The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control

Steven L. Scott, *Student Member, IEEE,* and Gurindar S. Sohi, *Member, IEEE*

Abstract—**In this paper, we propose the use of feedback control schemes in multiprocessor systems. In a multiprocessor, individual processors do not have complete control over, nor information about, the overall state of the system. The potential exists, then, for the processors to unknowingly interact in such a way as to degrade the performance of the system. An example of this is the problem of tree saturation caused by hot spot accesses in multiprocessors using multistage interconnection networks. Tree saturation degrades the performance of all processors in the system, including those not participating in the hot spot activity.**

We see that feedback schemes can be used to control tree saturation, reducing degradation to memory requests that are not to the hot spot, and thereby increasing overall system performance. As a companion to feedback schemes, damping schemes are also considered. Simulation studies presented in this paper show that feedback schemes can improve overall system performance significantly in many cases and with relatively little hardware cost. Damping schemes in conjunction with feedback are shown to further improve system performance.

Index Terms—**Damping, feedback, hot spots, multistage interconnection networks, tree saturation control.**

I. Introduction

ONE of the most important and widely used concepts in the design of engineering control systems is the concept of *feedback* [3]. Feedback is primarily used to prevent instability in a system and keep system outputs within a desired range. Fig. 1 illustrates how feedback works. Without feedback [Fig. 1(a)], the inputs of the system are independent of events that might be occurring in the system. Consequently, an unstable situation could arise in which system outputs take on undesired values. This is referred to as an open loop system. With feedback [Fig. 1(b)], the outputs of the system (and possibly other state values of the system) are fed back to the input generator. Based upon the feedback information, the system input generator tries to modify the system inputs to keep the system outputs at some set of desired values or within some set of desired ranges. This is referred to as a closed loop system.

Modern computing systems have evolved into large-scale parallel processors that consist of possibly hundreds of processors and memory modules interconnected together in some fashion. Fig. 2 illustrates a typical processing system based on a shared memory programming paradigm [1], [16]. The processing system consists of a set of processing elements, a set of memory modules, and an interconnection network. The interconnection network is logically broken into a forward network and a reverse network though it is possible that the two networks could be the same physical network (for example, a set of buses). It is important to realize that the overall performance of such a processing system is not determined solely by the performance of the individual components; it is affected by how the components interact dynamically.

Let us compare Figs. 1 and 2. If we assume that the forward interconnection network is the system, then the inputs to the system are the requests generated by the processors and the outputs of the system are the requests arriving at the memory modules. Let us consider, in particular, the *arrival rates* of the requests at the memory modules. The desired value for the arrival rate of requests to a memory module is the minimum of the rate at which the requests can be accepted by the memory module, and the aggregate access rate desired by the processors. If the arrival rate is lower than that desired by the processors, and lower than that which can be accepted by the memory module, then we are using system resources poorly by wasting memory module cycles. If the arrival rate is higher than that at which the requests can be accepted, then blockage occurs within the network. This blockage unnecessarily reduces the effective bandwidth of the entire system (interconnection network), as will be discussed in Section II.

For performance reasons, then, we want to maintain the desired arrival rates at all memory modules, a task that depends largely upon the input streams from the processors. Considering this dependence, and the resemblance between Figs. 1 and 2, we ask ourselves: 1) why has explicit feedback not been used thus far in the design of computing systems and 2) why might it be useful now?

The answers lie in the distributed nature of modern parallel processing systems. Traditionally, a computing system consisted of a single processor (system input generator). This processor generated requests to memory (there could be either a single or multiple outstanding requests) and had to wait for results back from the memory. This implicit feedback (waiting for the memory responses) sufficed to regulate the arrival rate of requests at the memory. In a multiprocessor system, however, many processors are generating requests without knowledge of the state of other components in the system. In such a processing system, it is possible that the collective input of the processors could interact in such a way as to cause unnecessary degradation of the network. This can occur if the arrival rate at any memory or intermediate point within the network is too high. The implicit feedback to processors (via memory responses) is not sufficient, as we shall see in Section IV, to convey enough information to correct the anomalous behavior. To provide enough information to correct the behavior, explicit feedback mechanisms may be warranted.

One could alter the processing system of Fig. 2 to resemble the system of Fig. 1(b) by providing explicit feedback from

Manuscript received September 8, 1989; revised March 14, 1990. S. L. Scott was supported by fellowships from the Wisconsin Alumni Research Foundation and the Fannie and John Hertz Foundation. G. S. Sohi was supported in part by National Science Foundation Grant CCR-8706722. A preliminary version of this paper appeared in the 16th International Symposium on Computer Architecture, Jerusalem, Israel, June 1989.

The authors are with the Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706.

IEEE Log Number 9036058.

1045-9219/90/1000-0385$01.00

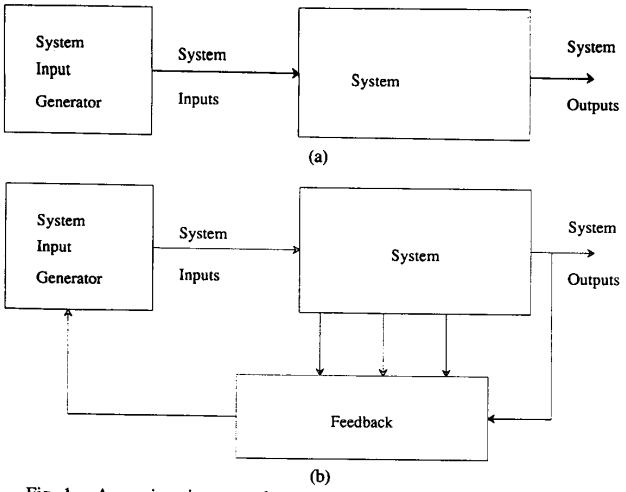


Fig. 1. An engineering control system. (a) Open-loop system. (b) Closed-loop (feedback) system.

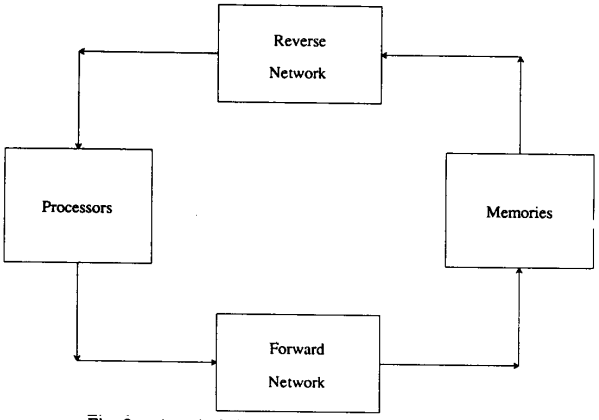


Fig. 2. A typical shared memory multiprocessing system.

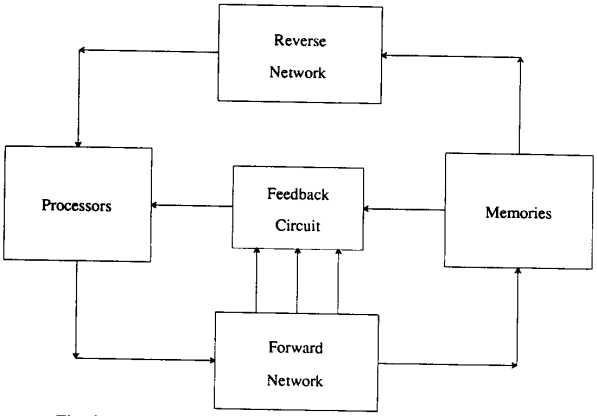


Fig. 3. A shared memory multiprocessing system with feedback.

points within the forward interconnection network to the processors (see Fig. 3). This explicit feedback could then be used to detect congestion at points within the network and instruct the processors to modify their inputs to the network such that the congestion is relieved. By reducing congestion and blockages within the network, the overall performance of the system could be enhanced.

In this paper, we illustrate the use of feedback in computer systems design by targeting the problem of *tree saturation* in parallel computer systems that use buffered multistage interconnection networks (MIN's). Tree saturation is a special form of network congestion that can severely degrade the performance of MIN's [14]. We demonstrate how feedback concepts can be used to control tree saturation and reduce the performance degradation that it causes.

The remainder of this paper is as follows. In Section II, we consider the undesirable situation of tree saturation in multistage interconnection networks that use a distributed routing control. We point out that if tree saturation could be controlled, the overall bandwidth of the network (and consequently the throughput of the multiprocessor) could be improved in many cases. In Section III, we propose schemes for controlling tree saturation. In Section IV, we present the results of a simulation analysis carried out to test the effectiveness of the tree-saturation-controlling mechanisms. In Section V, we present a discussion of the feedback concept in light of the results of Section IV, and in Section VI we present concluding remarks.

II. Tree Saturation in Multistage Interconnection Networks

A widely proposed interconnection network for medium to large scale multiprocessors is a blocking, buffered $O(N \log N)$ multistage interconnection network with distributed routing control. An example of such a MIN is the Omega network [11]. An Omega network consists of $\log N$ stages of switching elements (or switches). Memory requests enter the network at the inputs to the first stage and proceed to the outputs of the last stage, one stage at a time. Routing decisions are made local to each switch. Since there is no global control mechanism, the state of any particular switch is unknown to other entities (processors, memories, other switches) in the multiprocessor, and a particular input request pattern to the network might cause the request arrival rate at some point to exceed the capacity with which that point can service requests.

A. Discussion of the Problem

The problem of *tree saturation* is the direct result of an arrival rate being too high. This problem was first observed by Pfister and Norton in conjunction with requests to a *hot spot* [14]. In their analysis, the hot spot was caused by accesses to a shared synchronization variable. When the average request rate to a *hot memory module* exceeds the rate at which the module services the requests, the requests will back up in the switch which connects to the hot module. When the queue in this switch is full, it will back up the queues in the switches that feed it. In turn, those switches will back up the switches feeding them. Eventually, a tree of saturated switches results. Depending upon the number of outstanding requests and the reference patterns of the various processors, this tree of saturated queues may extend all the way back to every processor. Any request which must pass through any of the switches in the saturated tree, whether to a hot module or not, must wait until the saturated queues are drained. Thus, even requests whose destinations are idle will be blocked for potentially long periods of time, leading to unnecessary degradation of the network bandwidth and increase in the latency through the network.

Since the problem of tree saturation (caused by hot spot activity or otherwise) can be catastrophic to the performance of systems such as the NYU Ultracomputer [6], Cedar [8], and the IBM RP3 [15], all of which use buffered MIN's, considerable effort has been devoted to studying the problem and suggesting solutions to it [1], [6], [12], [14], [18].

When the problem is caused by accesses to synchronization variables (or more generally, by accesses to the same memory location), *combining* can be used. *Hardware combining* uses special hardware in the network switches to combine requests destined to the same memory location [1], [6]. On the return trip, the response for the combined request is broken up (decombined) into responses for the individual requests. It is estimated that using combining hardware in the network switches would increase the hardware cost of a multistage interconnection network by a factor of 6 to 32 [14]. *Software combining* uses a tree of variables to effectively spread out access to a single, heavily-accessed variable [5], [18]. It is applicable only to known hot spot locations such as variables used for locking, barrier synchronization, or pointers to shared queues.

Since the overall bandwidth of the network is determined by the number (or equivalently the rate) of requests that have to be serviced by the hot module, combining can improve overall network bandwidth by reducing the rate at which requests are submitted to the hot module. Combining also improves the latency of memory requests that do not access the hot memory module since it alleviates tree saturation. Unfortunately, combining cannot alleviate the bandwidth degration or the tree saturation if the hot requests are to different memory locations within same memory module, that is, the entire memory module is hot. Such a situation could arise from a larger percentage of shared variables residing in a particular module, stride accesses that result in the nonuniform access of the memory modules, or temporal swings in which variables stored in a particular module are accessed more heavily. In these cases, one module will receive more requests than its uniform share, just as if it contained a single hot variable. Recognizing this problem, the RP3 researchers have suggested scrambling the memory to distribute memory locations randomly across the memory modules [2], [13]. With a scrambled distribution it is hoped that nonuniformities will occur less often though we are unaware of any hard data to support this fact.

Even though processor requests may be distributed uniformly among the memory modules, tree saturation can still occur if any of the switches in the network have a higher load (in the short term) than other switches at the same stage [10]. Alternate queue designs may improve the latency of memory requests that do not access the hot module, in the short term, but eventually tree saturation will occur even with alternate queue designs [17].

To alleviate the problem of tree saturation in general, we need a mechanism that detects the possibility of tree saturation and instructs the processors[1] to hold requests that might contribute to the tree saturation. If many of the problem-causing requests can be held outside the network, the severity of the problem can be reduced.

Before proceeding further, let us convince the reader that alleviating the congestion caused by tree saturation can indeed result in an increase in the overall performance of the system. We shall only consider hot requests that cannot be combined in this paper, since no solution to the problem of tree saturation is known in this case. We shall also restrict ourselves to $N \times N$ Omega networks though our results can easily be extended to other MIN's.

B. Bandwidth Degradation Due to Tree Saturation

Consider a processing situation in which a fraction f of the processors (the *hot processors*) are making requests to a hot module with a probability of h on top of a background of uniform requests to all memory modules, and the remaining processors (the *cold processors*) are making only uniform requests. Processors may have multiple outstanding requests. This is a likely scenario if more than one job is run on the multiprocessor. Let r_1 be the rate (requests per network cycle) at which the hot processors generate requests and let r_2 be the rate at which the cold processors generate requests. The number of requests per cycle that appear at the hot module is therefore

$$R_{\text{hot}} = fNr_1\left(h + \frac{(1-h)}{N}\right) + (1-f)Nr_2\left(\frac{1}{N}\right)$$
$$= fr_1(hN + (1-h)) + (1-f)r_2. \quad (1)$$

Since the hot module can service only one request in each memory cycle, the maximum value of R_{hot} is 1. Constraining the right-hand side of (1) to be less than or equal to 1 and rearranging terms, we get

$$r_1 \le \frac{1-(1-f)r_2}{f(1+h(N-1))}. \quad (2)$$

To calculate the overall bandwidth of the network, we observe that the fN hot processors require a bandwidth of r_1 and the $(1-f)N$ cold processors require a bandwidth of r_2. Therefore, the average bandwidth per processor is constrained by

$$\text{BW} = \frac{fNr_1 + (1-f)Nr_2}{N}$$
$$\le \frac{1 + (1-f)h(N-1)r_2}{1+h(N-1)}. \quad (3)$$

[1] For the remainder of this paper, the words "processors" and "processors network interfaces (PNI's)" will be used interchangeably. For feedback schemes, the entities responding to the feedback information are actually the PNI's though for explanation purposes it is easier to talk of the "processors" using the feedback information to alter the request pattern into the network.

If there were no tree saturation in the network and the cold processors could proceed without *any* interference at all, they could achieve a *best-case* throughput of 1 request per cycle, i.e., $r_2 = 1$. Therefore, the best-case (or cutoff) per-processor bandwidth of the system would be

$$\mathrm{BW}_{\mathrm{cut}} = \frac{1 + (1-f)h(N-1)}{1 + h(N-1)} = 1 - \frac{fh(N-1)}{1 + h(N-1)}. \tag{4}$$

Unfortunately, tree saturation prevents the cold processors from proceeding without interference and achieving a throughput r_2 that is close to the best case. With tree saturation, cold processors as well as hot processors suffer degraded service. It does not matter which processors are responsible for the tree saturation, *all* processors are affected by it. Empirically, we have observed that in this case the system behaves as if all processors were participating in the hot spot activity. The rate at which hot requests are issued when fN processors have a hot rate of h is equal to the rate when all N processors have a smaller hot rate of fh. Thus, when hot requests are permitted to cause tree saturation, we have observed that the average cutoff bandwidth per processor can be estimated by the standard formula for a hot rate of fh:

$$\mathrm{BW}_{\mathrm{cut}} = \frac{1}{1 + fh(N-1)}. \tag{5}$$

Since (4) estimates the bandwidth of the system when the cold processors are not degraded by tree saturation and (5) estimates the bandwidth when they are, we can estimate the bandwidth improvement (if tree saturation is controlled and the cold processors allowed to proceed without any interference) by comparing (4) and (5).

Fig. 4 plots the bandwidths suggested by (4) and (5) as a function of f, for $h = 4\%$, and $N = 256$. These equations represent upper limits only, and in particular, for very small f, normal traffic contention will limit the bandwidth per processor to below 1. Simulation results presented in Section IV, however, confirm that the bandwidth degradation due to tree saturation (Fig. 6) closely corresponds to that predicted in (5), and that the relationship shown in Fig. 4(b) also occurs.

As can be seen from Fig. 4, it appears that the overall bandwidth of the network can be improved significantly if the problem of tree saturation is alleviated, allowing the cold processors to access the network without interference caused by the tree saturation. The bandwidth improvement is zero at the endpoints, i.e., all processors are making uniform requests ($f = 0$) and all processors making hot spot requests ($f = 1$), and the greatest when only half of the processors are making hot requests ($f = 1/2$). Our experimental results in Section IV will confirm this.

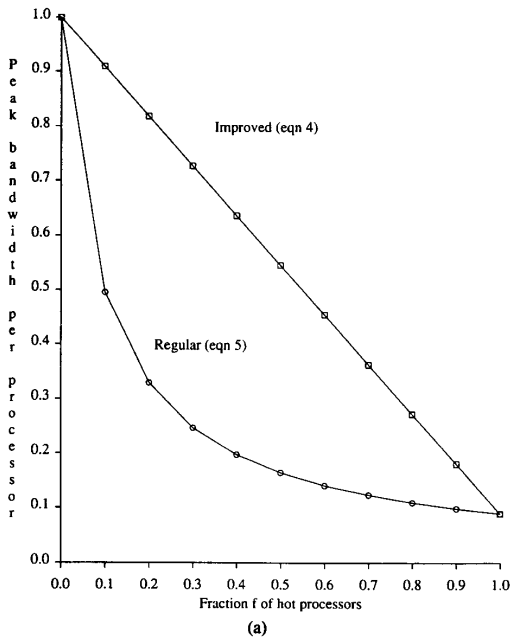


Fig. 4. Estimated peak bandwidth per processor with and without tree saturation control. (a) Estimated peak bandwidth per processor from (4) and (5). (b) Estimated bandwidth improvement [(4) relative to (5)].

III. Controlling Tree Saturation

To alleviate the problem of tree saturation, requests that compound the problem must be prevented from entering the network until the problem has subsided. Ideally, requests to a hot module must be made to wait outside the network, at the processor–network interface (PNI), until the hot module is ready (or slightly before it is ready) to service them, and then enter the network at a rate at which they can be serviced by the hot module. If this can be done effectively, then tree saturation will be eliminated, and system throughput should increase as shown in Fig. 4. In practice, it is unlikely that tree saturation can be completely eliminated with no side effects, but the closer we come to this goal, the better we can expect our performance to be. In Section III-A, we present a basic feedback scheme that attempts to achieve this goal, and in Section III-B we present extensions to this scheme that significantly improve its effectiveness.

A. The Basic Feedback Scheme

Fig. 3 presents a multiprocessing system with feedback. Select state information is tapped from the MIN and the memory modules and fed back to the PNI's. The PNI's use this information to modify their collective input to the network when it is causing state values of the system (in our case, the arrival rate of requests at the memory modules) to exceed their desired values. The feedback information therefore must indicate to the processors when such a performance-degrading situation is occurring.

Since tree saturation is caused by the arrival rate at the memory modules exceeding the rate at which the module can service requests, the difference between the arrival rate and module service rate is the state information that needs to be fed back to the processors. Unfortunately, the arrival rate by itself might be hard to measure. However, a good indication that the arrival rate at a module is too high, and that tree saturation might develop, is that the length of the queue feeding the module begins to grow. This queue length, then, would be a good candidate for use as state information to be fed back to the processors. As the queue length for a particular memory module grew, the PNI's could respond by reducing the rate at which requests to that module are submitted into the network. Tree saturation will not occur if the queues can be kept from overflowing.

The desired range for the length of a memory module queue is between one (zero if there are no outstanding requests for that memory module) and the capacity of the buffer used to contain the queue. If the queue becomes empty while the memory module is idle, and there are outstanding requests for that module, then we are making inefficient use of the memory module. If the queue overflows its capacity, then tree saturation begins to develop and can interfere with requests to other memory modules.

There is a considerable amount of freedom available in the feedback policy chosen to keep the queue lengths within the desired range. The scheme we have used in this paper is quite simple. We define a memory module to be hot if its queue length exceeds a certain threshold T, and define it to be cold otherwise. It is this binary value for each module that is fed back to the PNI's. The PNI's respond by allowing requests to cold modules to enter the network freely, while blocking requests to hot modules outside the network. This causes the arrival rate of requests at a hot module to drop sharply, shortly after the module becomes hot. By compensating for periods of high arrival rate with periods of low (or zero) arrival rate, it is hoped that average arrival rate will remain at its desired value and the queue length will remain within its desired range, thus avoiding the occurrence of tree saturation.

The choice of this simple feedback scheme allows for an implementation with very low hardware complexity. All that we need to do is monitor the size of the queue at each memory module and notify the processors when a module becomes hot (the size of the queue exceeds a predefined threshold T) and when a module becomes cold (the size of the queue falls below the threshold T). The most obvious way to do this is to have an N-bit feedback bus from the memory modules to all the processors. Memory module i sets the ith bit on the bus if it is hot. The processors can determine the status of each memory module simply by looking at the information on the bus.

As system size increases, distribution of a bus can become difficult. In this event, a fanout tree of buses can be used. This causes the latency of feedback to increase, but speed is not critical. It has been estimated [9] that the onset of tree saturation occurs as quickly as several network traversal times. Even using a fanout tree, feedback information can be supplied to the processors in less than a single network traversal time.

In addition, when the information content of the information being fed back to the PNI's is low, i.e., the number of modules that experience hot-cold or cold-hot transitions in a given cycle is low, alternate designs can be used for this feedback circuit. Preliminary experience shows that large scale parallel programs typically have only one or two hot spots at a time [7], [14]. In such cases, an N-bit feedback bus is clearly wasteful.

The feedback circuit that we use is a $\log_2 N + 2$ bit bus; $\log_2 N$ bits to indicate the module number and 2 control bits to indicate whether the module indicated by the $\log_2 N$ bits experienced a hot to cold, cold to hot, or no transition in the particular network cycle. As before, this bus can be implemented as a tree, if necessary. Since multiple transitions may take place in a single cycle, arbitration is required, and this may also be implemented as a tree for large systems. The hardware required for this feedback, while not trivial, is small in comparison to the MIN itself. In particular, due to the lenient speed requirements, the branching factor for a feedback tree would be quite large.

Each PNI maintains a lookup table in a fast memory. This table stores a single bit per memory module (indicating if the module is hot or cold), and is updated automatically from the transition messages on the feedback bus. When submitting a request into the network, the PNI checks the status of the request's destination module in the table and submits the request only if the destination module is cold. When a request is being held back at the PNI, the processor is stopped from making additional requests.

The feedback scheme outlined above prevents a hot module from causing full tree saturation because, as soon as the module becomes hot, requests to that module are stopped from entering the network. However, it has some problems analogous to the *undershoot* and *overshoot* problems in classical control systems theory [3].

First, let us examine undershoot and what might cause it. When a module becomes hot, requests in transit destined for that module continue while new requests destined for that module are held back. At some point in the future, the memory module will drain its queue beyond the threshold, the module will be declared cold again, and the requests to that module that were being held back in the PNI's will be released into the network. If the memory module consumes the rest of its queued requests before the newly released requests arrive, the module will become idle for one or more cycles. This is an especially inefficient use of resources given the fact that the hot module is a system bottleneck.

Now let us examine the more detrimental effect of overshoot in the simple feedback scheme. While a module is hot, requests to that module are held back at the PNI's. With a high hot rate, there may be requests waiting at the majority of the PNI's by the time the module becomes cold again. When the module becomes cold, all of the held back requests are released simultaneously. This can cause temporary overflow of the module's queue when the requests reach their destination, leading to partial tree saturation in the network.

While it has been estimated in the literature [9] that the onset of tree saturation occurs as quickly as several network traversal times (the time for a packet to traverse the network in one direction), the onset of this partial tree saturation can occur much more quickly due to the surge of traffic associated with overshoot. With no measures taken to prevent undershoot and

overshoot, we would expect a system to oscillate between periods of partial tree saturation, and periods in which the hot module becomes idle. Thus, we must address both of these problems in order to minimize performance degradation.

The problem of undershoot can be dealt with in a rather straightforward way. Recall that the problem occurs when the number of requests in a hot module's queue falls to below its threshold, the module is declared cold, and new requests to that module can be released into the network. If the module consumes the rest of its queued requests before the newly submitted requests arrive, then it will become idle temporarily. The solution is to inform the PNI's sooner that they may resume submitting requests. This can be done by setting the feedback threshold to a sufficiently large number such that newly released requests will arrive before, or shortly after, the queue has been completely drained. Our simulation results in Section IV will confirm the effectiveness of this approach.

Unfortunately, the problem of overshoot is not dealt with as easily. The response to the signal that a hot module has become cold with the simple feedback scheme is simply too strong; nothing can prevent blockage with the network once all the previously held back requests are released. To reduce the response to the hot-cold transition, some form of *damping* needs to be introduced.

B. Using Damping with Feedback

To prevent overshoot, we need a method that gradually releases held back requests into the network, rather than letting them all go at once on a hot-cold transition. By preventing the sudden surge of hot spot traffic associated with overshoot, the feedback mechanism would have more time to detect impending queue overflow and instruct the processors to modify their inputs. As a result, queue overflow and the resulting partial tree saturation could be significantly reduced or eliminated completely. We will present two damping schemes which attempt to do this. The first is an idealized and very strong form of damping called *limiting,* which we include for purposes of comparison. While performing very well, limiting would be prohibitively expensive to build in a real system. The second is a more practical scheme that uses adaptive backoff techniques to dynamically alter its behavior based upon the intensity of the hot spot.

1) Limiting Damping: Limiting uses global arbitration to limit the number of requests that enter the network each cycle for each memory module. When a module is cold, we allow a maximum of two requests destined for that module to enter the network in a given cycle. This limits the rate at which the memory module queue can grow, giving the feedback mechanism time to signal the processors if the queue reaches its threshold and avoid any significant queue overflow. When a module is hot, we allow a maximum of one request destined for that module to enter the network in a given cycle. Allowing a single request to enter even when the module is hot makes it very unlikely that the module's queue will become empty.

This form of damping is quite strong and requires a significant hardware overhead. N arbiters are required (one for each memory module), each of which must be capable of arbitrating between N processors on every cycle. Clearly such hardware would carry a prohibitive cost, but, as mentioned earlier, the scheme is included as an example of a very strong form of damping for comparison purposes.

It is worth noting that global limiting alone (with no feedback mechanism) can be used to prevent tree saturation, but only at the expense of decreasing overall system bandwidth for normal traffic. By limiting the number of requests allowed into the network for each memory module to a maximum of one per cycle, no tree saturation is allowed to occur. However, this unnecessarily limits throughput for uniform traffic. The reason for this is that without the feedback information, the global limiting must be blindly applied to all requests. Thus, requests issued in the same cycle to the same memory module are *always* constrained to enter the network one at a time. When the destination module is cold, however, this constraint is unnecessary, as multiple requests could enter the network in the same cycle and proceed to the final stage of the network without hindering requests to other memory modules. In the final stage, they could queue up and be serviced one at a time.

2) Adaptive Backoff Damping: The motivation for our adaptive backoff damping scheme comes from a need to regulate the release of hot requests, as in the limiting scheme discussed above, but without paying the price of global arbitration mechanisms. We need a purely distributed scheme that tells us how long to hold a given hot request in the PNI before releasing it into the network after its destination module becomes cold. This is difficult to do in general because the period of time over which we should release the requests depends upon the number of requests being held back at all PNI's at the instant the destination module becomes cold, which is information that individual PNI's do not have. The more requests that are being held back, the longer the period of time over which we must release the requests in order to prevent a surge of traffic that causes tree saturation.

Our approach is to dynamically adapt the length of this release period for each memory module according to the intensity of the hot spot at that module. We can determine whether the current release period for a module is too long or too short, by monitoring the module's *cold time.* The cold time for a module is the time for which the module remained cold the last time it changed from hot to cold. If the cold time was too small (it became hot again quickly), it indicates that the release period last used was too small. If the cold time was very large (it took a long time to become hot again), it indicates that the release period last used was too large. By adjusting the release period appropriately, the PNI's can independently adapt their release schedules to the request rate for each memory module, and thus avoid the surges of traffic associated with overshoot.

The implementation we have used to filter the release of requests into the network is to require a match of the low b bits of the source PNI address and the current time. If $b = 1$, then requests at half of the PNI's will be eligible for release on a given cycle. If $b = 4$, then requests at one sixteenth of the PNI's will be eligible for release on a given cycle. If $b = \log_2 N$, then only the request at a single PNI will be eligible for release on a given cycle. Thus, for a given value of b, all held back requests will be released over a period of 2^b cycles. This filtering is done only for requests being held back at the time a hot module becomes cold; requests to other modules are not affected, and subsequent requests to this module (which is now cold) proceed without delay.

The current value of b is saved for each memory module in the lookup table proposed earlier for saving the hot/cold states of each module. In addition, we save the last hot-cold transition time for each module so that the cold time can be computed when the module next becomes hot. The value of b is initially set to 0 for all PNI's, and is incremented by some appropriate value (either positive, negative, or zero) on each hot-cold

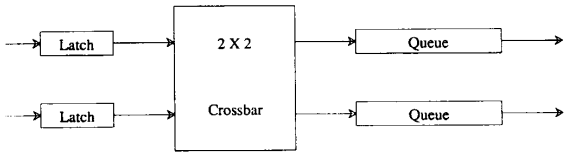


Fig. 5. Switch used in model and simulations.

transition, based upon the value of cold time relative to the network traversal time. This allows b to become properly adjusted within 2 or 3 hot-cold transitions.

The cost of adding this damping scheme to the basic feedback scheme is increased PNI complexity, and storage required to save the hot-cold transition times and values of b for each memory in each PNI. Since b can take on values from 0 to $\log_2 N$, we need $\log_2(\log_2 N)$ bits to represent a value of b (3 bits in a 256 processor system, 4 bits in a 64K processor system). To dynamically change b, we need to measure cold times up to a maximum of K network traversal times (in our simulations, K was 6). Thus, the number of bits needed for a hot-cold transition time is $\log_2(K * \log_2 N)$ bits plus 2 bits to handle wraparound (8 bits in a 256 processor system, 9 bits in a 64K processor system). If we liberally allow 2 bytes for storing a (hold-cold transition time, b) pair, then we require 512 bytes per processor in a 256 processor system, and 128K bytes per processor in a 64K processor system. While still a small cost in relation to the MIN, this might be expensive.

An alternative to storing the complete table is to cache the information for recently hot modules. While assuming a full table for our simulations, we believe that caching would work quite well. When a hot module becomes cold, either its entry resides in the caches, or it is placed in the caches with b set to 0 (which is what b should be if the module has been cold for a long time). On subsequent hot-cold transitions, the entry should reside in the caches. If the number of active hot modules is low, then we should see very little contention and excellent temporal locality in the caches, allowing the size to be quite small. Caching the full table is a topic for further study.

IV. Simulation Model and Results

A. Network and Workload Model

For all our experiments we considered an $N \times N$ Omega network connecting N processors to N memory modules. A forward network carries requests from the processors to the memories and a reverse network is used for responses from the memory modules to the processors.

A 2×2 crossbar switching element (shown in Fig. 5) is used as the basic building block. The size of the queue at each output is Q requests and each queue can accept a request from both inputs simultaneously if it has room for the requests. The order in which multiple inputs are gated to the same output is chosen randomly.

Requests move from one stage of the network to the next in a single network cycle. Each memory module can accept a single request every network cycle and the latency of each memory module is one network cycle. Therefore, the best-case roundtrip time for a processor request is $2\log_2 N + 2$ network cycles [issue (1) + forward network hops ($\log_2 N$) + memory module service (1) + reverse network hops ($\log_2 N$)].

The workload model that we use is the same as the one that has been widely used in the hot spot literature [9], [10], [12], [14], [17]. In each network cycle, a processor makes a request with a probability of r. A fraction f of the processors (the hot processors) make a fraction h of their requests to a hot memory module and the remaining $(1 - h)$ of their requests are distributed uniformly over all memory modules. The remaining fraction $(1 - f)$ of the processors (the cold processors) make uniform requests over all memory modules. A processor may have multiple outstanding requests, but is blocked from making further requests when its last request has not been accepted into the network by its PNI. This is due to the fact that there are no queues in the PNI's.

B. Simulation Results

The results presented in this section are for 256×256 ($N = 256$) Omega networks with queue sizes of 4 elements ($Q = 4$) at each switching element output, a main memory module latency of 1 network cycle, and no queues in the PNI's. We have also simulated 64×64, 128×128, and 512×512 Omega networks, each with varying queue sizes and memory latencies and with and without queues in the PNI's. The results follow a similar trend to the results we report for 256×256 networks with queue sizes of 4 and memory latencies of 1 and no queues in the PNI's, though the magnitude of the results are different. For reasons of brevity, we shall not present those results in this paper.

Five varieties of networks were simulated:

- regular Omega networks
- networks with simple feedback (threshold $T = 1$, 2, 3, and 4 queue elements)
- networks with feedback ($T = 4$) and adaptive backoff damping
- networks with feedback ($T = 1$) and limiting damping
- networks with straight limiting (one request per module per cycle).

The choice of ($T = 4$) for feedback with adaptive backoff damping is to minimize undershoot. With limiting damping, undershoot is not a problem, and ($T = 1$) is used to maximize control of tree saturation.

1) Bandwidth Improvement: Fig. 6 plots the saturation bandwidth per processor for a regular Omega network (without feedback). Several hot rates h are considered, and f is varied from 0 to 1. The saturation bandwidth is calculated by offering the network a load of one (each processor makes one request per cycle), and measuring the realized throughput of the network after cold start effects have subsided. From Fig. 6 we see that as the fraction of processors making hot requests increases, the overall system bandwidth decreases. The higher the hot rate h, the faster the bandwidth drops off. When all processors are making hot requests, the bandwidth is severely affected by the hot rate.

The purpose of feedback schemes and limiting schemes is to control tree saturation and consequently improve overall network bandwidth. At the endpoints of each curve in Fig. 6, i.e., $f = 0$ and $f = 1$, feedback is of little use in improving the bandwidth (but as we shall see in Section IV-B2, it can still improve memory latency). This is because when all processors are making uniform requests ($f = 0$), little tree saturation occurs, and when all processors are making hot requests ($f = 1$), the bandwidth is limited by the rate at which the hot module can service requests and not by the tree saturation that is present. Overall bandwidth of the network can be improved by controlling tree saturation only when the tree saturation is actually limiting the bandwidth, i.e., f is between 0 and 1.

Now we consider the use of our basic feedback scheme of

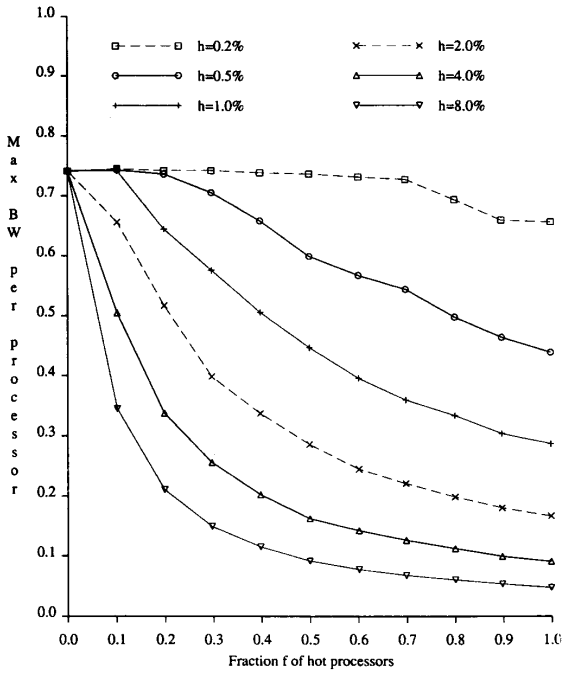


Fig. 6. Maximum bandwidth per processor with a regular Omega network ($N = 256$).

Section III-A. Fig. 7(a), (b), and (c) plots the bandwidth improvements for networks with feedback thresholds (T) of 1, 3, and 4, respectively. The bandwidth improvement is the bandwidth of the modified network (with feedback) divided by the bandwidth of a regular network with no feedback or limiting. A value of 1 for the improvement indicates that the two networks have the same bandwidth, a value greater than 1 indicates that the modified network has a higher bandwidth, and a value of less than 1 indicates that the modified network has a lower bandwidth.

From Fig. 7(a)–(c) we see that when f lies between 0 and 1, the use of feedback alleviates the tree saturation caused by the hot requests, allowing the processors making uniform requests to proceed with less interference, and increasing network bandwidth. The actual magnitude of the improvement is less than what is possible, since tree saturation has not been eliminated, but rather just reduced. Qualitatively, however, these simulation results confirm the predictions of Section II.

We can make two observations concerning the threshold values for feedback. The first observation is that under high hot rates, lower thresholds perform better than higher thresholds. This is due to the fact that lower thresholds prevent hot traffic from entering the network sooner and thus have less temporary hot module queue overflow and partial tree saturation. With a threshold $T = 1$, a hot module's queue can accept three more requests at the time it becomes hot, before overflowing and causing partial tree saturation. With a threshold, $T = 4$, the queue is already full by the time the module is declared hot and further requests to the module that are already in the network will cause partial tree saturation in earlier stages of the network.

The second and more striking observation is that using thresholds that are too small can limit the bandwidth of a network with feedback to *less* than the bandwidth of a regular network, for uniform and low hot-rate traffic. The smaller the threshold, the more likely a request is to be held back at the PNI even though the destination memory module of the request is receiving an average of less than one request per cycle. Networks with larger thresholds are less likely to restrict bandwidth unnecessarily due to temporal fluctuations in the traffic pattern. Another reason that smaller thresholds restrict bandwidth is that they allow the hot module's queue to become empty for longer periods of time (the undershoot problem, as discussed in Section III-A).

On closer look at Fig. 7(c), we see that even with a high feedback threshold ($T = 4$), the bandwidth is sometimes slightly less than the bandwidth of a regular network for low hot rates. This can be attributed to the problem of the hot module's queue occasionally becoming idle for a few cycles. When $f = 0$ (no hot spots) the relative bandwidth is unity. This indicates that normal traffic is not being restricted. If the queue sizes permitted a threshold equal to the number of stages in the network, the problem of a hot module's queues becoming idle could essentially be eliminated.

The higher the hot rate, the more the overall network bandwidth is improved by using even a simple feedback scheme. With a hot rate of 4 or 8%, significant increases in system bandwidth occur even with a small percentage of processors making hot requests. As systems become larger, the tree saturation caused by a given hot rate will become more severe, and the hot rate needed to cause a given level of tree saturation will decrease. In such cases, the need for feedback schemes is even more compelling.

Now let us consider the results of using the simple feedback scheme with an adaptive backoff damping strategy. Fig. 7(d) shows the relative bandwidth for a system which uses a high feedback threshold ($T = 4$), to minimize undershoot, and adaptive backoff damping to minimize overshoot. We see that the relative bandwidth does not fall appreciably below unity at any point, and the peak bandwidth improvement is significantly better than that obtained using simple feedback [Fig. 7(a)–(c)]. This demonstrates that the addition of damping to simple feedback is effective in reducing the amount of overshoot which occurs when a hot module becomes cold.

Now let us consider feedback with a limiting damping strategy. Since the damping mechanism prevents undershoot by allowing a single request for a module to enter each cycle when the module is hot, we can use a small feedback threshold ($T = 1$). Fig. 7(e) shows the relative bandwidth improvement when our strong form of limiting damping is used in conjunction with feedback. We can see that the results are quite good, controlling tree saturation even better than the adaptive backoff method. With 50% of the processors making requests with a hot rate of 8%, system bandwidth is improved by a factor of 4. It is worth noting here that since the feedback and limiting are not improving the bandwidth of the processors making hot requests (they cannot, since the bandwidth of the processors making hot requests is being limited by the number of requests to the hot module), and since the *average* bandwidth is increased by a factor of 4, then the bandwidth of the processors making uniform requests is actually being increased by a factor of 7.

Finally, Fig. 7(f) shows the use of limiting alone without any feedback information. We can see that performance is very good for high hot rates, but that bandwidth is significantly degraded for low hot rates. This agrees with the observations made in Section III-B. In the absence of feedback information, limiting must be applied to all references. While this prevents tree saturation from occurring, it also restricts normal traffic.

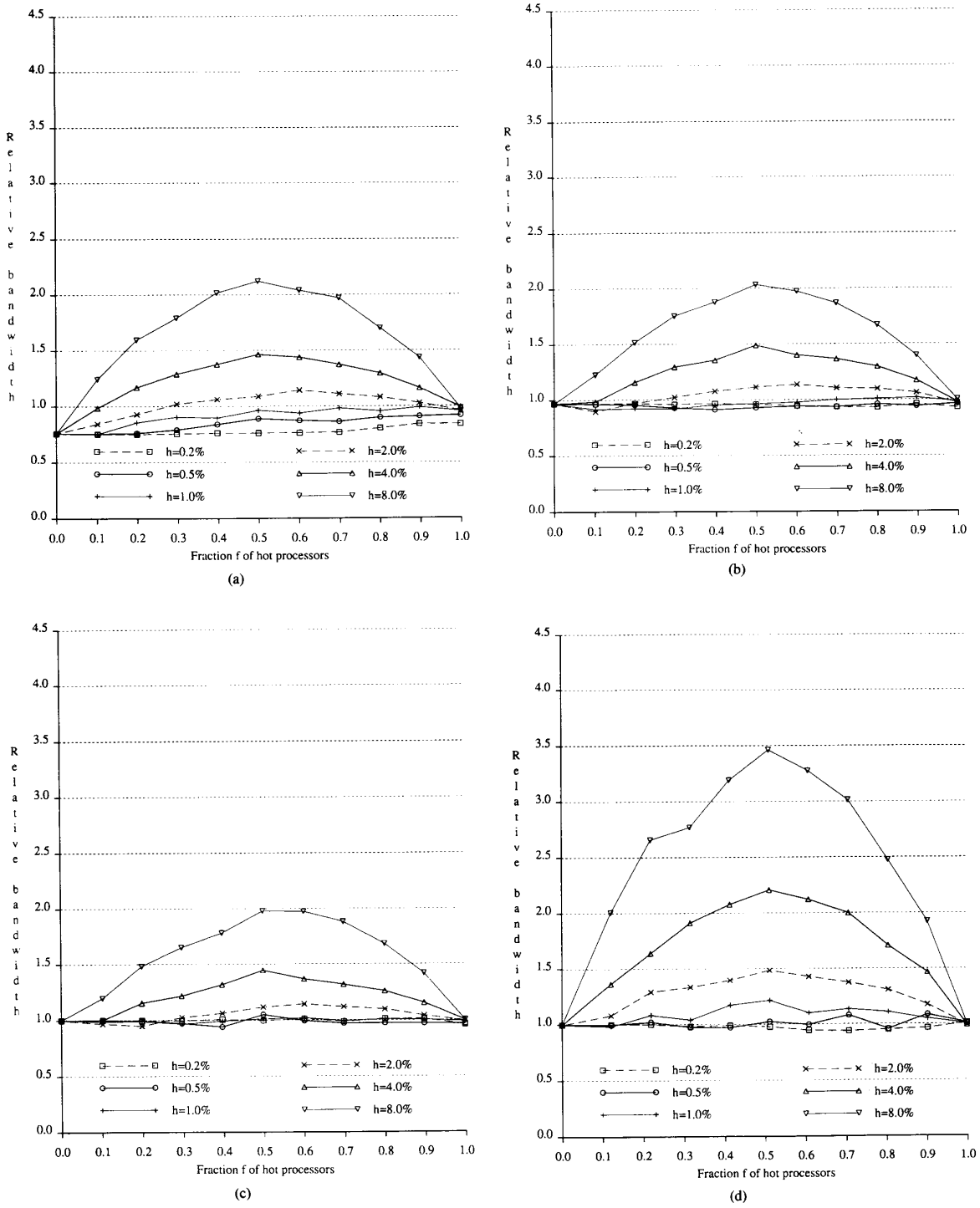


Fig. 7. Maximum bandwidth per processor with tree saturation control mechanisms, relative to a regular Omega network ($N = 256$). (a) Feedback, $T = 1$. (b) Feedback, $T = 3$. (c) Feedback, $T = 4$. (d) Feedback ($T = 4$) with adaptive backoff damping.

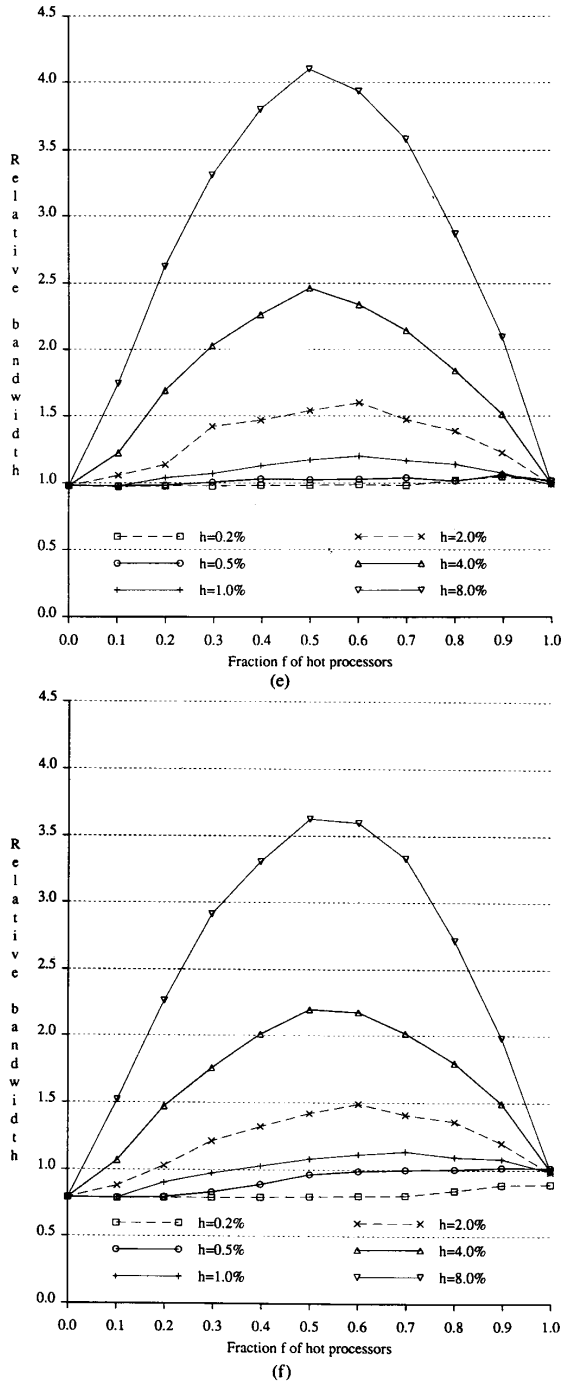


Fig. 7. (*continued*). (e) Feedback ($T = 1$) with limiting damping. (f) Plain limiting.

2) Latency Improvement: So far we have seen how bandwidth can be improved when some processors are making hot requests and the rest are making uniform requests. The improvement stems from reducing the tree saturation that blocks the processors not participating in the hot spot activity. However, for all schemes, no bandwidth improvement is obtained in the cases where all processors are making hot requests ($f = 1$). How can we be sure that tree saturation is being controlled even in this case (and in cases where little bandwidth improvement is obtained)? As we have noted before, the maximum bandwidth is inherently limited by the number of requests that must all go to the same module, and the only thing that can be done to improve the bandwidth is to cut down on the number of requests. However, the roundtrip latency experienced by the requests also gives us a good handle on the degree of tree saturation in the network. If tree saturation were being controlled, the latency experienced by cold requests should be considerably less than it would if tree saturation was present in the network. Furthermore, the latency of hot requests should not be significantly worse (and, as we shall see, might actually be better), as the hot requests will simply wait at the PNI rather than waiting within the network.

Fig. 8 shows the roundtrip latency of cold requests as a function of realized bandwidth for various values of h and the various networks. The latency is measured as the time taken by a request since its generation by the processor until the time the processor receives a response from the memory module (waiting times in all queues are included). The curves were generated by varying the offered load (requests per processor per cycle) from 0 to 1. In all cases, the realized bandwidth reaches some maximum less than one due to congestion in the network. As the offered load increases above this maximum realized bandwidth, latencies increase. All latencies reach a maximum value, however, due to the finite capacity of the queues; if the queues were infinite, then latency would tend to infinity.

Under uniform loading, the latency rises gradually as the bandwidth approaches saturation. In the presence of hot spots, however, latency increases sharply at the point that the maximum bandwidth is reached. This is due to the onset of tree saturation at this point, which quickly transforms a lightly loaded network into a very congested one. This phenomenon can be seen in Fig. 8(a) for a regular Omega network. The higher the hot rate, the lower the saturation bandwidth, and the higher the maximum (due to finite queues) roundtrip latency. This is consistent with the results reported by Pfister and Norton [14].

Fig. 8(b) shows the same curves for a network with simple feedback with $T = 4$. Note that the saturation bandwidths are no different than for the regular network, since all processors are participating in the hot spot activity ($f = 1$). We can see, however, that tree saturation is being reduced, because the latencies experienced are significantly lower than those in the regular network, especially with higher hot rates. We can also see that some tree saturation is still present, because the maximum latency of cold requests still rises sharply at the saturation bandwidths. If tree saturation were completely eliminated, the network buffers would be empty (except, of course, for the normal contention due to uniform traffic) and then the latency of cold requests would not increase at all when the saturation bandwidth was reached.

Feedback with adaptive backoff damping does a much better job of controlling the tree saturation, as can be seen in Fig. 8(c). As the saturation bandwidth for a given hot rate is reached, the latency of cold requests increases slightly, showing some congestion present in the network, but the amount is clearly quite small compared to the regular network and the network with simple feedback. The stronger limiting damping [Fig. 8(d)] can be seen to be even more effective in eliminating tree saturation. With this scheme, latency of cold requests is barely increased

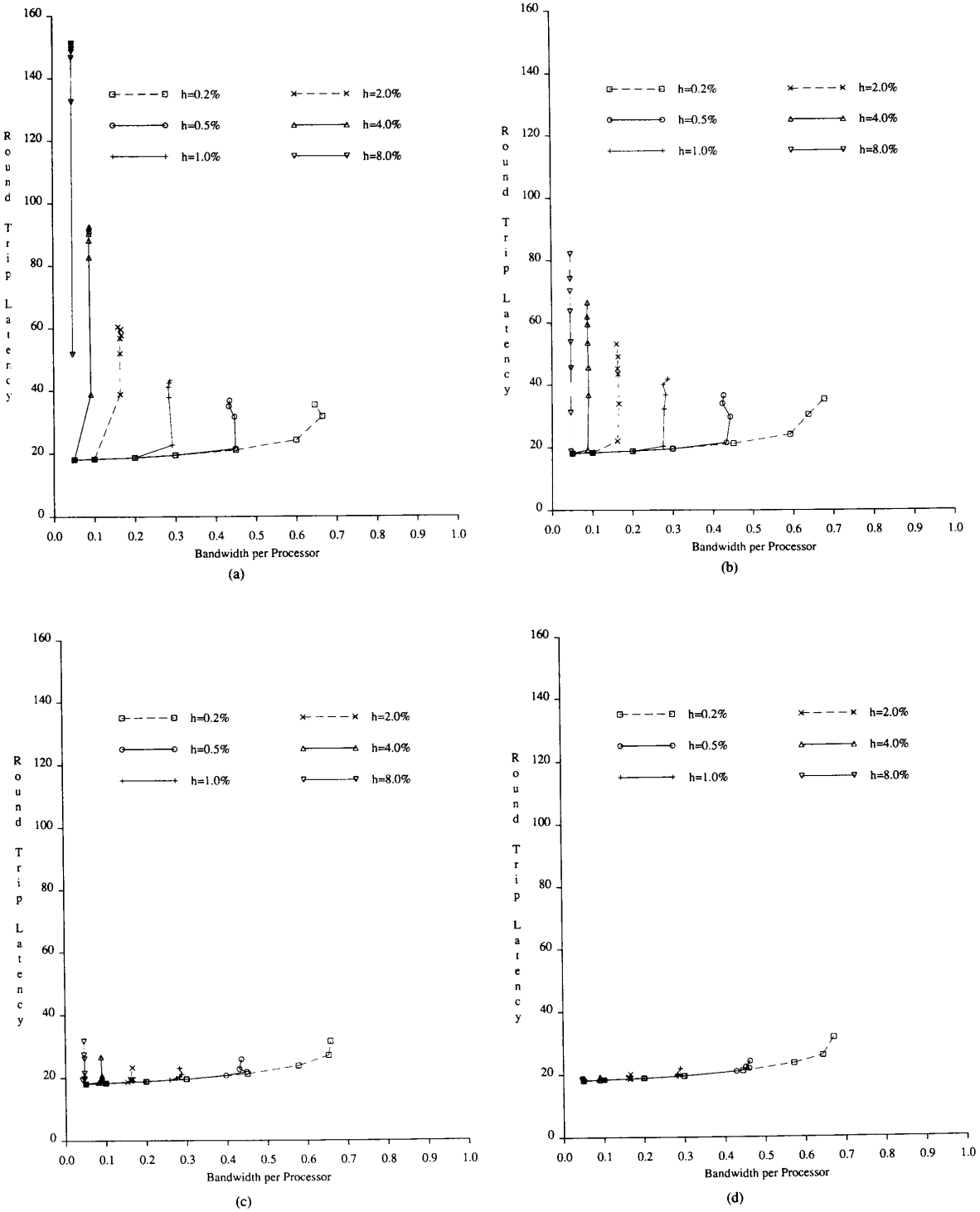


Fig. 8. Latency of cold requests versus bandwidth ($N = 256$, $f = 1$). (a) Regular Omega network. (b) Feedback, $T = 4$. (c) Feedback with adaptive backoff damping. (d) Feedback ($T = 1$) with limiting damping.

when the saturation bandwidth is reached indicating that tree saturation has been eliminated completely.

Fig. 9 presents latency curves for hot requests similar to the latency curves of Fig. 8 for cold requests. In comparing Figs. 8 and 9, several observations are in order. The first observation is that the latencies of hot requests (Fig. 9) are significantly higher than the latency of cold requests (Fig. 8), even in the case of a regular Omega network. (Note that the scale of the Y-axis is different for Figs. 8 and 9.) At first, this may seem incorrect since Pfister and Norton reported that the latencies of hot requests are degraded only slightly more than latencies of cold requests [14]. The difference, however, occurs because of a difference in the networks considered. In [14], processor requests are queued before entering the network, allowing the processors to continue making requests before their last requests have entered the network. In the presence of a hot spot, tree saturation reaches all the way up these queues to the processors, and all requests must wait in these queues until released into the network. Since these queues are the farthest from the hot module, they drain the slowest and the majority of a request's latency consists of waiting in its processor queue. For this reason, hot requests in [14] show only a slightly higher latency than cold requests. Our model, however, does not include queues between the processors and the network. A processor is blocked from submitting another request until its last request has entered the network. Thus, when a given request is made, it effectively starts out its journey at what would be the head of the queues used in [14]. Since half of the cold requests will be able to miss the tree saturation by being routed away from the hot module at the first switch, the average latency of cold requests is substantially lower than that of hot requests in our model. The saturation bandwidths, however, are not affected by this difference because, when $f = 1$, the saturation bandwidths are determined solely by the rate at which the hot memory module can service requests and not by other artifacts of the network.

A second observation is that, except for the highest hot rates, the tree saturation controlling schemes did not reduce the latency of the hot requests, as they did for the cold requests. This is due to the fact that the latencies of hot requests are determined by contention for the hot memory module, *not* by tree saturation or other contention within the network. It can be shown that, as long as the hot memory module remains busy, the average time taken for a hot request to be serviced by the hot memory module is equal to the number of outstanding hot requests (those issued, but not yet serviced) in the system at the time that the request is issued. Thus, only by reducing the number of outstanding hot requests, can the average latency of hot requests be reduced.

A final observation is that under the highest hot rates, some reduction in hot request latency *is* seen in the networks with tree saturation controlling mechanisms. As discussed above, however, this is due solely to a reduction in the number of outstanding hot requests in the system. The tree saturation controlling mechanisms detect the presence of a hot spot and cause subsequent hot requests to be held in the PNI's, blocking the processors from making further requests. This limits the number of outstanding hot requests to one per processor plus the number in the system at the time the hot spot was detected. For the higher hot rates, there would be more than this number of hot requests outstanding, on average, in a regular network. Thus, the tree saturation controlling mechanisms have the effect of reducing the number of outstanding hot requests and so reducing the average latency of hot requests. As in the case of cold requests, if queues existed between the processors and the network, this effect would not be seen, because blocking requests at the PNI would not inhibit the generation of further hot requests.

V. Discussion

So far we have seen that using a simple feedback scheme, which results in the PNI's withholding requests to congested areas in the network, can lead to a reduction in tree saturation and improve overall network performance. By adding damping to the basic feedback system, performance can be improved even further. What other enhancements might we make to achieve further improvements?

One seemingly obvious improvement is to use large queues at the memory modules to increase the buffering of temporary tree saturation. Using larger queues toward the memory side of the network has already been proposed in [16] for general networks. This technique is particularly appropriate for networks with feedback. First, it allows for larger thresholds, as discussed in Section III-A. Recall from the simulation results that the larger the threshold, the less bandwidth was unnecessarily restricted (undershoot). If the threshold can be set sufficiently large such that when a module becomes cold its queue will not drain before new requests arrive, then undershoot can essentially be eliminated. Larger queues at the modules also buffer more of the overshoot tree saturation that occurs with feedback. We have simulated networks with larger queues at the memory modules and found a small, but noticeable, improvement.

Another possible way to improve upon a feedback scheme with damping would be to shorten the delay between inputs and feedback. This would involve taking feedback from points internal to the network. Performance would be enhanced by detecting possible congestion at earlier stages in the network and restricting requests that would aggravate this congestion until it clears. Alternately, mechanisms that fed information back *into* switches within the network could be constructed. The design of such mechanisms is the subject of further study.

Other possibilities include using different feedback information or different damping mechanisms. We have used only the binary value (hot/cold) for the memory module state that is fed back. Perhaps a scheme which fed back a "temperature" indication with greater granularity could be used. This might allow a subtler response on the part of the processors. Alternate damping mechanisms could also be considered. The adaptive backoff scheme presented in this paper does not adapt as quickly as might be desired. It takes a few cold-hot-cold transitions before the proper filtering value is settled upon. Furthermore, the simulation results indicate that our adaptive backoff scheme does not reduce tree saturation as much as the global limiting scheme does. This indicates room for further improvement.

In addition to asking how we can improve upon feedback, we may also ask if it is really needed. It is clear that for the experimental workload used in this paper, tree saturation does occur and feedback techniques are quite effective in its control. However, the assumption that processors pipeline requests and the lack of synchronization constraints between processes may overestimate the problem of tree saturation. But will tree saturation occur for actual workloads in real systems? The analysis in Section II indicates that tree saturation should become a problem with only slight imbalances in network traffic, and that the problem should become worse as system sizes grow. This would indicate that tree saturation from hot memory modules is likely to occur in actual large-scale systems. Moreover, researchers from the IBM RP3 project have verified the presence of significant tree saturation in a running 64-processor system [4]. Thus.

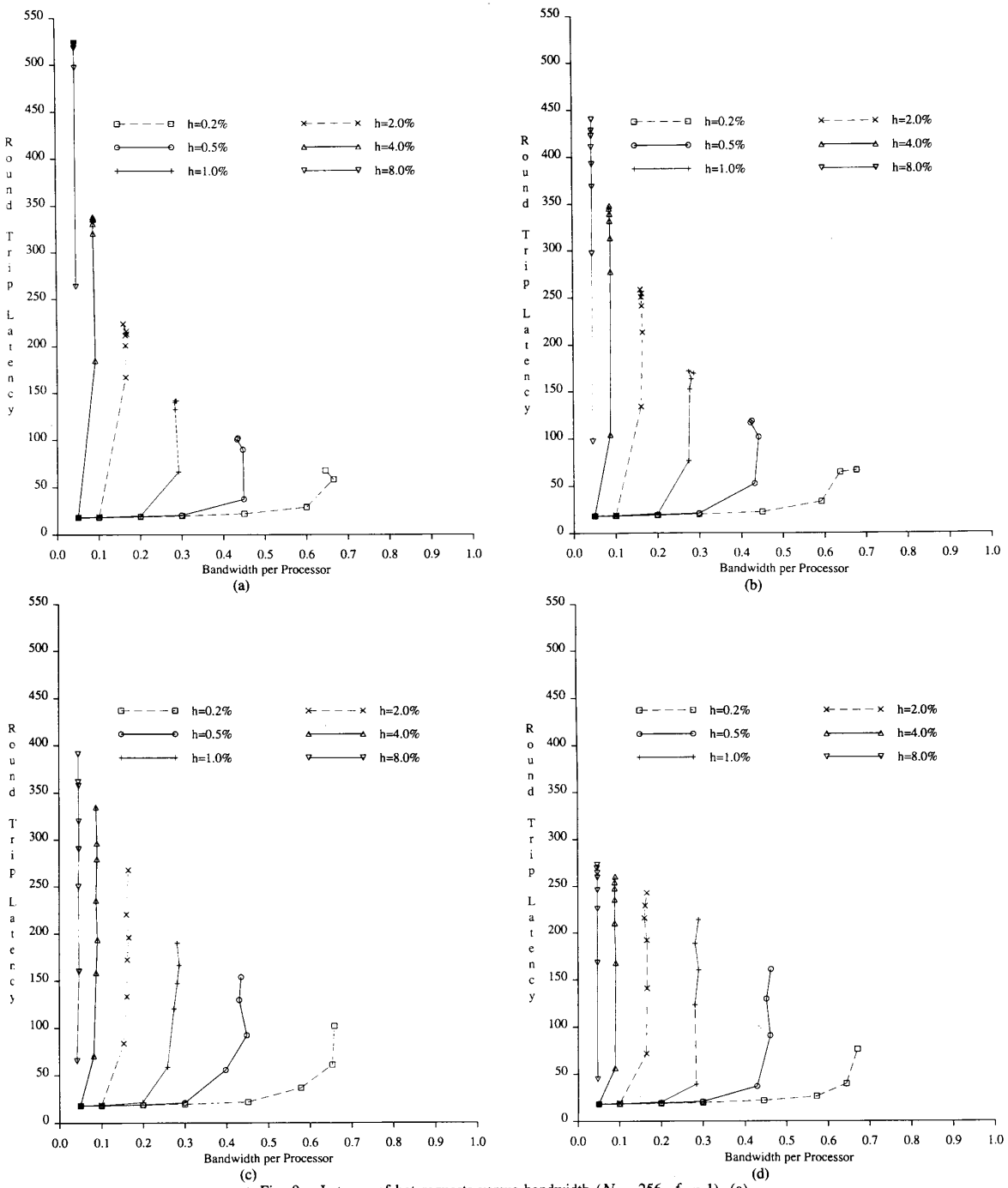


Fig. 9. Latency of hot requests versus bandwidth ($N = 256$, $f = 1$). (a) Regular Omega network. (b) Feedback, $T = 4$. (c) Feedback with adaptive backoff damping. (d) Feedback ($T = 1$) with limiting damping.

we can tentatively answer yes to the above question. The answer should become clearer as more experience is gained in this area. Feedback schemes offer a possible solution if the problem *does* exist. In any case, the cost of implementing a feedback scheme is low compared to the MIN, and, if properly designed, it should have negligible effects on uniform traffic. Furthermore, a feedback mechanism such as the one proposed in this paper can be added to an existing design without changing of the network itself and with minimal additional logic in the memory queues and the PNI's. Because of its simplicity and potential payoff, a

feedback scheme might be warranted to guard against the occurrence of tree saturation even if the possibility of it occurring is remote.

As stated in the Introduction, feedback is not a replacement for combining (either hardware or software), but rather a technique that can be applied where combining cannot. Feedback can be used to prevent tree saturation resulting from a hot module, rather than just from a hot memory location. As such, feedback could be used in conjunction with software or hardware combining to provide protection against hot spots that are not caused by access to synchronization variables. It also can prevent a single process from crippling the entire system by creating a hot spot. In addition, feedback, unlike combining, is not just useful for request–reply type networks. It can be used for networks routing information in one direction only, such as those that might exist in a dataflow machine or a simulation engine.

VI. Concluding Remarks

In this paper, we have proposed the use of feedback in multistage interconnection networks as an aid in the distributed routing process and evaluated the effectiveness of feedback mechanisms in controlling the tree saturation problem in such networks. We saw that, with feedback mechanisms, tree saturation can be limited. This is accomplished by detecting hot spots, and holding requests destined for these spots outside of the network until the congestion clears. This prevents these requests from waiting within the network where they would consume buffer space and block other requests that might otherwise be able to proceed. When tree saturation is limited, system bandwidth can be significantly improved in many cases, and the latency of requests reduced.

We have shown that a system using only simple feedback can exhibit behavior analogous to overshoot and undershoot in classical control theory. Two damping mechanisms, designed to reduce this behavior, were proposed. One of these mechanisms is feasible to implement, while the other was included as a basis for comparison. Both damping schemes significantly improve the performance of a simple feedback mechanism.

The hardware requirements of feedback are relatively modest. Feedback from the destinations to the sources requires no alteration of the interconnection network itself, and thus could be added to existing networks with minimal upheaval. The logic required for simple feedback is straightforward, and that for damping is still small compared to the interconnection network itself.

Feedback is general enough to find many applications in computer system design. It can be used in any parallel or distributed system in which a resource can be accessed without global control and in which contention for access to this resource can degrade the overall system. In the case of multistage interconnection networks, it provides distributed input sources with the information they need to avoid causing tree saturation. A network with feedback presents a promising alternative to a network with global control, which is expensive to implement, or a network with purely distributed routing control, which is prone to degradation due to nonuniform use of its resources.

References

[1] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Redwood City, CA: Benjamin/Cummings, 1989.

[2] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 processor-memory element," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 782–789.

[3] W. L. Brogan, *Modern Control Theory.* New York: Quantum, 1974.

[4] R. Bryant, personal communication, IBM T. J. Watson Research Center, July 1989.

[5] J. R. Goodman, M. K. Vernon, and P. J. Woest, "A set of efficient synchronization primitives for a large-scale shared-memory multiprocessor," in *Proc. ASPLOS-III,* Boston, MA, Apr. 1989, pp. 64–73.

[6] A. Gottlieb *et al.,* "The NYU Ultracomputer—Designing a MIMD, shared memory parallel machine," *IEEE Trans. Comput.,* vol. C-32, pp. 175–189, Feb. 1983.

[7] M. Kalos *et al.,* "Scientific computations on the ultracomputer," Ultracomputer Note 27, Courant Institute, New York Univ., New York, NY.

[8] D. J. Kuck *et al.,* "Parallel supercomputing today and the Cedar approach," *Science,* vol. 21, pp. 967–974, Feb. 1986.

[9] M. Kumar and G. F. Pfister, "The onset of hot spot contention," in *Proc. 1986 Int. Conf. Parallel Processing,* Aug. 1986, pp. 28–34.

[10] T. Lang and L. Kurisaki, "Nonuniform traffic spots (NUTS) in multistage interconnection networks," in *Proc. 1988* Int. Conf. Parallel Processing, Aug. 1988, pp. 191–195.

[11] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.,* vol. C-24, pp. 1145–1155, Dec. 1975.

[12] G. Lee, C. P. Kruskal, and D. J. Kuck, "The effectiveness of combining in shared memory parallel computers in the presence of 'hot spots'," in *Proc. 1986 Int. Conf. Parallel Processing,* Aug. 1986, pp. 35–41.

[13] A. Norton and E. Melton, "A class of Boolean linear transformations for conflict-free power-of-two access," in *Proc. 1987 Int. Conf. Parallel Processing,* Aug. 1987, pp. 247–254.

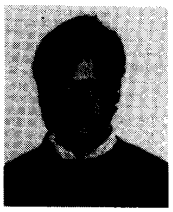
[14] G. F. Pfister and V. A. Norton, "'Hot-spot' contention and combining in multistage interconnection networks," *IEEE Trans. Comput.,* vol. C-34, pp. 943–948, Oct. 1985.

[15] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 764–771.

[16] H. S. Stone, *High-Performance Computer Architecture.* Reading, MA: Addison-Wesley, 1987.

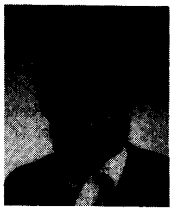
[17] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for VLSI communication switches," in *Proc. 15th Annu. Symp. Comput. Architecture,* Honolulu, HI, June 1988, pp. 343–354.

[18] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large scale multiprocessors," *IEEE Trans. Comput.,* vol. C-36, pp. 388–395, Apr. 1987.



Steven L. Scott (S'87) received the B.S. degree in electrical engineering in 1987 and the M.S. degree in computer science in 1988, both from the University of Wisconsin at Madison.

He is currently a Hertz Foundation fellow and Ph.D. candidate in computer science at the University of Wisconsin. His research interests include multiprocessor architectures and parallel programming.



Gurindar S. Sohi (S'85–M'85) received the B.E. (Hons.) degree in electrical engineering from the Birla Institute of Science and Technology, Pilani, India, in 1981 and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana–Champaign, in 1983 and 1985, respectively.

Since September 1985, he has been with the Computer Sciences Department at the University of Wisconsin–Madison where he is currently an Assistant Professor. His interests are in computer architecture, parallel and distributed processing, and fault-tolerant computing.