Scheduling Soft Real-Time Jobs Over Dual Non-Real-Time Servers

Benjamin Kao and Hector Garcia-Molina, *Member, IEEE*

Abstract—In this paper, we consider soft real-time systems with redundant *off-the-shelf* processing components (e.g., CPU, disk, network), and show how applications can exploit the redundancy to improve the system's ability of meeting response time goals (soft deadlines). We consider two scheduling policies, one that evenly distributes load (*Balance*), and one that partitions load according to job slackness (*Chop*). We evaluate the effectiveness of these policies through analysis and simulation. Our results show that by intelligently distributing jobs by their slackness amount the servers, *Chop* can significantly improve real-time performance.

Index Terms—Real-time systems, scheduling, dual servers, soft deadlines, distributed systems.

1 Introduction

Traditional hard real-time systems handle jobs with stringent time constraints. Job deadlines are guaranteed by 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 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 [10]. Here again, some missed deadlines (up to 5%) may be tolerable.

Even in cases where end-users have no deadlines, it may still be highly desirable to make applications more responsive to user 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 [4] 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 [18] 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.

• B. Kao is with the Princeton University Department of Computer Science. His current address is: Department of Computer Science, Stanford University, Stanford, CA 94305. E-mail: kao@cs.stanford.edu.
• H. Garcia-Molina is with the Department of Computer Science, Stanford University, Stanford, CA 94305. E-mail: hector@cs.stanford.edu.

Manuscript received Jan. 13, 1993; revised Mar. 15, 1995.

For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number D95061.

To build a soft real-time system, one may or may not be able to use specialized real-time hardware, such as a real-time network or a disk that schedules I/O requests according to deadlines. Real-time hardware would obviously make it easier to meets system deadlines. Unfortunately, real-time hardware is more expensive and is not readily available. For instance, off-the-shelf, easy-to-maintain and inexpensive networks do not usually provide real-time facilities. 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. Even for systems that provide priorities, they usually allow only a limited number of priority levels [32]. This renders real-time scheduling algorithms ineffective [3]. As another example, *sophisticated* disk controllers that employ algorithms such as the Elevator Algorithm [27] 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 [6], [2].

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. However, 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 [1]), 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 [10]. 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 paper we study how these redundant components can be exploited 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, we will first consider a dual-server system as shown in Fig. 1. (We will extend our model later and consider an n-server system in Section 7.) This system processes jobs with soft real-time constraints. Servers are assumed to be identical in their processing ability. 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.

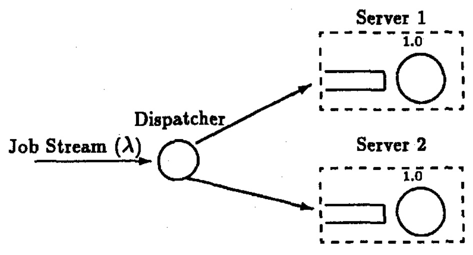


Fig. 1. A queueing model.

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 trade-offs 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 two 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.

1.1 Related Work

Systems using dual (or multiple) servers processing a single job stream have been analyzed extensively by queueing theorists [23], [31], [15], [17], [12]. These studies usually consider non-real-time systems and focus on the queueing delay of jobs. For example, in [15], 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 [12], [23] 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. Others studies that consider queueing systems employing real-time scheduling algorithms like earliest-deadline-first can be found in, e.g., [35], [11]. However, these studies are usually confined to single-queue systems.

The problems of scheduling *hard* real-time jobs in distributed and multiprocessor environments are studied in [33], [28], [29], [36]. The focus is on coordinating the schedulers at the multiple components closely to ensure job deadlines are always met. This paper assumes a much looser control over the local schedulers. For example, each component employs its own local scheduling routine. Also, the focus is on using off-the-self standard components for *soft*, instead of hard, real-time jobs.

There is also work done on real-time communication using multiple paths. In [20], 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. Another idea on sending *duplicate* packets over redundant paths to enhance communication timeliness is studied in [7], [25], [30], [8]. Finally, the use of disk shadowing and disk striping to improve fault-tolerance and I/O system performance is considered in [5], [9].

The rest of this paper is organized as follows. Sections 2 through 4 present the analytical model and results. In particular, Section 2 presents the model, Section 3 shows the analysis of a single server system, and Section 4 extends the analysis to a dual-server system with various dispatcher strategies. Sections 5 presents the results of Ethernet simulations and disk simulations. A discussion on some of the practical aspects of job prioritization is presented in Section 6. In Section 7, some extensions and variations of the base model are discussed. For examples, we look at periodic tasks, multiple servers, and other queueing disciplines. Finally, we conclude the paper in Section 8.

2 The Model

Our analytical model is shown in Fig. 1. 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. 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 paper:
   a) A uniform distribution over the range $[S_{min}, S_{max}]$ (Fig. 2a). That is, slacks that 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

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

   b) A discrete two-spike distribution (Fig. 2b). 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.

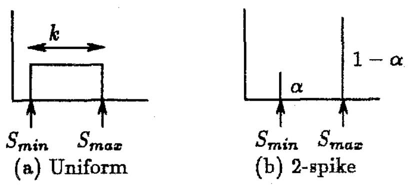


Fig. 2. Slack distributions.

As stated in the introduction, in our study 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. We assume that jobs in the system is of equal importance and the primary performance metric is the percentage of jobs that missed their deadlines. The smaller this number, the better a system performs.[1]

As discussed in Section 1, 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).[2] Again, in our simulations (Section 5) 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 7.

3 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 Fig. 1, 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 4.

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 [14]:

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

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

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

Using the uniform slack distribution of Section 2 (Fig. 2a) and defining $MD_{1s,uniform}$ to be the fraction of missed deadlines for the single server case with the uniform slack distribution, we obtain:

$$\begin{aligned} MD_{1s,uniform} &= \int_{S_{min}}^{S_{max}} \frac{1}{k}[\lambda e^{-(1-\lambda)y}]dy \\ &= \frac{\lambda}{k(1-\lambda)}[e^{-(1-\lambda)S_{min}} - e^{-(1-\lambda)S_{max}}]. \end{aligned} \qquad (2)$$

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 Fig. 3 for $S_{max} = 10$, 30, and 100 time units.[3] 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.

1. An alternative model would consider jobs of different *values*, and the performance goal would be to maximize the total value obtained by completing jobs in time (see [13]). We do not consider this model in this paper.

2. 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 [24], and a variation of the Virtual Time CSMA protocol [22]) approximate a FCFS order of packet transmissions on a bus.

3. Recall that the average job service time is 1 time unit.

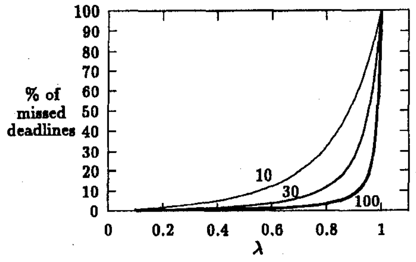


Fig. 3. $MD_{1s,uniform}$ as a function of λ. S_{min} = 0.5. (Curves are labeled with corresponding S_{max} values.)

Next we consider the 2-spike slack distribution (see Fig. 2b). Recall that α is the fraction of jobs with slack = S_{min}. Substituting this distribution into (1) we obtain:

$$MD_{1s,2\text{-}spike} = \alpha[1 - T(S_{min})] + (1 - \alpha)[1 - T(S_{max})]$$

$$= \alpha\lambda e^{-(1-\lambda)S_{min}} + (1 - \alpha)\lambda e^{-(1-\lambda)S_{max}}. \quad (3)$$

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 $S_{max} = 19$. Fig. 4 shows $MD_{1s,2\text{-}spike}$ (solid line) as a function of λ. (The dotted lines in Fig. 4 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 is the same in Fig. 3 (curve for $S_{max} = 30$) and Fig. 4 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 $\lambda = 0.5$, $MD_{1s,2\text{-}spike}$ = 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 Fig. 4 are the components of $MD_{1s,2\text{-}spike}$ (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 (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 $MD_{1s,2\text{-}spike} = C + D$.

From Fig. 4, we observe that even though tight slack jobs represent only 20% of the population, they contribute much more to $MD_{1s,2\text{-}spike}$ 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.

4 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

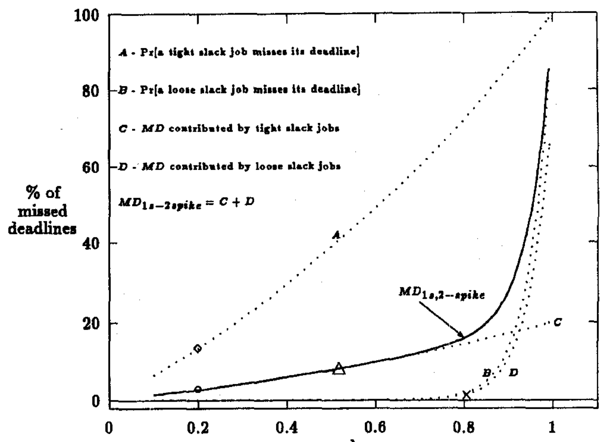


Fig. 4. $MD_{1s,2\text{-}spike}$ as a function of λ, two job classes with slacks equal 0.5, and 19, $\alpha = 0.2$.

slack jobs. It handles jobs whose slacks are at the lower *p* quartiles of the slack distribution. Server S_L handles the rest of the load, i.e., jobs with slack in the upper (1 – *p*) quartile of the distribution (looser slack) (See Fig. 5 for the uniform slack distribution case).

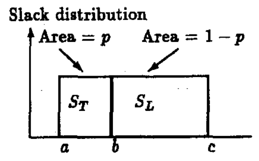


Fig. 5. *Chop* strategy: Tasks with slack in the range [*a*, *b*] ([*b*, *c*]) are serviced by S_T (S_L).

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.[4] Instead, the goal is to have a reduced load on S_T. To see the 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 Fig. 4. (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 $p = 0.2$ (Fig. 6), 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

4. If the slack and execution time distributions are not independent (for example when lengthier 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.

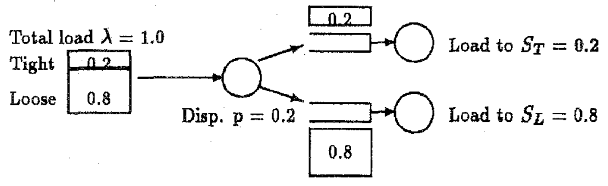


Fig. 6. Example of *Chop* with 2-spike slack distribution.

from curve *A* in Fig. 4. (This is the percentage of 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 (2) and (3), with λ replaced by $\lambda/2$:

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

and

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

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 (2), the fraction of missed deadlines for S_T is

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

By similar argument,

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

Finally,

$$MD_{Chop,uniform}$$
$$= p \cdot MD_{Chop,uniform}(S_T) + (1-p) \cdot MD_{Chop,uniform}(S_L). \qquad (4)$$

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

CASE 1: ($\alpha \le p$). 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 (3) we have:

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

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

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

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 (3), we have,

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

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

In either case 1 or 2,

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

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 trade-offs involved. For the base setting we then vary one or two parameters at a time, showing their impact on performance. Due to space limitation we do not present all of our results; we have selected the ones we consider the most illustrative.

For our base setting we choose $\lambda = 1.0$,[5] 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.

Fig. 7 shows $MD_{Chop,uniform}$ as a function of *p*. Also shown is $MD_{Balance,uniform}$ (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.

Fig. 8 shows a graph similar to Fig. 7 but for the 2-spike slack distribution. We see in this case that *Chop* performs bet-

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

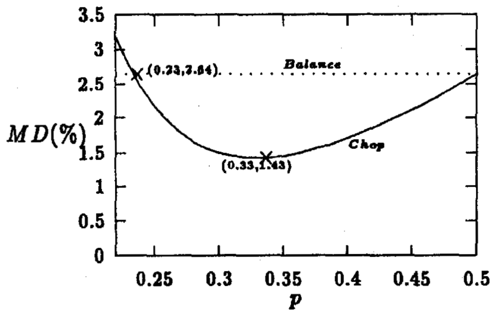


Fig. 7. $MD_{Chop,uniform}$ as a function of p. $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 30$. Numbers in parentheses give the coordinates of marked point.

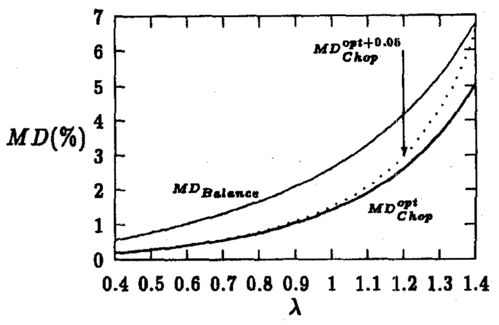


Fig. 9. $MD_{Balance,uniform}$, $MD_{Chop,uniform}$ as λ varies.

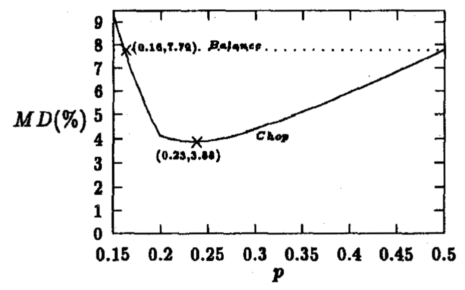


Fig. 8. $MD_{Chop,2-spike}$ as a function of p. $\alpha = 0.2$, $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 19$.

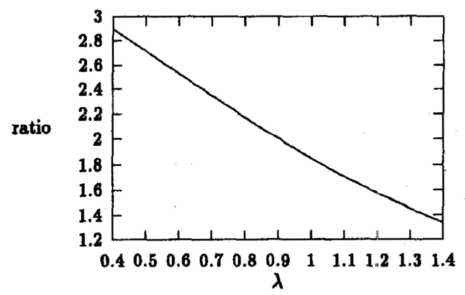


Fig. 10. $MD_{Balance,uniform}/MD_{Chop,uniform}$ as λ varies.

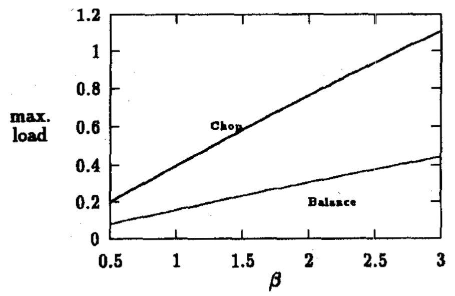


Fig. 11. Maximum load for *Balance* and *Chop* as β varies.

ter 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%).

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.) Fig. 9 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.)

Fig. 10 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 be missed. Looking at Fig. 9, 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 Fig. 9, 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 $MD^{opt+0.05}_{Chop}$ in Fig. 9. 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 6.)

Another way to study the trade-off 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. Fig. 11 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 op-

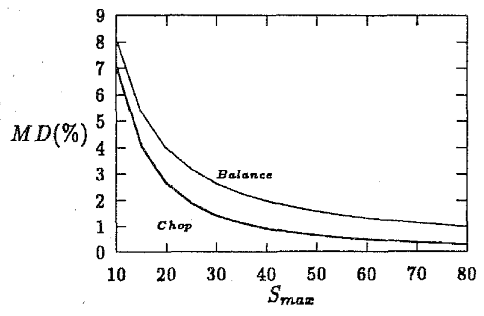


Fig. 12. $MD_{Balance,uniform}$, $MD^{opt}_{Chop,uniform}$ as S_{max} varies.

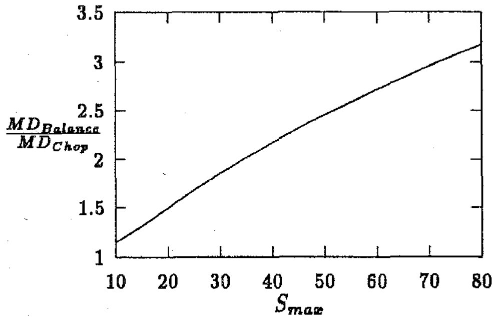


Fig. 13. $MD_{Balance,uniform}$ / $MD^{opt}_{Chop,uniform}$ as S_{max} varies.

timal.) 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. Figs. 12 and 13 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*.

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 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[6]: One for a dual-Ethernet system (Section 5.1) and another for a dual-disk system (Section 5.2). 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[7] [34]. 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 see how this strategy fairs when applied to dual-Ethernet and dual-disk systems.

5.1 Ethernet Simulation

We simulate a dual-Ethernet system with n nodes (or stations) (see Fig. 14). 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 [21]. 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 transmission time[8] (i.e., 0.04 time unit).

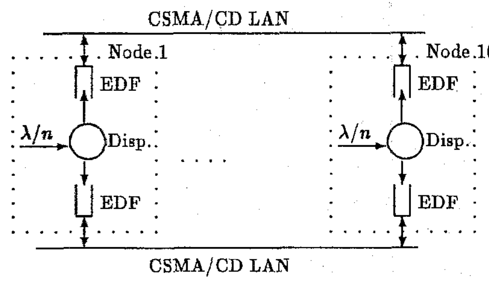


Fig. 14. Ethernet simulation model. (EDF—Earliest Deadline First queue)

Each simulation run lasts 200,000 time units. For example, with a packet arrival rate of λ = 1.0, this means about 200,000 packets are processed per run. Each data point is obtained by averaging the results of three simulation runs, each with a different random number seed. The 95% confidence interval is ± 0.13 percentage point. Table 1 shows the setting of our Ethernet baseline experiment.

TABLE 1
BASELINE SETTING FOR ETHERNET SIMULATION

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

6. Our simulators are written in the simulation language DeNet [19].

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

8. Based on a 512 bit contention slot time and a 1.5kB packet size (plus overhead) [34].

The solid lines in Fig. 15 show the simulation results comparing the *Balance* and *Chop* strategies as p varies for the baseline setting. The curves in Fig. 7 are reproduced in Fig. 15 as dotted lines. These dotted lines show the results as predicted by the analytical model.

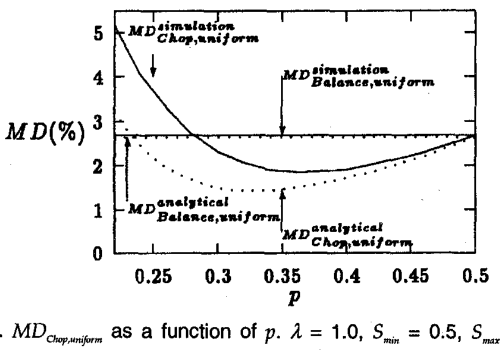


Fig. 15. $MD_{Chop,uniform}$ as a function of p. $\lambda = 1.0$, $S_{min} = 0.5$, $S_{max} = 30$, $n = 10$. Ethernet simulation.

Comparing the simulation results (solid lines) and the analytical results (dotted lines), we observe that the analytical model underestimates the missed deadlines *MD* when Ethernets (Fig. 14) rather than FCFS servers with global queues (Fig. 1) 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, Fig. 15 shows that both the simulation and analytical results display similar *trends*. In particular,

1) *Chop* performs better than *Balance* over a significant range of p values,
2) 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
3) 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 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.

5.2 Disk Simulation

Our next experiment simulates a dual-disk system (see Fig. 16). We make the following assumptions concerning the physical disks and the I/O requests.

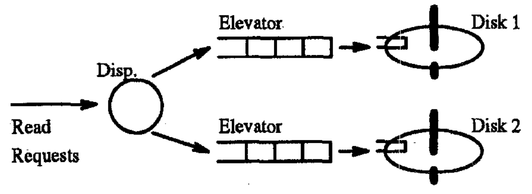


Fig. 16. Disk simulation model.

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 paper, 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.[9] 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 [2].

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

9. A write request is usually performed first on a local main memory copy of the data item [16]. 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.

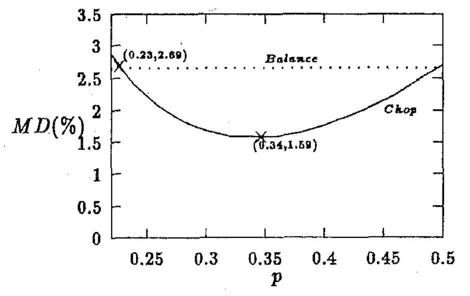


Fig. 17. MD_{Chop} as a function of p. λ = 43 req./sec., S_{min} = 36ms, S_{max} = 326ms. Disk simulation.

[5]. 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 [5].

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

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 Poisson distribution at a rate of λ jobs per seconds. We model disk requests at the track level. Rotational optimization is not considered in this paper. 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[10] of requests are computed according to the following formula:

$$Deadline = Arrival\_Time + Uniform([S_{min}, S_{max}])^{11}$$

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 2 shows the parameter setting of our disk baseline experiment. Again, simulation results are obtained by averaging the statistics of three 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 ± 0.13 percentage point.

TABLE 2
BASELINE SETTING FOR DISK SIMULATION

MaxTrack	λ	S_{min}	S_{max}	*DiskFactor*	*DiskConstant*
1,000	43 req./sec.	36ms	326ms	0.6	15ms

Fig. 17 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} 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 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 Fig. 7), Ethernet (2.69% in Fig. 15), and disk (2.69% in Fig. 17). 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 (Fig. 15) 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. Figs. 18a and 18b compare the miss deadlines percentage of *Balance* and *Chop* in the Ethernet and disk models, respectively. From Fig. 18a, 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 Fig. 18b. The *Chop* strategy is seen to blend well with the Elevator algorithm even when the system load is high.

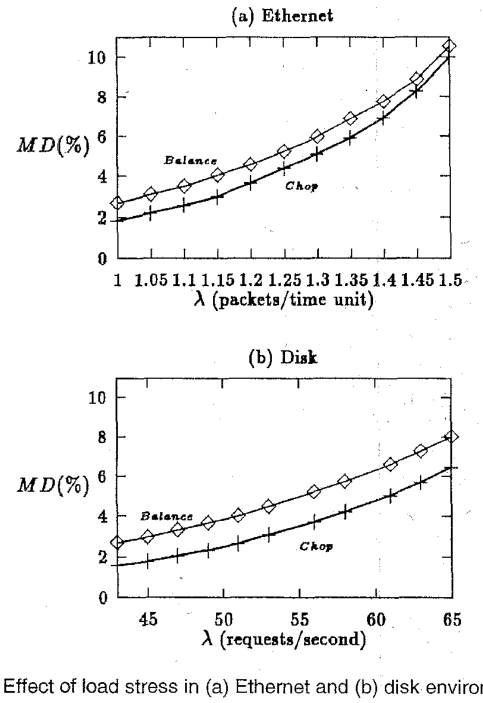


Fig. 18. Effect of load stress in (a) Ethernet and (b) disk environments.

10. 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.

11. Note that the execution time does not appear in the deadline formula. It is because the amount of time it takes for 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.

In this section, we showed some representative simulation results of Ethernet and disk schedulings and compared them to that obtained by the analytical model. Though not shown in this paper, 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.

6 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, Fig. 7) 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 describe one possible implementation of this dynamic tuning on a dual-Ethernet system. Due to space limitations, our description is brief.

As explained in Section 4, 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 coordinator 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.[12] 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 (4)), a local optimum is also the global optimum [26]. 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 4, 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 Fig. 14 with an exponential distribution for transmission time of mean 1 msec. The coordinator 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. Fig. 19 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 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.

12. If the coordinator 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*.

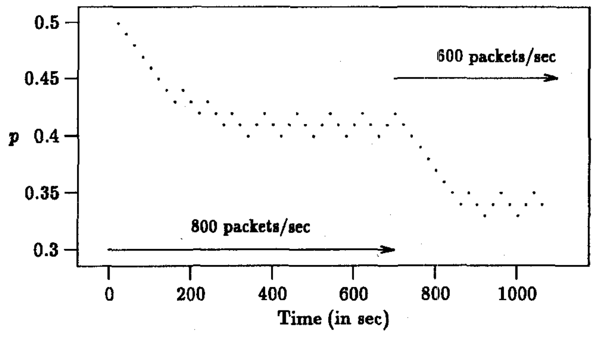


Fig. 19. p adopted by nodes.

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.

7 OTHER VARIATIONS OF THE BASE MODEL

Although the simulations of Section 5 show the performance of the *Balance* and *Chop* strategies in specific scenarios, we still continued to make three fundamental assumptions:

1) there were only two servers,
2) tasks were aperiodic, and
3) 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.

7.1 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 Fig. 20). The load to $Server_i$ is $\lambda \cdot p_i$. We can use (2) to compute the fraction of missed deadlines at each server. Since the p_is 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_is are then determined by locating the minimum of $MD_{ns,uniform}()$.

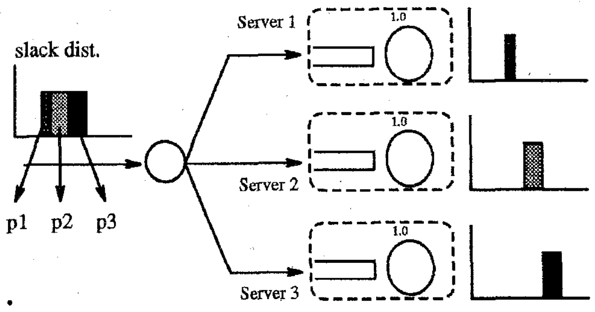


Fig. 20. n servers.

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_is 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 $\lfloor n/2 \rfloor$ 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 three-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_is 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 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 subdivide 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.

Fig. 21 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].

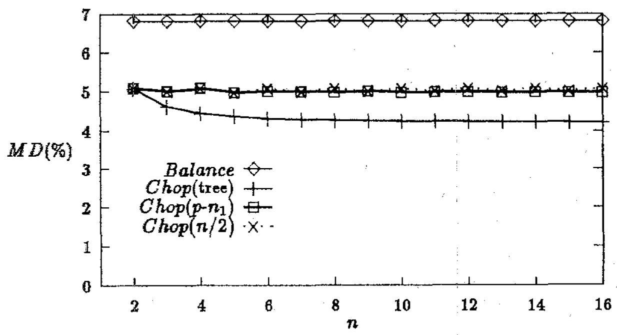


Fig. 21. Performance of *Chop* with different approximation methods.

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.

7.2 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. Due to space limitations, 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 ($1 \le i \le 10$) has a period of $1.5 + 2.95i$.[13]

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

Note that tasks with shorter period contribute more load to the system than tasks with 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.

Fig. 22 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 nine 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).

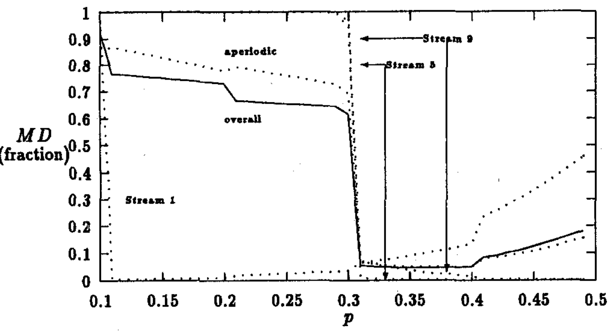


Fig. 22. Simulation result with periodic tasks.

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.)

7.3 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 paper, 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 Fig. 10). 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.

8 Conclusion

In this paper, 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 Fig. 13) 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.

REFERENCES

[1] Tandom Computers, *Introduction to NonStop Systems, 16270.* Cupertino, Calif., 1991.

[2] R. Abbott and H. Garcia-Molina, "Scheduling I/O requests with deadlines: A performance evaluation," *IEEE Real-Time System Symp.*, pp. 113-124, 1990.

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

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

[5] D. Bitton and J. Gray, "Disk Shadowing," *Proc. 14th VLDB Conf.*, pp. 331-338, 1988.

[6] M.J. Carey, R. Jauhari, and M. Livny, "Priority in DBMS resource scheduling," *Proc. 15th VLDB Conf.*, pp. 397-410, 1989.

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

[8] H. Garcia-Molina, B. Kao, and D. Barbara, "Aggressive transmission over redundant paths," *Proc. 11th IEEE Int'l Conf. Distributed Computing Systems*, pp. 198-207, 1991.

[9] J. Gray, B. Horst, and M. Walker, "Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput," *Proc. 16th VLDB Conf.*, pp. 148-161, 1990.

[10] J. Gray and A. Reuter, *Trans. Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[11] 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*, vol. 38, no. 12, pp. 1,736-1,744, Dec. 1989.

[12] I. Iliadis and L.Y Lien, "Resequencing delay for a queueing system with two heterogeneous servers under a threshold-type scheduling," *IEEE Trans. Comm.*, vol. 36, no. 6, pp. 692-702, June 1988.

[13] E.D. Jensen, C.D. Locke, and H. Tokuda, "A time-value driven scheduling model for real-time operating systems," *IEEE Real-Time System Symp.*, 1985.

[14] L. Kleinrock, *Queueing Systems*, vol. 1. John Wiley & Sons, 1976.

[15] C. Knessl, B.J. Matkowsky, Z. Schuss, and C. Tier, "Two parallel queues with dynamic routing," *IEEE Trans. Comm.*, vol. 34, no. 12, pp. 1,170-1,175, , Dec. 1986.

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

[17] W. Lin and P.R. Kumar, "Optimal control of a queueing system with two heterogeneous servers," *IEEE Trans. Comm.*, vol. 29, no. 8, pp. 696-703, Aug. 1984.

[18] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms, "A snapshot differential refresh algorithm," *Proc. ACM SIGMOD*, pp. 53-60, 1986.

[19] M. Livny, **DeNet** user's guide, technical report, Univ. of Wisconsin-Madison, 1990.

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

[21] R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM*, vol. 19, no. 7, pp. 395-404, July 1976.

[22] M.L. Molle and L. Kleinrock, "Virtual time CSMA: Why two clocks are better than one," *IEEE Trans. Comm.*, vol. 33, no. 9, Sept. 1985.

[23] M. Nakamura, I. Sasase, and S. Mori, "Two parallel queues with dynamic routing under a threshold-type scheduling," *IEEE GLOBECOM*, pp. 1,445-1,449, 1989.

[24] R.M. Newman and J.L. Hullet, "Distributed queueing: A fast and efficient packet access protocol for QPSX," *Proc. 8th Int'l Conf. Computer Comm.*, pp. 294-299, 1986.

[25] A. Orda and R. Rom, "Routing with packet duplication and elimination in computer networks," *IEEE Trans. Comm.*, vol. 36, no. 7, pp. 860-866, July 1988.

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

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

[28] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," *13th Int'l Conf. Distributed Computing Systems*, pp. 108-115, 1990.

[29] K. Ramamritham, J.A. Stankovic, and P. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184-194, 1990.

[30] P. Ramanathan and K.G. Shin, "A multiple copy approach for delivering messages under deadline constraints," *Proc. FTCS-21*, pp. 300-307, 1991.

[31] I. Sasase and S. Mori, "Analysis of queueing systems with multiple servers under a threshold-type scheduling," *IEEE GLOBECOM*, pp. 1,450-1,454, 1989.

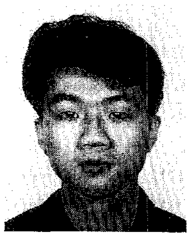
[32] L. Sha and J.P. Lehoczky, Performance of real-time bus scheduling algorithms," *ACM Performance Evaluation Rev.*, special issue, vol. 14, no. 1, pp. 44-55, 1986.

[33] J. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Trans. Computers*, vol. 34, no. 12, pp. 1,130-1,143, 1985.

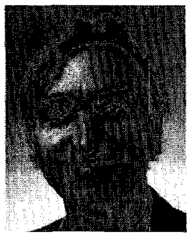
[34] A.S. Tanenbaum, *Computer Networks*. Prentice Hall, 1988.

[35] Y.C. Tay, "A behavioral analysis of scheduling by earliest deadline," Technical Report No. 532, Dept. of Mathematics, National Univ. of Singapore, 1992.

[36] F. Wang, K. Ramamritham, and J. Stankovic, "Bounds on the performance of heuristic algorithm for multiprocessor scheduling of hard real-time tasks," *IEEE Real-Time Systems Symp.*, pp. 136-145, 1992.



Benjamin Kao received the BS degree from the University of Hong Kong in 1989 and the MS degree from Princeton University, Princeton, N.J., in 1991; both degrees were in computer science. He was a teaching and research assistant at Princeton University from 1989 to 1991. Since 1992, he has been a research fellow in the Computer Science Department at Stanford University. His research interests include database management, distributed algorithms, real-time systems, and information retrieval systems.



Hector Garcia-Molina received a BS in electrical engineering from Instituto Tecnologico de Monterrey, Mexico, in 1974; an MS in electrical engineering from Stanford University in 1975; and a PhD in computer science from Stanford in 1979. From 1979 to 1991, he was on the faculty of the Computer Science Department at Princeton University. Dr. Molina is currently a professor in the Department of Computer Science and the Department of Electrical Engineering at Stanford University. His research interests include distributed computing systems and database systems. He is a member of the ACM and the IEEE.