19$.

(dotted line), which is a constant 2.64%. We see that the performance of *Chop* is sensitive to the choice of p. If p is chosen too small (< 0.23), server S_L is congested with a load (> 0.77) that it cannot properly handle. Strategy *Chop* is counterproductive in this case. However, over a wide range of p values, [0.23,0.5] *Chop* does outperform *Balance*. (Recall that p is always less than 0.5.) Finally, we note that the optimal value of $MD_{chop,uniform}$, $MD^{opt}_{chop,uniform} = 1.43\%$ occurs at $p = 0.33$. This is about 46% fewer missed deadlines than strategy *Balance*, quite a significant improvement.

Figure 5.11 shows a graph similar to Figure 5.10 but for the 2-spike slack distribution. We see in this case that *Chop* performs better than *Balance* over a wider range of p [0.16,0.5] and its optimal point (3.88%) is about 50% better than $MD_{Balance}$ (7.79%).

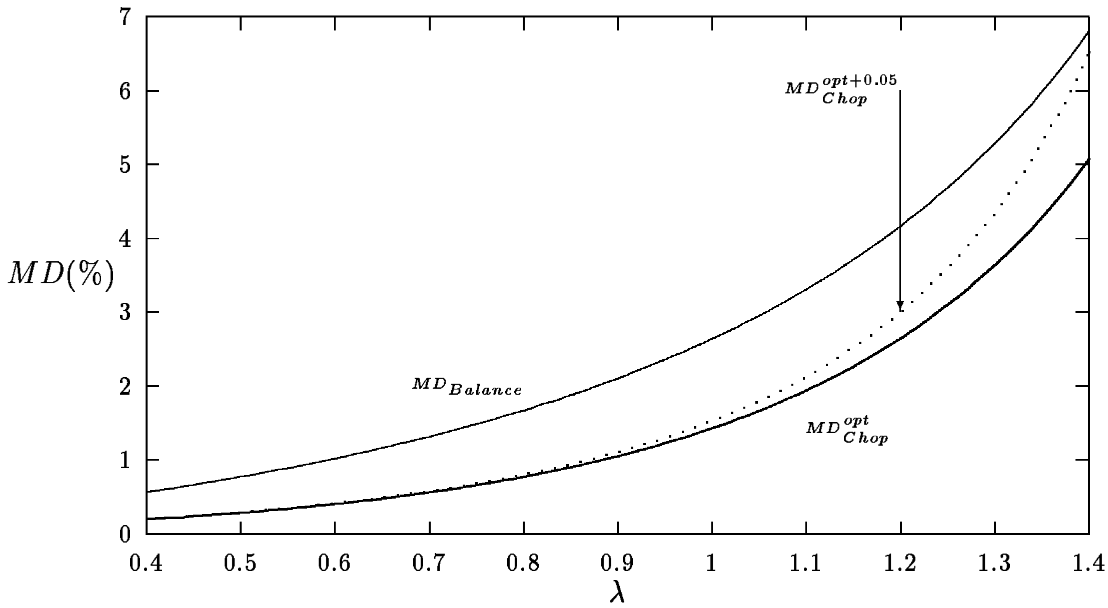


Figure 5.12: $MD_{Balance,uniform}$, $MD_{Chop,uniform}$ as λ varies.

To study the impact of load on performance, we vary the load λ from 0.4 to 1.4, under the uniform slack distribution ($S_{min} = 0.5$, $S_{max} = 30$). (Results for the two spike distribution are similar and are not shown here.) Figure 5.12 shows $MD_{Balance,uniform}$ and $MD^{opt}_{Chop,uniform}$ as λ varies. (For a given λ, $MD^{opt}_{Chop,uniform}$ is obtained by selecting the p that minimizes the percentage of missed deadlines.) We see that when λ is small, $MD^{opt}_{Chop,uniform}$ increases at a slower rate than $MD_{Balance,uniform}$. It then slowly catches up as λ becomes larger. (The dotted curve $MD^{opt+0.05}_{Chop,uniform}$ is discussed below.)

Figure 5.13 shows the relative gain in switching from *Balance* to *Chop* by presenting the ratio $MD_{Balance,uniform}/MD^{opt}_{Chop,uniform}$. We see that *Chop* performs much better than *Balance* under low load. As a matter of fact, one would expect the load to be quite low in any system with real-time deadlines, else too many deadlines would

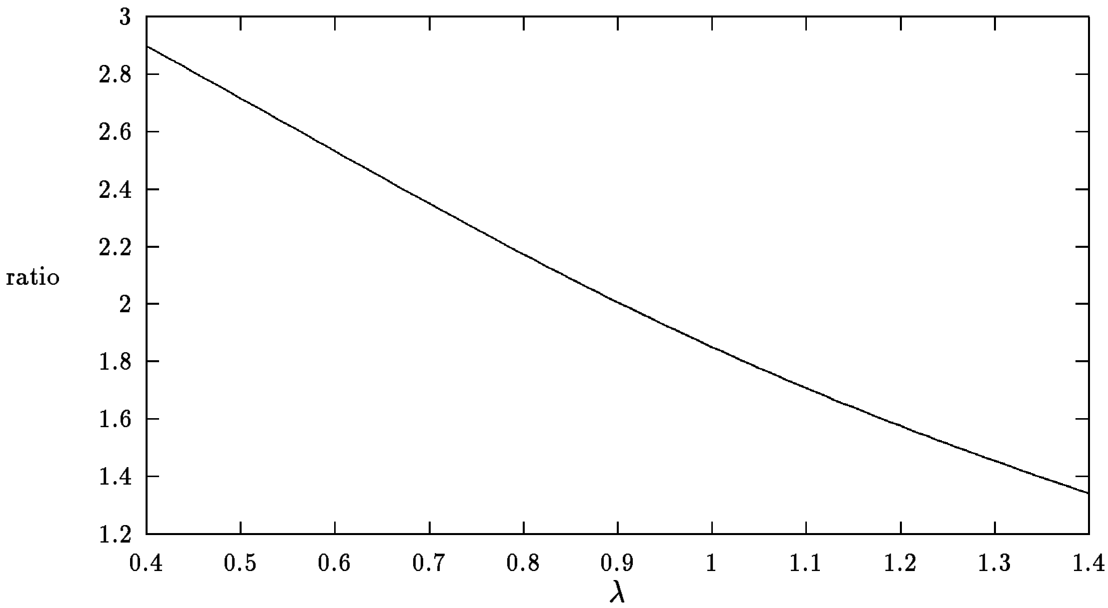


Figure 5.13: $MD_{Balance,uniform}/MD_{Chop,uniform}$ as λ varies.

be missed. Looking at Figure 5.12, we would expect λ to be say less than 0.8, so that very few deadlines are missed. And it is precisely in this range that *Chop* performs the best. For instance, at $\lambda = 0.4$ *Balance* misses nearly thrice as many deadlines at *Chop*! So clearly, the *Chop* strategy is paying off.

As pointed out earlier, MD_{Chop} is sensitive to the choice of p. But how bad is it if we choose the wrong p? In Figure 5.12, we show $MD_{Chop,uniform}$ evaluated at a suboptimal $p = p_{opt} + 0.05$, where p_{opt} is the optimal p value. Since $p < 0.5$, the selected p is at least 10% off its optimal value. The resulting curve is labeled $MD_{Chop}^{opt+0.05}$ in Figure 5.12. We observe that under low load, the percentage of missed deadlines hardly changes as a result of the variation in p. As the load increases beyond 1.2, the variation in p is significant, and *Chop* begins to perform as poorly as *Balance*. Again, we note that a real-time system should have considerable spare capacity, so that the vast majority of deadlines can be met. Therefore, under normal condition, the system should not be operating at a high load. This suggests that inaccuracy in finding an optimal p should not diminish the advantages of $Chop$. (We return to the selection of p in Section 5.7.)

Another way to study the tradeoff between load and deadlines is to assume the system can tolerate a maximum number of missed deadlines, and to control the load. For instance, consider a voice/data communication system that can tolerate missing the deadlines of β (< 1.0) voice packets and still provide a satisfactory service. We want to know, for a given value of β, what the maximum value of λ can be, provided that $MD \leq \beta$. To do this, we will use the 2-spike distribution and set S_{max} to ∞, which means that data packets do not have deadlines at all. Figure 5.14 shows the maximum load the system can handle for both $Balance$ and $Chop$ strategies, as β ranges from 0.5% to 3%. (We assume $\alpha = 0.20$ and p is optimal.) We see that $Chop$ can accommodate a load that is about 2.5 times of $Balance$'s over the range of β values shown.

We next consider the impact of slackness on performance. While keeping λ fixed at 1.0 and S_{min} at 0.5, we vary S_{max} from 10 to 80. Figures 5.15 and 5.16 show $MD_{Balance,uniform}$, $MD^{opt}_{Chop,uniform}$, and their ratio as S_{max} varies. As can be seen in the figures, the gain by using $Chop$ over $Balance$ can be very significant. It is also interesting to note that as the slack range becomes larger, $Chop$ outperforms $Balance$ by a wider margin ($Chop$ misses a factor of three fewer deadlines at high slacks). The explanation is that if the slack range is too small, then no matter how one divides the load, the servers will find themselves serving jobs of very similar slack. If jobs are of more or less the same slack then $Balance$ is just a special (and optimal) case of $Chop$. Therefore, for small value of S_{max}, the performance of $Balance$ approaches the optimal performance of $Chop$.

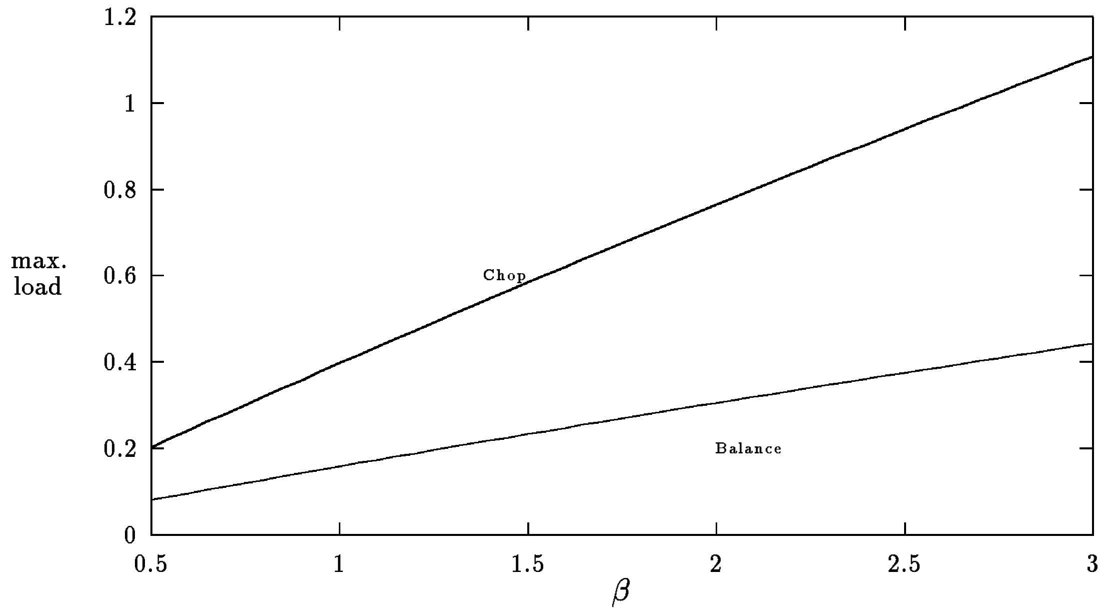


Figure 5.14: Maximum Load for *Balance* and *Chop* as β varies.

In conclusion, our simple analytical model indicates that *Chop* outperforms *Balance*, as long as a good p value can be selected. The gain can be very significant if either the system load is low or there is a wide variation in job slackness.

5.6 Simulation

In the previous sections, we presented an analytical evaluation on two dispatcher strategies for a simple dual-server system. Several simplifying assumptions (e.g. FCFS server queues) were made for tractability of the analysis. Many real systems are likely to deviate from our simple model. In order to see how real systems would perform, we implemented two detailed event-driven simulators: One for a dual-Ethernet

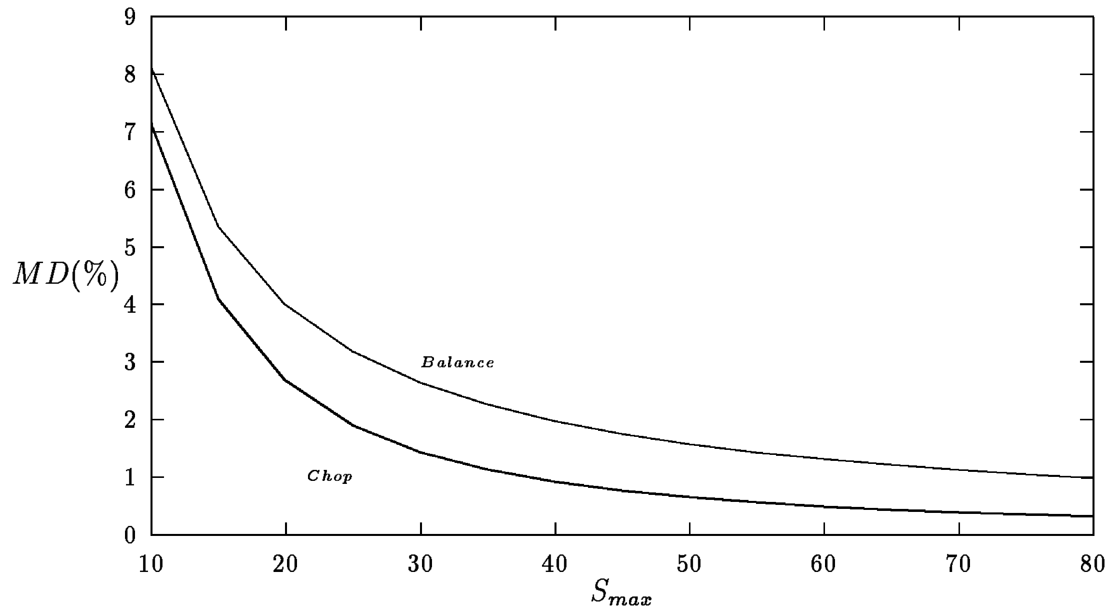


Figure 5.15: $MD_{Balance,uniform}$, $MD^{opt}_{Chop,uniform}$ as S_{max} varies.

system (Section 5.6.1) and another for a dual-disk system (Section 5.6.2). We present our simulation results in this section.

The reasons for choosing Ethernet and disk for our simulation studies are that they are commonly found in computer systems; they have interesting job servicing orders (CSMA-CD with binary exponential backoff for Ethernet and the Elevator algorithm for disk); and most importantly they differ in their response to load stress. The Ethernet protocol, which is based on contention resolution, does not handle high load situation well[5] [Tan88]. The Elevator algorithm, on the other hand, accommodates high load better by picking up disk requests that are "on the way" as the disk head moves. In fact, the average seek distance decreases as the load increases. Since *Chop* has the effect of stressing one server while relaxing the other, it is thus interesting to

[5]CSMA-CD protocols thrash under high load due to the large amount of bandwidth wasted in contention resolution.

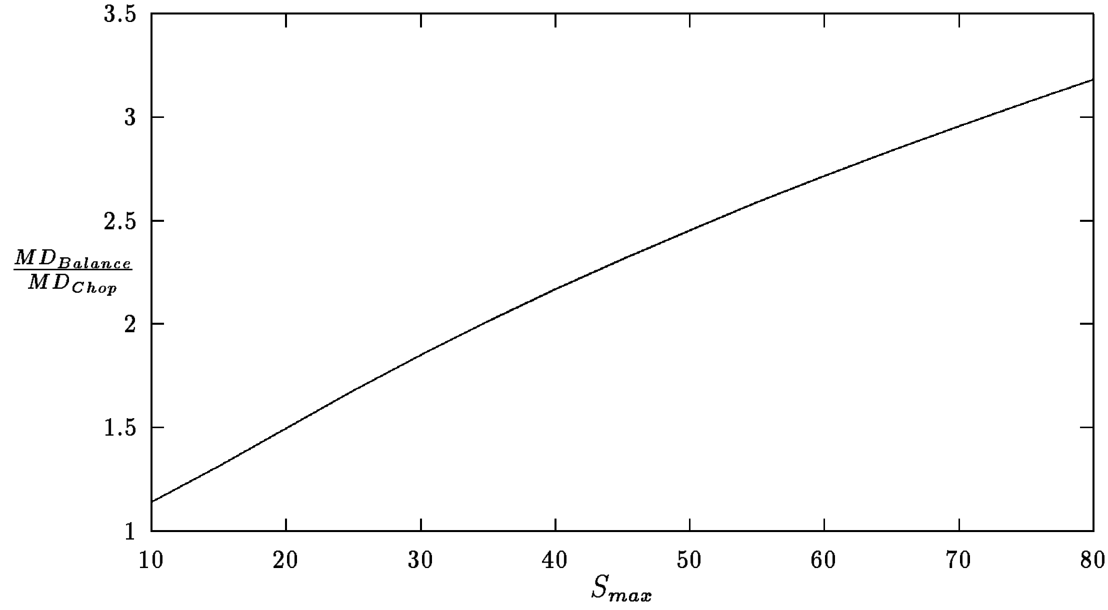


Figure 5.16: $MD_{Balance,uniform}/MD^{opt}_{Chop,uniform}$ as S_{max} varies.

see how this strategy fares when applied to dual-Ethernet and dual-disk systems.

5.6.1 Ethernet Simulation

We simulate a dual-Ethernet system with n nodes (or stations) (See Figure 5.17). Each node generates a Poisson packet stream with mean rate λ/n (total load thus equals λ). Two buffer queues at each node, one for each network, are used to internally store packets waiting for service. The queueing discipline is non-preemptive earliest deadline first (EDF). 1-persistent CSMA/CD with binary exponential backoff is implemented as the network access protocol [MB76]. The transmission time is still assumed to be exponentially distributed with mean equal to 1 time unit. Finally, the round-trip latency of the network (τ) is taken to be 4% of the mean packet

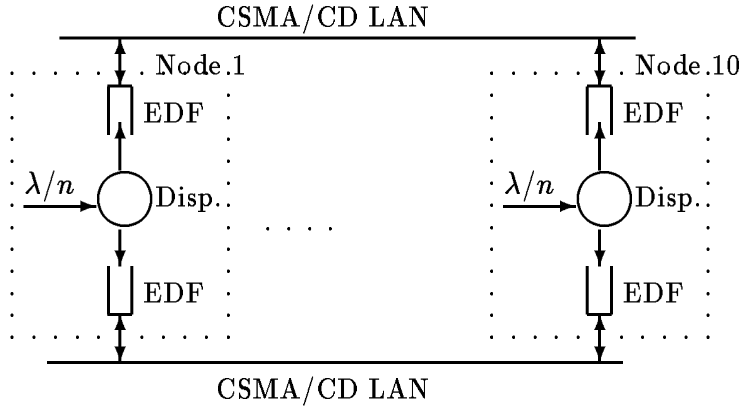


Figure 5.17: Ethernet Simulation Model. (EDF – Earliest Deadline First Queue)

n	λ	S_{min}	S_{max}	τ
10	1.0	0.5	30.0	0.04

Table 5.1: Baseline setting for Ethernet simulation.

transmission time[6] (i.e., 0.04 time unit).

Each simulation run lasts 200,000 time units. For example, with a packet arrival rate of $\lambda = 1.0$, this means about 200,000 packets are processed per run. Each data point is obtained by averaging the results of 3 simulation runs, each with a different random number seed. The 95% confidence interval is $\pm$ 0.13 percentage point. Table 5.1 shows the setting of our Ethernet baseline experiment.

The solid lines in Figure 5.18 show the simulation results comparing the *Balance* and *Chop* strategies as p varies for the baseline setting. The curves in Figure 5.10 (page 98) are reproduced in Figure 5.18 as dotted lines. These dotted lines show the results as predicted by the analytical model.

Comparing the simulation results (solid lines) and the analytical results (dotted

[6]Based on a 512 bit contention slot time and a 1.5kB packet size (plus overhead) [Tan88].

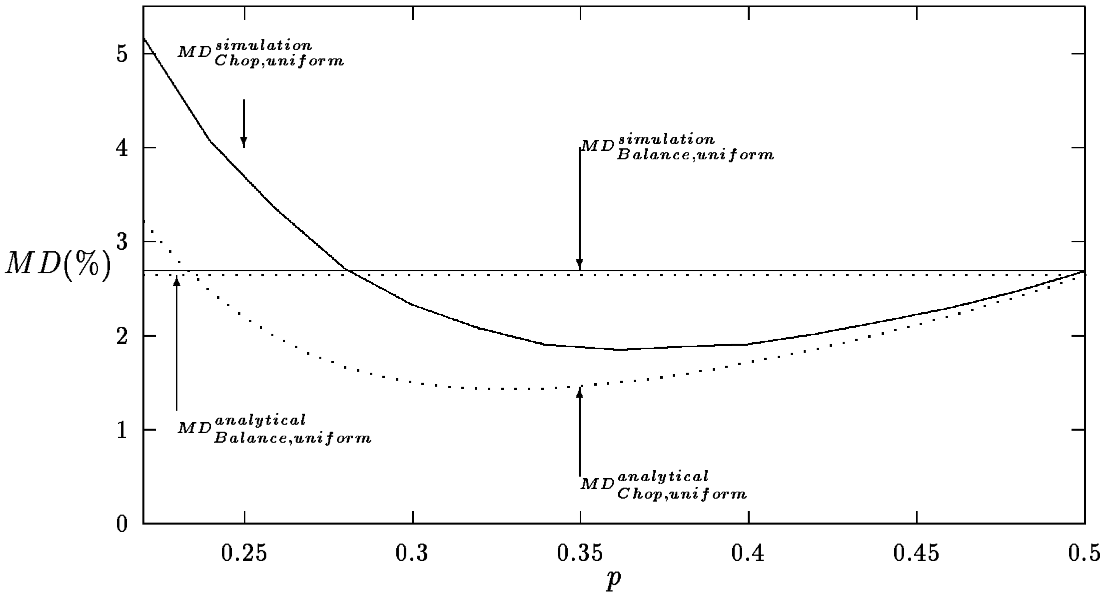


Figure 5.18: $MD_{Chop,uniform}$ as a function of p. $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 30$, $n = 10$. Ethernet Simulation.

lines), we observe that the analytical model underestimates the missed deadlines MD when Ethernets (Figure 5.17) rather than FCFS servers with global queues (Figure 5.4) are assumed. We also note that the underestimation decreases as p approaches 0.5.

This underestimation is attributable to the additional delay introduced by network contention. Time is wasted in packet collisions. Furthermore, a packet may be delayed because of multiple conflicts, due to the probabilistic nature of the access protocol. This effect translates into a higher MD value than predicted. Moreover, when p is small (e.g. 0.25), the low priority network is given a high load (e.g. 0.75). The degree of contention in the low priority network is thus high, making the difference between the analytical MD value and the simulation MD value larger.

Notwithstanding the discrepancy, Figure 5.18 shows that both the simulation and analytical results display similar *trends*. In particular, (a) $Chop$ performs better than $Balance$ over a significant range of p values, (b) the slope of MD_{Chop} is relatively flat at the optimal point, so that a minor deviation from the optimal value can be tolerated, and (c) the gain by using $Chop$ over $Balance$ is still significant ($Chop$ misses 31% fewer deadlines than $Balance$ does).

Among the other experiments we have done with our Ethernet simulator, we have looked at the effect of changing the number of stations (while keeping the system load constant) and the round-trip latency of the network. Our results show that increasing the number of stations increases the probability of network contention, and thus more overhead in resolution. For example, if we have 10 packets and only one (sending) station, all the packets are queued up in the station and no collision occurs. On the other hand, if we have 10 stations, in the worst case, each station has one packet to send, and severe contention ensues. However, we observe that the

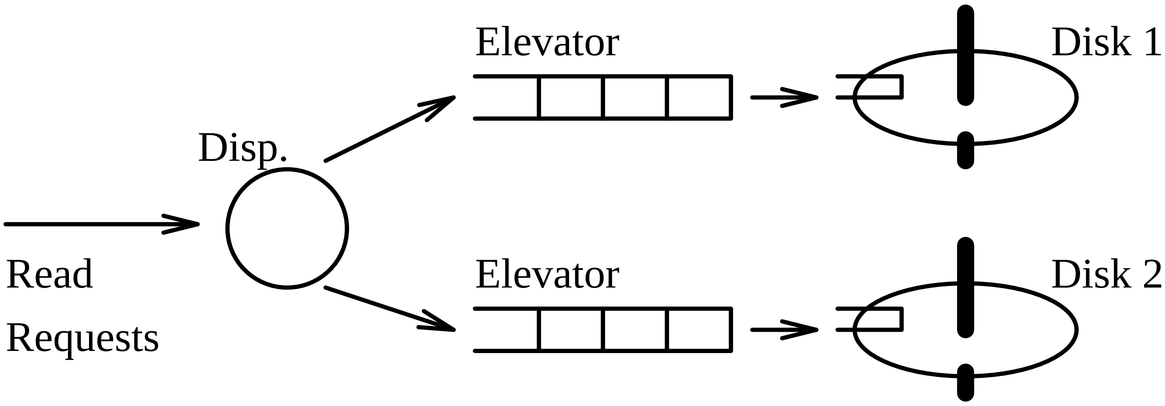


Figure 5.19: Disk Simulation Model.

increment in collision probability due to each additional station becomes quiescent once the number of stations reaches five. Thus, in our baseline setting of $n = 10$ stations, results are not very sensitive to n.

Another set of experiments concerns the round-trip latency of the network. The general observation from this sensitivity test is that the smaller the round-trip latency (τ) is, the better *Chop* performs compared with *Balance*. For example, when $\tau = 0.05$, *Chop* misses 28% fewer deadlines than *Balance*. When τ is reduced to 0.04, the improvement climbs to 31%. This can be explained by the fact that a smaller round-trip latency means smaller contention slots, and thus the contention resolution overhead is reduced. *Chop* can then free up the high-priority network exclusively for packets with stringent time constraints without causing much overhead in the low-priority network, which is loaded with all the loose-slack packets.

5.6.2 Disk Simulation

Our next experiment simulates a dual-disk system (see Figure 5.19). We make the following assumptions concerning the physical disks and the I/O requests.

First of all, we assume that the disks mirror each other and they are equally capable in satisfying I/O requests. Disk requests are of two types: reads and writes. In this experiment we only consider read requests, and therefore each request can be serviced by either one of the two disks. Write requests are ignored for two reasons: First, since the disks are mirror to each other, a write has to be processed by both disks. The net effect (if writes were considered) is a lowered disk processing capacity, and this affects both *Balance* and *Chop* the same way. Second, if I/O requests are generated by real-time transactions then read requests are performed *before* the transaction is committed while writes are performed *after* the commitment.[7] The implication is that read requests have explicit timing constraints (as imposed by the issuing transactions). Write requests, on the other hand, usually do not have such constraints; they are typically issued by non-real-time processes, such as the buffer manager [AGM90].

We consider high end disk controllers (such as DEC HSC, IBM 3880) which internally perform I/O request buffering and scheduling (e.g., the Elevator algorithm). Furthermore, we assume that seek times are non-linear with seek distance [BG88]. In particular, we use the following formula to compute the access time ($Access(n)$) for an I/O request n tracks away from the current head position [BG88].

$$Access(n) = (DiskFactor\sqrt{n} + DiskConstant)\,\mathrm{msec},$$

where $DiskFactor$ is the seek time scaling factor and $DiskConstant$ measures the rotational latency plus the transfer time of an average request.

As in our Ethernet simulation, I/O requests are being generated according to a

[7] A write request is usually performed first on a local main memory copy of the data item [KS86]. For recoverability, a log record is then prepared for the write. High performance systems usually store the log file in stable memory or in a separate log disk or tape. These log operations do not interfere with the disk reads. The physical update to the disk copy of the data item is done by the buffer manager, typically after the transaction commit.

$MaxTrack$	λ	S_{min}	S_{max}	$DiskFactor$	$DiskConstant$
1000	43 req./sec.	36ms	326ms	0.6	15ms

Table 5.2: Baseline setting for disk simulation.

Poisson distribution at a rate of λ jobs per seconds. We model disk requests at the track level. Rotational optimization is not considered here. Each disk request can thus be represented by a track number. We assume that these track numbers (of disk requests) are uniformly distributed in the range $[1, MaxTrack]$, where $MaxTrack$ is the highest track number of the disk. Deadlines[8] of requests are computed according to the following formula:

$$Deadline = Arrival\_Time + Uniform([S_{min}, S_{max}])^{9}$$

where $Arrival\_Time$ is the arrival time of the request and $Uniform([a, b])$ represents a random slack uniformly taken from the range $[a, b]$.

Table 5.2 shows the parameter setting of our disk baseline experiment. Again, simulation results are obtained by averaging the statistics of 3 runs. Each run lasts 4,500 seconds simulation time. At a load of 43 requests/sec., about 193,500 requests are generated per run. The 95% confidence interval is $\pm$ 0.13 percentage point.

Figure 5.20 shows the percentage of missed deadlines by *Balance* and *Chop* obtained from our simulator as p varies. From the figure we see the characteristic performance curve of *Chop*. In the baseline experiment, the optimal value of MD_{Chop}

[8]Here a deadline of an I/O request is the time by which the request has to be *completed*, not *started* as we assumed for the analytical and Ethernet models.

[9]Note that the *execution time* does not appear in the deadline formula. It is because the amount of time it takes to satisfy a disk read depends on the position of the disk head before the read, which cannot be predicted. The *slack time* in the formula, generated from the uniform distribution, thus represents how much time is given to the disk read, instead of how much waiting time the request can be delayed without causing a missed deadline.

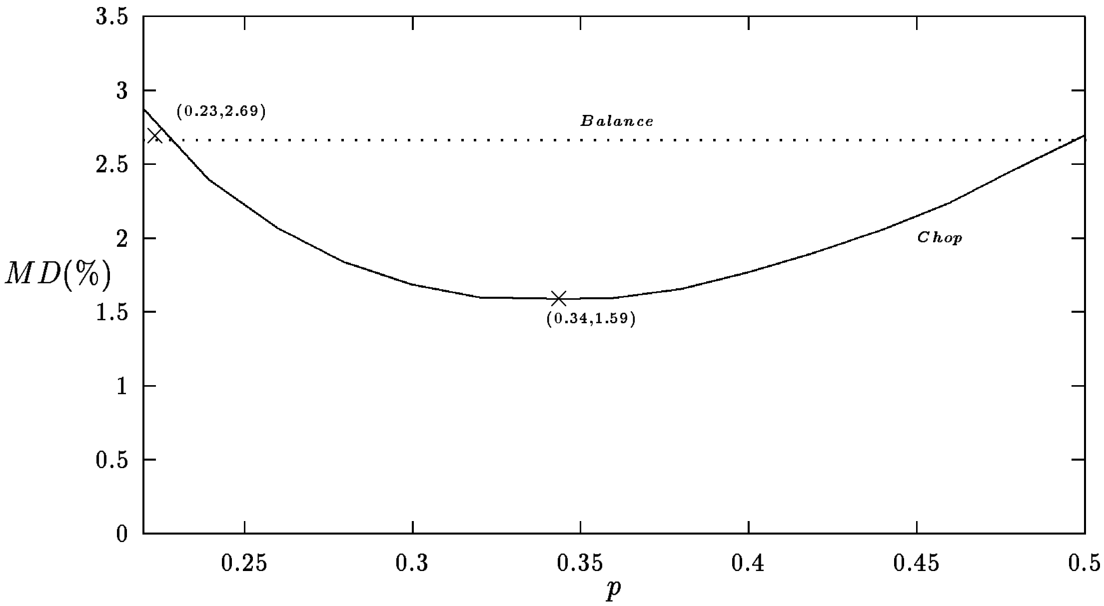


Figure 5.20: MD_{Chop} as a function of p. $\lambda = 43$ req./sec., $S_{min} = 36$ms, $S_{max} = 326$ms. Disk simulation.

is 1.59% which occurs at $p = 0.34$. This is about a 41% improvement over *Balance* (2.69%).

For disk scheduling, we do not compare the simulation result directly and point-by-point to the analytical result as we did for the Ethernet case. This is because the access time (or the service time) distribution of disk requests is not exponential. Yet, we would like to remark that the baseline setting (Table 5.2) of our disk experiment is chosen such that the values of $MD_{Balance}$ are relatively the same in all three cases we studied: analytical (2.64% in Figure 5.10), Ethernet (2.69% in Figure 5.18), and disk (2.69% in Figure 5.20). The performance of *Chop* in these cases thus reflects how well this strategy works compared with *Balance* in the various environments. From the figures, we notice that the performance of *Chop* in the disk simulation mimics closely to what the analytical model shows. Moreover, the relatively large underestimation of MD_{Chop} by the analytical model at low value of p in the Ethernet case (Figure 5.18) does not occur in the disk case. This is because no particular overhead is incurred when a disk is stressed with high load.

To further illustrate the effect of load stress on the performance of *Chop* in the Ethernet and disk environments, we increase the load of the two systems gradually from the baseline-setting load up to 50% more. Figures 5.21(a) and 5.21(b) compare the miss deadlines percentage of *Balance* and *Chop* in the Ethernet and disk models respectively. From Figure 5.21(a), we see the toll taken by load stress on *Chop* in the Ethernet environment. At high load, severe network contention discourages *Chop* from keeping the load of the high priority network low while routing too many loose-slack packets on the other network. As a result, *Chop* does not perform much better than *Balance* does at a high load situation. The same phenomenon does not occur in the disk case as shown by Figure 5.21(b). The *Chop* strategy is seen to blend well with the Elevator algorithm even when the system load is high.

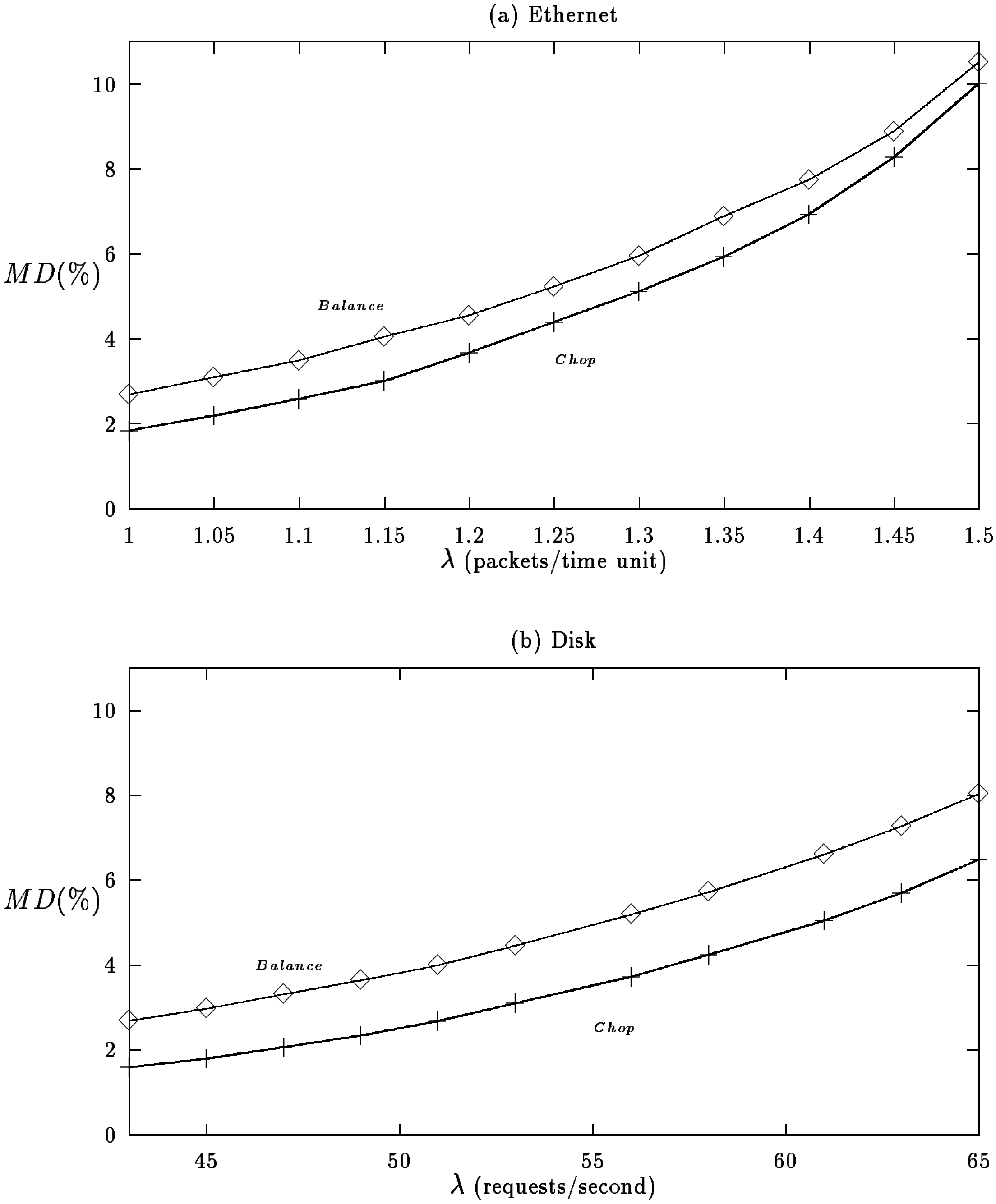


Figure 5.21: Effect of load stress in (a) Ethernet and (b) disk environments.

In this section we showed some representative simulation results of Ethernet and disk schedulings and compared them with that obtained by the analytical model. Though not shown in this thesis, we have performed extensive experiments with the simulators, varying parameters over wide ranges. In all cases, the simulation model confirms the trends identified by the analytical model. Among the observations we drew from the simulation results, we saw an important relationship between the performance of *Chop* and the reaction of a server to load stress. We noticed that *Chop* was particularly effective in cases when a high load did not cause significant overhead at the server. In such a case, the strategy of stressing one server while relaxing the other yielded a significant reduction in the missed deadline percentage.

5.7 Choosing p

As has been pointed out earlier, the performance of the *Chop* strategy depends on the p values used by the dispatchers. Although it is not crucial to have exactly the optimal p value for a given configuration (see, for example, Figure 5.10), it is important to have a reasonable value. Thus, an important question is how a real system would select such a p value.

There are many ways this could be done. For example, if the system parameters are static, the analytic model could be used to select the optimal p, which our simulation results show to be close to the actual optimal p. If the system is dynamically changing, it is possible to design an adaptive strategy that automatically tunes the p choice. Here we briefly describe one possible implementation of this dynamic tuning on a dual-Ethernet system.

As explained in Section 5.5, the search for the optimal p can be confined to the range [0,0.5]. We can thus start by operating the system at $p = 0.5$, which guarantees a performance at least as good as *Balance*'s. Our strategy is to continuously explore the neighborhood values of p for improvement. To achieve this, we elect a network co-ordinator whose job is to periodically collect missed deadlines statistics from its peer nodes, decide on a new p (by incrementing or decrementing the current value by some amount x), and then broadcast its decision.[10] If the new p does not cause a reduction in missed deadlines, the coordinator will switch the direction of p updates (e.g., from increments to decrements).

Since $MD_{Chop,uniform}$ is a convex function of p (see Equation 5.4), a local optimum is also the global optimum [PS82]. The above mentioned strategy, which is an example of nearest neighbor algorithms, is thus capable of locating the optimal p value with a discrepancy of the size of the stepwise update x. This argues for a small value of x. On the other hand, if x is chosen too small, it will take a long time for the system to converge to the optimal p value. Fortunately, as discussed in Section 5.5, the MD curve is quite flat near the optimal point. A small value of x, and hence an accurate estimation of the optimal p, is therefore unnecessary.

To demonstrate how well the algorithm works, we simulate a 10 node dual-network system as shown in Figure 5.17 with an exponential distribution for transmission time of mean 1 msec. The co-ordinator changes p once every 20 seconds by a step of 0.01. System load is initially 800 packets/second. To illustrate how the strategy adjusts to a changing load, at time 700 the load is reduced to 600 packets/second. Figure 5.22 shows a trace of the value of p as a function of time. From the figure we see, for example, that after startup it takes about 5.7 minutes (or 17 updates) for the

[10] If the co-ordinator dies, either a re-election is held or all dispatchers set p to 0.5. Again, the latter option ensures a performance no worse than *Balance*'s.

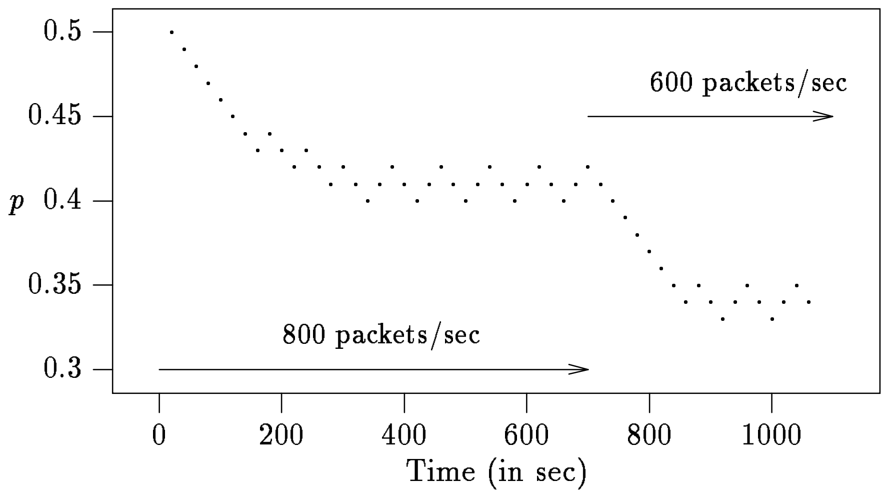


Figure 5.22: p adopted by nodes

system to arrive at its optimal operating point (at $p = 0.41$). At time 700 it takes 3.3 minutes (10 updates) for the system to automatically adjust to the new system load.

We emphasize that the p selection algorithm we have described does not make any assumptions on the packet arrival/service-time distributions. (Of course, the simulator used for testing it is using a specific choice of functions.) So, as long as the fraction of missed deadlines, MD_{Chop} is a convex function of p, the algorithm can locate the optimal p value.

5.8 Other Variations of The Base Model

Although the simulations of Section 5.6 show the performance of the *Balance* and *Chop* strategies in specific scenarios, we still continued to make three fundamental assumptions: (a) there were only two servers, (b) tasks were aperiodic, and (c) servers were not using real-time scheduling locally. How critical are these assumptions? Will *Chop* continue to perform well even if they are changed? In this section we address these questions. In particular, we extend our base model and study other system configurations. We first discuss how to modify the *Chop* strategy to accommodate more than two servers. We then look at the case when there are periodic tasks as well as aperiodic ones. The impact of having other execution time distributions (besides exponential), and other real-time queueing algorithms such as earliest-deadline-first and least-slack-first is also discussed.

5.8.1 Multiple Servers

Our discussion so far only considers a dual-server system. The idea of *chopping* up the load according to task slack can be applied to an n-server system as well. Suppose we have n servers named $Server_1$, $Server_2$, ..., $Server_n$. One way to partition the load would be to direct a p_1 fraction of the load with the tightest slack to $Server_1$, a p_2 fraction of the load with the *next* tightest slack to $Server_2$ and so on. As an example, if the slack distribution is uniform in the range $[S_{min}, S_{max}]$, then $Server_i$ would get all the tasks whose slack is in the range $[S_{min} + k(p_1 + ... + p_{i-1}), S_{min} + k(p_1 + ... + p_i)]$ where $k = S_{max} - S_{min}$ (see Figure 5.23). The load to $Server_i$ is $\lambda \cdot p_i$. We can use Equation 5.2 on page 91 to compute the fraction of missed deadlines at each server. Since the p_i's sum to one, the overall fraction of missed deadlines can be expressed as a formula with $n-1$ parameters: $MD_{ns,uniform}(p_1, ..., p_{n-1})$. The values of the p_i's are then determined by locating the minimum of $MD_{ns,uniform}()$.

The above way of determining the servers' load works but is computationally expensive and is not practical especially when n is large. Alternatively, we can approximate the values of the optimal p_i's by considering a *tree of dispatchers*. We first divide the server pool into a *tight-half* with $\lceil n/2 \rceil$ servers and a *loose-half* with $\lfloor n/2 \rfloor$

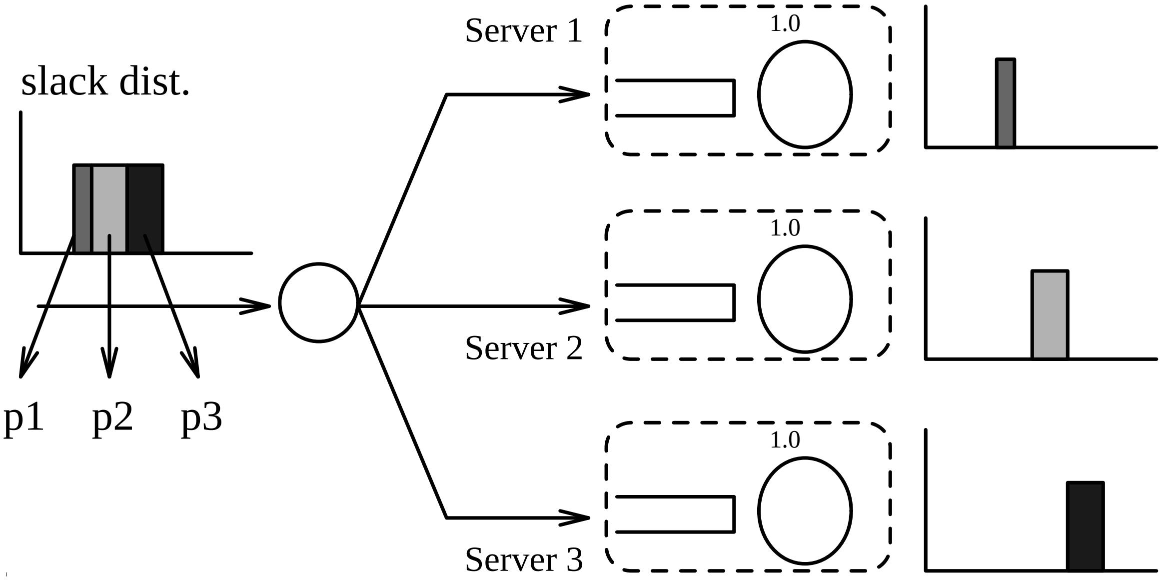


Figure 5.23: n servers.

servers. We then consider the two halves as two servers with servicing capacities of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ times of a single server. The original *Chop* strategy is then applied to determine the fraction of the load (and hence the slack range) to be directed to the two halves. This process is recursively repeated for each half until the load to each individual server is determined. With the *tree* method, we have to locate the minimum points of $n-1$ curves, each with one parameter. This is considerably less expensive to do than finding the optimal point for $MD_{ns,uniform}()$, which has $n-1$ variables. We have compared the two methods on a 3-server system and found that they give almost identical performance on missed deadlines.

When the system is dynamic, or when a priori information about load and slack is not available, the use of the analytical formula to estimate the values of the p_i's fails. To locate the optimal values of $n-1$ variables in a dynamic system could be very difficult. For this reason, methods that use less number of parameters are desirable. Here, we mention two. One method, which we refer to as the p-n_1 method, partitions

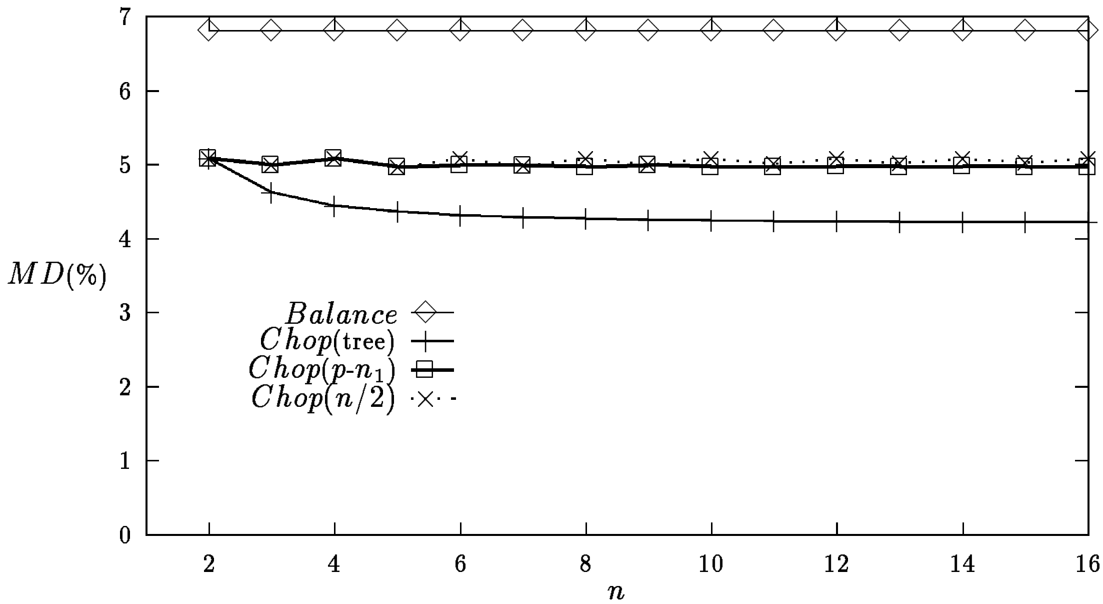


Figure 5.24: Performance of *Chop* with different approximation methods.

the server pool into two parts. One with n_1 servers, the other with $n - n_1$ servers. The two pools of servers are considered as two single servers with different capacities. The *Chop* strategy is then applied to this dual to determine their load. Among each server pool, the *Balance* strategy is used to sub-divide the assigned load. The p-n_1 method, which uses two variables (p and n_1), can be simplified further by fixing n_1 to $\lfloor n/2 \rfloor$. We call this the $n/2$ method.

Figure 5.24 compares the performance of *Chop* using different approximation methods on an n-server system. The performance of *Balance* is also shown. In the figure, we vary n, the number of servers, and measure the fraction of missed deadlines. The system load is $0.7 \times n$, the average utilization of the system is therefore 0.7. The slack distribution is [0.5,30].

From the figure, we see that *Chop* with the *tree* method performs best. It reduces MD by about 38% over *Balance*. The p-n_1 method and the $n/2$ method also do fairly well, with an MD reduction of about 27%. Further experiments with different parameter values show that the p-n_1 and $n/2$ methods perform very similarly. Therefore, in most situations, the use of the parameter n_1 is unnecessary. In conclusion, in an n-server system, if the load is relatively static with a known slack distribution, the *Chop* strategy with the tree method is the algorithm of choice. In a more dynamic environment, however, the $n/2$ method, which performs well, is more manageable with only one parameter to tune, and still gives significant improvements over the *Balance* strategy.

5.8.2 Periodic Tasks

Periodic tasks exist in many real-time systems. Sensor readings, voice packets, etc. are usually generated periodically. One nice feature of periodic tasks is that they are more or less predictable in terms of their execution time and slack time. This property should make it possible for a strategy like *Chop* to improve performance. To verify this, we simulated a system with both periodic and aperiodic tasks. Here, we only show results from a single experiment; many others were performed and their results are similar.

For this experiment, we simulated a dual FCFS server system with the *Chop* dispatcher installed. Tasks are of two types: periodic and aperiodic. There are 10 streams of periodic tasks. Stream i $(1 \leq i \leq 10)$ has a period of $1.5 + 2.95i$[11]. Note that tasks with shorter period contribute more load to the system than tasks with

[11] These numbers are picked so that the slack of periodic tasks spans the same interval ([0.5,30]) as that of aperiodic tasks.

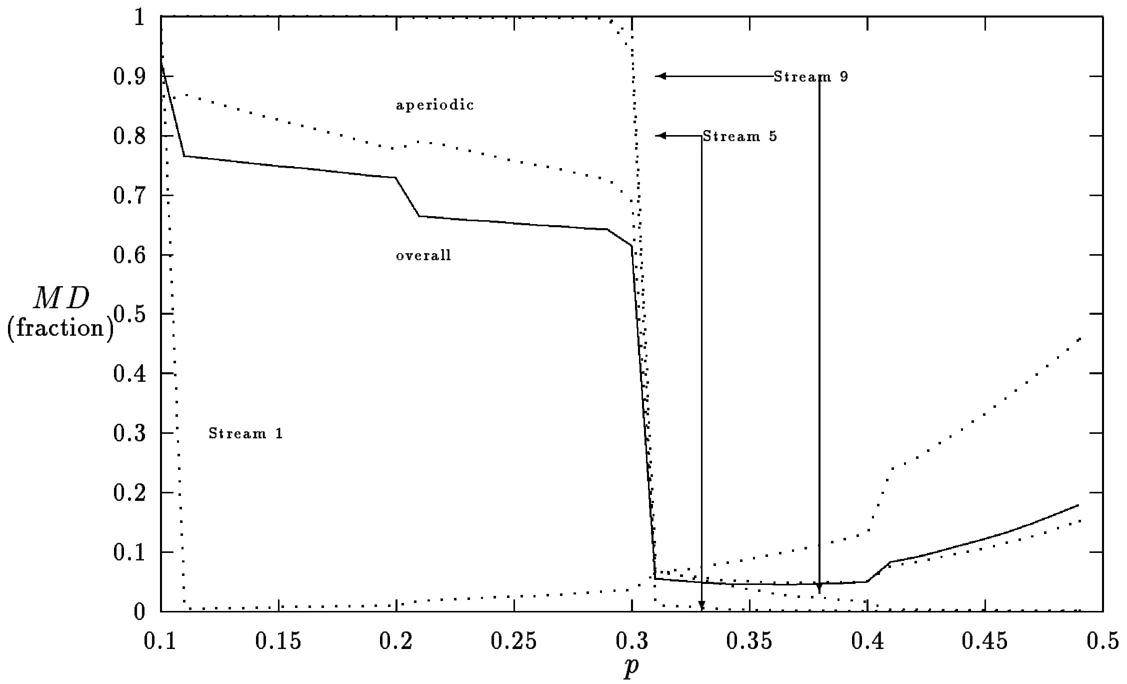


Figure 5.25: Simulation Result with Periodic Tasks.

longer period. We assume that the deadline of a periodic task is the arrival time of the next task of the same stream. The execution time of a periodic task is set at 1.0. For aperiodic tasks, they arrive according to a Poisson distribution. Their execution time is exponential with an average of 1.0. Their slack distribution is uniform in the range [0.5,30]. The total load to the system is 1.6, with which half of the load is due to aperiodic tasks.

Figure 5.25 shows the fraction of missed deadlines of three periodic task streams (Streams 1, 5, and 9), of the aperiodic task, and of the system as a whole. We see that there are "jumps" in the curves which occur at regular interval (when $p = 0.1$, 0.2, 0.3, 0.4,..). This is due to the fact that the dispatcher only direct tasks according to their slack, and periodic tasks take on only ten discrete levels of slackness in our simulation. As an example, when p is smaller than 0.1, all periodic tasks are directed to Server 2, saturating it. When p is between 0.1 and 0.2, Stream 1 goes to Server 1, which is lightly load, and almost all deadlines are met there. Server 2, on the other hand, is still saturated by the other 9 periodic streams, plus 90-80% of the aperiodics. When p is larger than 0.3, the load of Server 2 is light enough for it to handle most of its tasks, and there is a large drop in missed deadlines. Finally, when the value of p is between 0.3 and 0.4, the percentage of system missed deadlines ranges from 4.5% to 5.0%. This compares very favorably to *Balance*, which misses 8.7% of the deadlines (not shown in graph).

In conclusion, the inclusion of periodic tasks do change the behavior of the system, e.g., it introduces "jumps" in the MD curve. However, the curve still exhibits roughly the same convex shape. Also, *Chop* can still yield significant performance gain over *Balance* (almost halved the number of missed deadlines for the particular scenario shown here.)

5.8.3 Others

We have studied the performance of the dispatcher algorithms under other different scenarios. For example, we simulated the system with different task execution time distributions (normal, uniform, etc.) and different arrival patterns. We have also modified the *Chop* dispatcher so that it directs tasks not by their slackness but by their deadlines. In any case, the same general conclusion can be drawn as that from the base-line experiments.

In this chapter, we focus our attention on non-real-time servers. We assume that these servers are not capable of doing real-time scheduling, such as earliest-deadline-first and least-slack-first. Since the *Chop* strategy can very well be applied to systems with real-time servers, it is thus interesting to study how it fairs in this environment. To this end, we simulated a dual real-time server system. Our results show that the performance benefit of applying *Chop* over *Balance* is not impressive with real-time servers. The reason is that *Chop*, which stresses one server while relieving the other, does not perform outstandingly under a high load condition (see Figure 5.13). Also, under low load, the real-time schedulers alone are capable of maintaining a very low missed-deadline percentage and therefore a sophisticated dispatcher algorithm, such as *Chop*, is not necessary.

5.9 Conclusion

In this chapter we have discussed how multiple non-real-time servers can be used to support soft real-time scheduling. Our approach is unusual (at least from the perspective of conventional real time systems) because we are assuming that (1) we cannot afford or do not have available a real-time server; and (2) we can tolerate some missed deadlines (soft real-time systems). We believe there is a wide class of applications where this is true, and where strategies like *Chop* can significantly reduce the number of deadlines that are missed.

Through a simple analytical model, we studied and compared two ways of utilizing the replica: a balance scheme in which load is evenly distributed, and a prioritization strategy in which jobs are segregated according to their slackness. We show that in cases where tight slack jobs cause a disproportionately high number of missed deadlines, segregation can significantly improve the system's real-time behavior. This improvement is particularly marked (e.g. *Chop* misses up to a *factor* of 3 fewer deadlines than *Balance* in Figure 5.16) when the system load is moderate and there is a wide variation in job slackness. In practice, we would expect the load on a real-time network to be moderate, else too many deadlines would be missed. Thus, it seems reasonable to expect *Chop* to perform much better than *Balance*.

To verify the analytical results and to study more realistic environments, a dual-Ethernet system and a mirrored disks system were simulated. The results indicate that the analytical model indeed captures the essential system characteristics, and is a useful tool for studying such systems.

It is interesting to note that our basic idea, that of splitting load according to real-time slackness, can be applied to other replicated system components. For example, in a database system with replicated database servers, read requests could be routed according to transaction slack. Similarly, in a multiprocessor system, requests could be queued by slackness.

Chapter 6

Static Priority Schedulers

In the previous chapter we discussed how system performance degraded when real-time components were replaced by FCFS components. Very often, standard non-real-time components can do better than FCFS by scheduling tasks using static priority (e.g., POSIX UNIX and token ring networks). In this chapter we discuss how we can utilize these static priority schedulers in a distributed soft real-time system, and how much performance improvement one can achieve over FCFS schedulers.

One idea is to emulate a real-time scheduling algorithm using static priority. A translator is written for each component in the system that uses static priority levels (see Figure 6.1). The job of the translator is to intelligently pick a priority level based on the real-time attributes of a task, such as its deadline, arrival time, slack etc., and the number of priority levels provided by the underlying component. The idea is to pick these levels carefully such that the behavior of the priority scheduler mimics that of the target real-time scheduling algorithm.

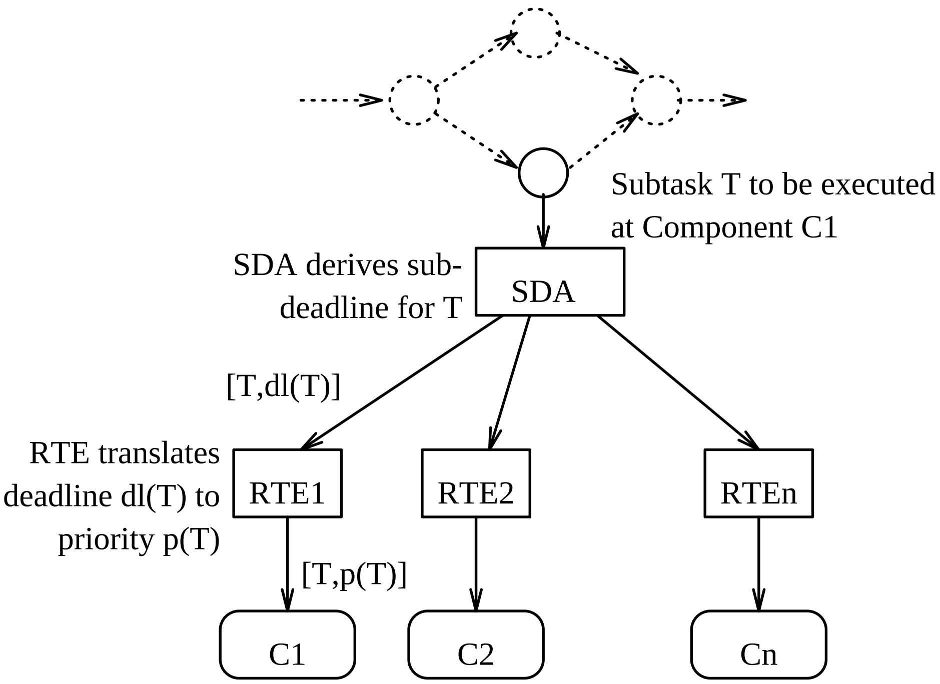


Figure 6.1: The Real-Time Emulation layer.

In [AGMK94], different mapping functions for various target real-time scheduling algorithms on a *uniprocessor system* are studied. Here we mention a simple one called *Earliest Deadline Relative* that can be easily adopted to our distributed system. Before a subtask T_i is submitted to a component C_j, the process manager first computes T_i's sub-deadline $dl(T_i)$ using EQS (see Chapter 4). The sub-deadline is then used to compute a priority level ($priority(T_i)$) for T_i according to a linear mapping:

$$priority(T_i) = (dl(T_i) - ar(T_i))/ts_j$$

where $ar(T_i)$ is the arrival time of T_i and ts_j is a tuning parameter for component C_j. In [AGMK94], it is suggested that ts_j be set to

$$ts_j = \frac{\mu_j + 3\sigma_j + S_{max}}{n_j}$$

where

μ_j and σ_j are the average and standard deviation of task execution time at Component j,

S_{max} is the maximum task slack at Component j, and

n_j is the number of priority levels at Component j.

Readers are referred to [AGMK94] for a derivation of the above formula.

To test the applicability of the emulation method, we implemented the mapping function in our process manager and studied performance. Figure 6.2 compares the global task miss rate under three scenarios of local schedulers: (1) all-FCFS, (2) all-EDF, and (3) all components use static priority scheduling with 4 priority levels, and the process manager uses the real-time emulation mapping with the ts's set to 4.5 (RTE).

From the figure, we see that even with only four priority levels, static priority scheduling with RTE performs comparably to EDF. At higher load, RTE actually outperforms EDF slightly. There are two reasons for the slight difference in performance between the RTE configuration and the all-EDF configuration. First, the RTE policy schedules tasks according to how much time they have got (deadline - arrival time) instead of their deadlines. Second, under high load, EDF tends to give preference to tasks that are late. Because of its coarse priority granularity, in overload situations RTE cannot order all tasks properly by their urgency. Thus, RTE essentially gives non-tardy tasks a chance to finish before their deadlines.

The disadvantage of the RTE approach is that its performance is sensitive to the ts value and the distribution of tasks' slack. In our baseline model, the slack distribution is uniform and thus a linear mapping works well. For other slack distributions, different mappings that can evenly distribute the task population into the limited

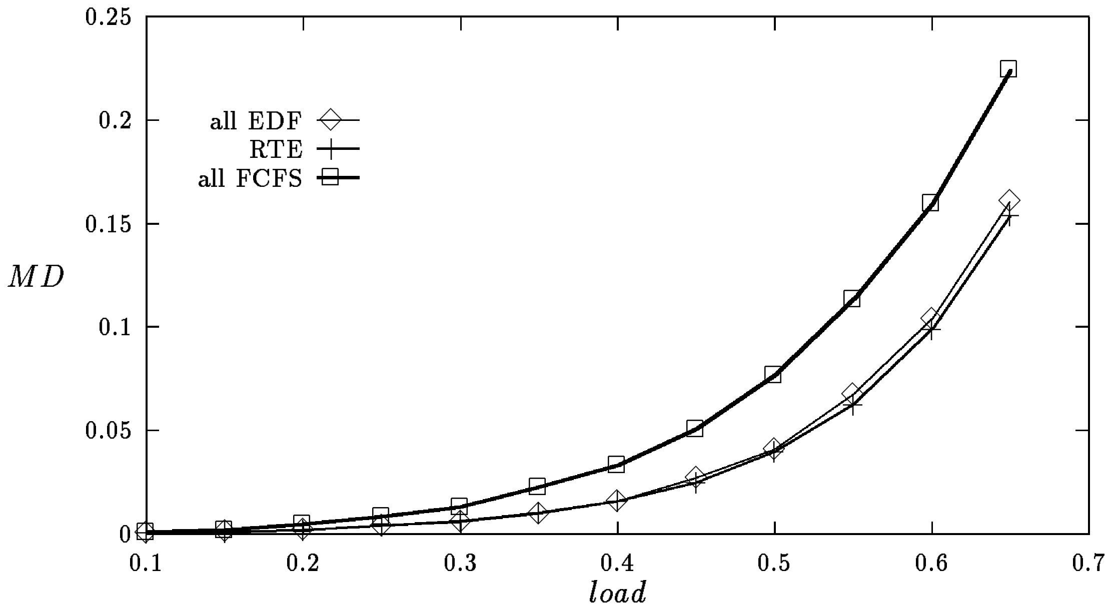


Figure 6.2: Performance of RTE

priority levels are needed to ensure good performance. This adds to the complication of the process manager. However, when a real-time component is not available, RTE is seen to be an effective and economical way of reducing missed deadlines.

Chapter 7

Conclusions

In this thesis we have explored a non-traditional type of real-time system where one has to use some conventional non-real-time components due to cost or availability considerations. Of course, in a *hard* real-time system, one cannot use any non-real-time components. However, we believe that there are many applications where deadlines are soft and one does wish to consider the various cost vs availability of components vs real-time performance tradeoffs.

In Chapter 4 we defined and studied the Subtask Deadline Assignment Problem. We showed how global task miss rate could be greatly reduced by appropriately assigning virtual deadlines to subtasks. We identified two sub-problems of SDA: the serial subtask problem (SSP) and the parallel subtask problem (PSP). Various deadline assignment strategies were suggested and their performance were compared through an extensive simulation study. The algorithms EQF and DIV were shown very effective for the SDA problem.

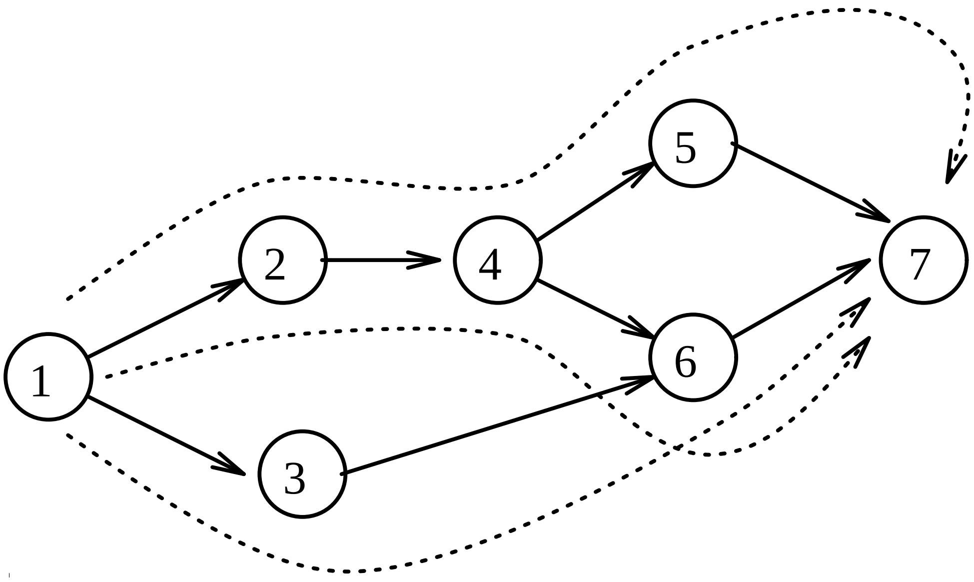


Figure 7.1: A non-serial-parallel task.

In our model, all global tasks have a serial-parallel structure with arbitrary levels of nesting. An extension to our SDA algorithm would be to include global tasks that are not serial-parallel (Figure 7.1). One method of assigning deadlines in this case is to identify all possible paths from the starting subtask (Task 1 in Figure 7.1) to the terminating subtask (Task 7). We then treat each path as a serial global task and use the SSP strategy to determine the deadlines of the subtasks along each path. If a subtask falls on multiple paths (such as Task 4), its deadline would be the minimum of the deadlines computed on all the paths. This method is, however, computationally expensive if global tasks are very complex with many forks.

Another assumption made in our SDA study was that the whole task graph of a global task was known when the global task was generated. This assumption may not be true for some applications. For example, there could be multiple *alternative* execution plans for a global task. The plan to be executed could depend on the result of a previous subtask. Figure 7.2 shows an example of alternative execution plans. In

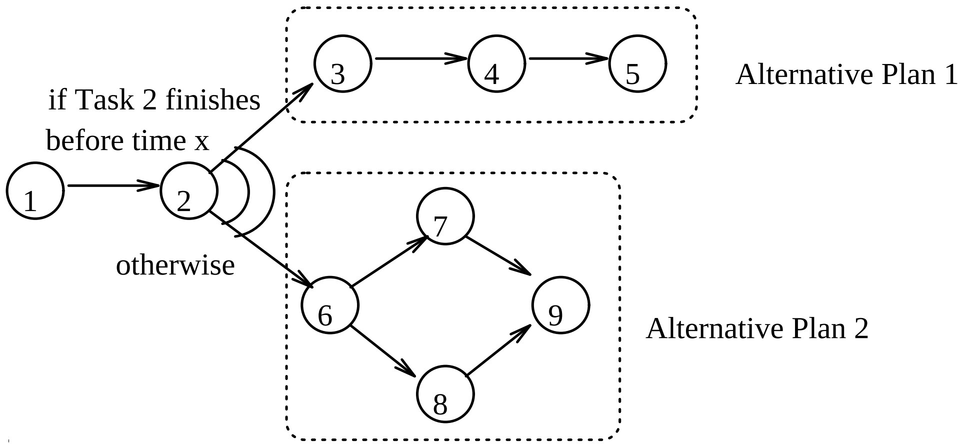


Figure 7.2: An example of alternative execution plans.

the picture Plan 1 is executed if Task 2 is completed before a certain time, otherwise, Plan 2 is executed. This corresponds to the situation when the system is delayed unexpectedly and a contingency plan (Plan 2), which could be less computationally demanding than the original plan, is executed instead. The SDA algorithms have to be modified to accommodate this type of global tasks.

In Chapter 5 we illustrated how system performance might degrade when EDF schedulers were replaced by FCFS schedulers. In general, when the system components are evenly utilized and their loads kept low, using a few non-real-time components does not overly increase the miss rate. It may be more cost effective to use a few standard components when specialized real-time ones are expensive. However, for systems that use one component more often than the others, it is important that the highly loaded components be equipped with real-time schedulers. In case real-time schedulers are not available, we should either increase the capacity of these components (for example, by duplication) to minimize the effect of FCFS scheduling (Chapter 5), or in case static priority schedulers are available, use a real-time emulation algorithm that maps deadlines to priorities (Chapter 6). Our results suggested that the latter method would perform very similarly to a real EDF scheduler.

When multiple identical components were available, our study shown that the server prioritization policy (*Chop*) could reduce the miss rate effectively. In many cases, up to 50% improvement was achievable over random routing (*Balance*). Although our study has only showed how server prioritization is applied to homogeneous components, the technique can be easily extended to handle heterogeneous servers – servers with different capacities, using the same basic principle.

When building a distributed soft real-time system, it is best to use components that can handle real-time tasks. If the local schedulers schedule tasks based on their deadlines, we should use one of the deadline assignment schemes to derive sub-deadlines for subtasks. If real-time components are not available or are too costly, we can use components with priority schedulers and build on top of them the real-time emulation layer. If FCFS schedulers are what we have, we should try to duplicate the components for more processing power. Server prioritization should be used to improve the system's ability of meeting deadlines. The subtask deadline assignment, real-time emulation, and server prioritization strategies are all orthogonal and can be employed together.

Bibliography

[AGM88a] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *ACM SIGMOD Record*, pages 1–12, 1988.

[AGM88b] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Procceedings of the 14th VLDB Conference*, 1988.

[AGM90] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 113–124, 1990.

[AGMK94] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *IEEE Real-Time System Symposium*, 1994.

[BG88] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th VLDB Conference*, pages 331–338, 1988.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BL90] R. Bettati and J. W. S. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Systems*, 1990.

[BL92] R. Bettati and J. W. S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *Proceedings of the 12th International Conference on Distributed Computing*, pages 452–459, 1992.

[Car95] J. A. Caruso. The challenge of the increased use of COTS: A developer's perspective. In *Proceedings of the Third Workshop on Parallel and Distributed real-Time Systems*, 1995.

[CJL89] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of the 15th VLDB Conference*, pages 397–410, 1989.

[CL87] W. W. Chu and L. Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on Computers*, C-36(6):667–679, 1987.

[CL88] S. Chiou and V. O. K. Li. An optimal two-copy routing scheme in a communication network. In *IEEE INFOCOM*, pages 288–297, 1988.

[CSR86] S. Cheng, J. A. Stankovic, and K. Ramaritham. Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–174, 1986.

[GHW90] J. Gray, B. Horst, and M. Walker. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. In *Proceedings of the 16th VLDB Conference*, pages 148–161, 1990.

[GMKB91] H. Garcia-Molina, B. Kao, and D. Barbara. Aggressive transmission over redundant paths. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, pages 198–207, 1991.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., 1993.

[HS92] C. Hou and K. G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 146–155, 1992.

[HTT89] J. Hong, X. Tan, and D. Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Trans. Computers*, 38:1736–1744, Dec. 1989.

[IL88] I. Iliadis and L. Y Lien. Resequencing delay for a queueing system with two heterogeneous servers under a threshold-type scheduling. *IEEE Trans. Communications*, 36(6):692–702, June 1988.

[KGM92] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. Technical Report STAN-CS-92-1452, Stanford University, 1992.

[Kle76] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, 1976.

[KMST86] C. Knessl, B. J. Matkowsky, Z. Schuss, and C. Tier. Two parallel queues with dynamic routing. *IEEE Trans. Communications*, 34(12):1170–1175, Dec 1986.

[KS86] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1986.

[KSF91] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-tiome communication in multi-hop networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 300–307, 1991.

[KSY84] J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *Computing Survey*, 16(1):43–70, 1984.

[L+87] E. L. Lawler et al. Pipeline scheduling: A survey. Technical report, Technical Report RJ 5738, IBM Research Division, San Jose, 1987.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD*, pages 53–60, 1986.

[Liv90] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[LK84] W. Lin and P. R. Kumar. Optimal control of a queueing system with two heterogeneous servers. *IEEE Trans. Communications*, 29(8):696–703, Aug 1984.

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

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[ML85] M. L. Molle and L.Kleinrock. Virtual time CSMA: Why two clocks are better than one. *IEEE Trans. Communications*, 33(9), sept 1985.

[ML90] J. W. Mark and B. Lee. A dual-ring LAN for integrated voice/video/data services. In *IEEE INFOCOM*, pages 850–857, 1990.

[NH86] R. M. Newman and J. L. Hullet. Distributed queueing: A fast and efficient packet access protocol for QPSX. In *Proceedings of the 8th International Conference on Computer Communications*, pages 294–299, 1986.

[NSM89] M. Nakamura, I. Sasase, and S. Mori. Two parallel queues with dynamic routing under a threshold-type scheduling. In *IEEE GLOBECOM*, pages 1445–1449, 1989.

[OR88] A. Orda and R. Rom. Routing with packet duplication and elimination in computer networks. *IEEE Trans. Communications*, 36(7):860–866, July 1988.

[PLC92] H. Pang, M. Livny, and M. J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of IEEE Real-Time Systems Symposiuim*, 1992.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, INC, 1982.

[PS85] J. L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[PSS+94] B. Purimetla, R. M. Sivasankaran, J. A. Stankovic, K. Ramamritham, and D. Towsley. Priority assignment in real-time active databases. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 176–184, 1994.

[Ram90] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[RS91] P. Ramanathan and K. G. Shin. A multiple copy approach for delivering messages under deadline constraints. In *FTCS-21*, pages 300–307, 1991.

[RSS90] K. Ramaritham, J. A. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, 1990.

[SM89] I. Sasase and S. Mori. Analysis of queueing systems with multiple servers under a threshold-type scheduling. In *IEEE GLOBECOM*, pages 1450–1454, 1989.

[SRC85] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on computers*, 34(12):1130–1143, 1985.

[SSS92] L. Sha, S. Sathaye, and J. Strosnider. Scheduling real-time communication on dual-link networks. In *Proceedings of IEEE Real-Time Systems Symposiuim*, pages 188–197, 1992.

[Tan88] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1988.

[tan91] *Introduction to NonStop Systems, 16270.* Tandom Computers Inc. Cupertino, CA, 1991.

[Tay92] Y. C. Tay. A behavioral analysis of scheduling by earliest deadline. Technical Report 532, Department of Mathematics, National University of Singapore, 1992.

[WRS92] F. Wang, K. Ramamritham, and J. Stankovic. Bounds on the performance of heuristic algorithm for multiprocessor scheduling of hard real-time tasks. In *IEEE real-time systems symposium*, pages 136–145, 1992.

[ZR87] W. Zhao and K. Ramamritham. Virtual time CSMA protocols for hard real-time communication. *IEEE Transactions on Software Engineering*, 13(8):938–952, 1987.