Distributing Hot-Spot Addressing in Large-Scale Multiprocessors

PEN-CHUNG YEW, MEMBER, IEEE, NIAN-FENG TZENG, MEMBER, IEEE, AND DUNCAN H. LAWRIE, FELLOW, IEEE

***Abstract*—When a large number of processors try to access a common variable, referred to as *hot-spot* accesses in [6], not only can the resulting memory contention seriously degrade performance, but it can also cause *tree saturation* in the interconnection network which blocks both hot and regular requests alike. It is shown in [6] that even if only a small percentage of all requests are to a hot-spot, these requests can cause very serious performance problems, and networks that do the necessary combining of requests are suggested to keep the interconnection network and memory contention from becoming a bottleneck.**

Instead we propose a software combining tree, and we show that it is effective in decreasing memory contention and preventing tree saturation because it distributes hot-spot accesses over a software tree whose nodes can be dispersed over many memory modules. Thus, it is an inexpensive alternative to expensive combining networks.

***Index Terms*—Combining networks, hot-spot memory, memory bandwidth, memory contention, software combining tree, synchronization.**

I. Introduction

A LARGE, shared-memory multiprocessor system such as Cedar [1], the NYU Ultracomputer [2] or IBM RP3 [3], may contain hundreds or even thousands of processors and memory modules. Multistage interconnection networks such as the Omega network [4] or its variations [5] are usually used to provide communication between these processors and memory modules.

In these systems, any variable shared by these processors will create memory contention at some memory modules. Those shared variables could be locks for process synchronization [15], loop index variables for parallel loops [12], etc. Even though accesses to these shared variables (called *hot-spot* accesses in [3] and [6]) may account for a very small percentage of the total data accesses to the shared memory (typically less than ten percent are observed in most applications), this memory contention can create a phenomenon called *tree saturation* [6], and can cause severe congestion in the interconnection network. It is shown [6], [14] that tree saturation due to hot-spot contention can seriously degrade the effective bandwidth of the shared memory system.

Various schemes like combining networks used in the IBM RP3 [3] and NYU Ultracomputer [2], or the repetition filter memory in the Columbia CHoPP [7] has been proposed to eliminate such memory contention. The basic idea of these schemes is to incorporate some hardware in the interconnection networks to trap and combine data accesses when they are fanning in to the particular memory module that contains the shared variable. Because data accesses can be combined in the interconnection network, it is hoped that memory contention at that memory module can be eliminated.

However, the hardware required for such schemes is extremely expensive. It is estimated [6] that the extra hardware increases the switch size and/or cost by a factor between 6 and 32, and this is only for combining networks consisting of 2×2 switches. With $k \times k$ switches ($k > 2$), the hardware cost will be even greater. The extra hardware also tends to add extra network delay which will penalize most of the regular data accesses that do not need these facilities, unless the combining network is built separately as in RP3 [6].

Furthermore, the effectiveness of the combining network depends very much on the extent to which such combining can be done. If such combining is restricted as described in [8], i.e., if the number of requests that can be combined is restricted to k in a $k \times k$ switch, then the effectiveness of the combining network can be limited. Unless this combining is unrestricted, tree saturation can still occur even in a combining network [8].

In this paper, we are studying this problem from a different perspective. We assume a shared memory multiprocessor system like Cedar [1] with a standard, buffered Omega network providing interconnection [9], and without expensive combining hardware. In addition we use a hardware facility in the shared memory modules to handle necessary indivisible synchronization operations for the shared variables [10]. Regular memory accesses bypass this hardware without delay and, hence, will not be penalized. Each memory module will handle memory accesses, including those memory accesses to shared variables, one at a time.

To eliminate memory contention due to the hot-spot variable, a software tree is used to do the combining. This idea is similar to the concept of a combining network, but it is implemented in software instead of hardware. We will show that this scheme can achieve quite satisfactory results as compared to more expensive hardware combining.

II. Hot Spots and Tree Saturation

The phenomenon of how hot-spot accesses can cause tree saturation is briefly described here. For a more detailed analysis and discussion, please refer to [6].

Manuscript received September 3, 1986; revised November 23, 1986. This work was supported in part by the National Science Foundation under Grants US NSF DCR84-10110 and US NSF DCR84-06916, and by the Department of Energy under Grant DOE DE-FG02-85ER25001.

P.-C. Yew and D. H. Lawrie are with the Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801.

N.-F. Tzeng is with AT&T Bell Laboratories, Columbus, OH 43213.

IEEE Log Number 8613051.

0018-9340/87/0400-0388$01.00 © 1987 IEEE

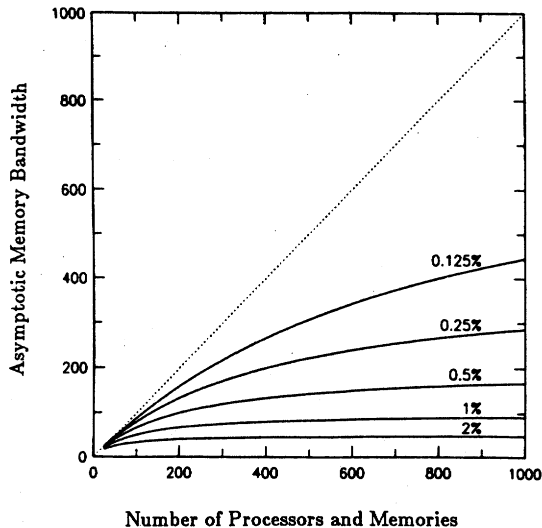


Fig. 1. Asymptotically maximum total network as a function of the number of processors for various fractions of the network traffic aimed at a single hot spot (results from [6]).

Assume N is the number of processors in the system, and there are also N memory modules in the shared memory system. Each processor issues r requests to the shared memory per network cycle ($0 \leq r \leq 1$). Among those requests, h percent of the requests are hot-spot requests. Thus, in each network cycle, there are Nrh hot-spot requests and $r(1 - h)$ normal requests directed to the "hot" memory module for a total of $Nrh + r(1 - h)$. If each memory module can accept 1 request per network cycle (i.e., the maximum rate), the maximum network throughput per processor is

$$H = 1/(1 + h(N-1)) \qquad \text{(A)}$$

and the total effective memory bandwidth for the shared memory system is

$$B = N/(1 + h(N-1)). \qquad \text{(B)}$$

Fig. 1 shows B as a function of N for various h. This clearly shows that in a system with 1000 processors, hot-spot traffic of only one percent can limit the total memory bandwidth B to less than ten percent.

Notice that this discussion assumes that hot-spot requests can continue to be issued from a processor even if that processor still has an unsatisfied hot-spot request pending in the network. In many applications this is not true, because hot-spot requests are usually related to some kind of synchronization operation: A processor usually has to wait for the outcome of the synchronization operation before it can issue another request to the synchronization variable. So, the issuing rate from a processor is inherently limited. We will address these issues in more detail in later sections.

III. Software Combining Tree

To illustrate the principle of a software combining tree, let us assume that we have a variable whose value is N, and that we want each processor to decrement this variable so that when all processors are finished, the value will be zero. This is a common way of making sure all processors are finished with a given task before proceeding with a new task, for example, and is one cause of hot-spot accesses. Now suppose that instead of one single variable, we build a tree of variables, assigning each to a different memory module, as shown in Fig. 2. If $N = 1000$ and assuming a fan-in of 10, we have 111 variables each with value 10. We partition the processors into 100 groups of ten, with each group sharing one of the variables at the bottom of the tree. When the last processor in each group decrements its variable to zero, it then decrements the value in the parent node. Thus, we have increased the total number of accesses from 1000 to 1110, but instead of having one hot spot with 1000 accesses, we have 111 hot spots with only 10 accesses each. It should be clear that this will result in a significant improvement in throughput rate and bandwidth, and the simulations we describe later verify that even if we account for the increase in total accesses, the improvement is still quite significant. It should also be clear that a three-level tree with fan-in equal to ten is not necessarily optimal, but that the optimal point depends on access times and on other factors.

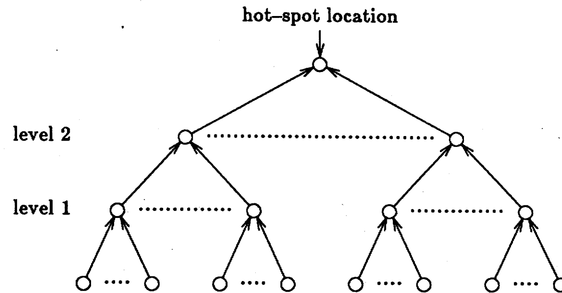


Fig. 2. A software combining tree with fan-in of 10.

Another basic operation that can be implemented with a software combining tree is *busy-wait*. Here it is assumed that processors are waiting for a shared variable to change in some way. Presumably some other processor will cause this change. We build a combining tree as before, this time assigning one processor to each node in the tree. Each processor monitors the state of its node by continually reading the node. When the processor monitoring the root node detects the change in its node, it in turn changes the state of its children's nodes, and so on until all processors have detected the change and are able to proceed with the next task.

This idea, in a sense, is not very different from a hardware combining tree built from 10×10 switches, except that the combining buffer that would be inside each switch now resides in a shared memory module in a software combining tree. One distinct advantage for a software combining tree is that we can tune performance by changing the fan-in of each node without incurring any hardware cost.

A. Modeling of Software Combining Tree

We will classify hot-spot accessing in two ways. First, accesses will be *limited* or *unlimited* depending on whether a given processor can have only one or more than one hot-spot request outstanding. We let η denote the number of outstanding hot-spot requests. Second, the number of accesses will be *fixed* or *variable* depending on whether the total number of

accesses is fixed, or whether the total number varies depending on the number of conflicts or some other factor. For example, assume we are adding a vector of numbers to form a sum. Then each processor can have more than one outstanding request to add an element to the shared sum, but since we assume the addition is done indivisibly by logic in the memory, the total number of requests generated by all the processors is fixed. This case is *unlimited-fixed.* A case like the one described earlier, where processors are decrementing a counter to see which is the last processor, is *limited-fixed.* A third example is illustrated by *busy waiting* where the processors may all be waiting for one processor to complete some task. Each processor continually reads the value of a shared variable until the value changes, for example from zero to one. Thus, the number of requests to the hot-spot depends on how soon the variable gets reset, and this case is *limited-variable.* Notice that a barrier synchronization [11] can be implemented by a counter decrement (*limited-fixed*), followed by a busy wait (*limited-variable*) triggered by the final processor which decrements the counter.

When we implement combining trees for hot-spot accesses, it is important to minimize the possible memory contention, so it is preferable that all shared variables in a software combining tree (i.e., the nodes of the tree) reside in separate memory modules. The largest combining tree we can construct for a hot-spot is a tree with minimum fan-in, i.e., a fan-in of two. The total number of nodes in a combining tree with N leaves is $N/2 + N/4 + \cdots + 2 + 1 = N - 1$. Hence, it is always possible to spread those nodes across N separate memory modules. Our simulations in this study assume all of the nodes in a software tree to be in separate memory modules.

We also assume the following system configuration in our simulations.

1) There are two identical, back-to-back, unidirectional Omega networks: one is for traffic from processors to the shared memory; the other is for traffic from memory returning to the processors. Both networks are packet-switching, pipelined networks.

2) Each network consists of 2×2 switching elements with an output buffer of finite size at each output port of a switching element. The fan-in capability of each output port is two, i.e., it can accept two simultaneous requests from its two input ports. One request is forwarded to the next stage and the other is stored in the output buffer. If the output buffer is full, no more requests are accepted by the output port. In our simulations, we assume the size of the output buffer to be four.

3) There are many different algorithms to implement software combining trees for various types of shared variables [17]. It is beyond the scope of this paper to describe those algorithms. Instead, we used a very general and simplified model to simulate a software tree. Each node of a software tree contains a counter with an initial value of 0. In the limited-variable case, the counter is decremented to -1 by the first processor which visits the node. The rest of processors sharing the node will be busy-waiting whenever the counter value is -1. The extra delay for busy-waiting is accounted for in simulations. The first processor will visit its parent node and bring back a positive value equal to the fan-out of the node. The counter is set to that value and the rest of processors can then decrement the counter and move on. The counter will eventually become 0 again and the whole process will repeat. This model is very similar to broadcasting a scalar to all processors through a software combining tree. The scalar may be updated from time to time. In the limited-fixed or the unlimited-fixed case, processors will increment the counter until it equals to the fan-out of a node. A representative is then chosen to reset the counter and also to increment the counter in its parent node. The whole process will repeat at the parent node. This model is very similar to the first part of a barrier synchronization where processors increment a counter to see if all of them have reached the barrier.

4) All requests are of the same length. In our simulations, we assume each request consists of only one packet.

5) The access time of a memory module is one network cycle, i.e., the time for a request to go through a switching element when no conflict exists.

B. Possible Overhead in a Software Combining Tree

As mentioned earlier, constructing a software combining tree creates many shared variables. Therefore, more hot-spot traffic is created even though that traffic generates less memory contention.

As before, let us assume that the hot-spot rate from a processor is $r \times h$, and the software combining tree has a fan-in of k for each node. For *fixed-type* access patterns, the fractional increase in hot-spot traffic will be

$$\sum_{l=1}^{\log_k N-1} rh/k^l = rh\left(\frac{1-(k/N)}{k-1}\right).$$

When $k = 2$, the increased hot-spot traffic is $rh(1 - 2/N)$, which approximates the original hot-spot traffic for large N. This means that the hot-spot traffic cannot be more than doubled after all of the extra hot-spot traffic is included. As we will see later in our simulations, the decreased memory contention will more than offset the increased hot-spot traffic if h is less than 30 percent.

For *variable* access patterns, the additional accesses caused by the combining tree are difficult to quantify because the number of accesses is not fixed to begin with. In practice, since busy-waiting is often the cause of *variable* access patterns (with $\eta = 1$), and the number of accesses for a busy-wait operation depends on how quickly the state change is propagated to the children in the tree, the total number of accesses could even be less than that required by a single shared variable; the state change can be propagated more quickly by the combining tree than by N accesses to a single shared variable.

IV. Bounds on Bandwidth

A. Unlimited Hot-Spot Requests Per Processor

In a packet-switching Omega network, with finite buffers in each switching element and with hot-spot rate $h = 0$, we still cannot achieve 100 percent memory bandwidth because of conflicts in the network [9]. These conflicts are also possible if

a crossbar switch is used. If we assume R to be the maximum request rate reaching a memory module when no hot spot exists, then in a steady state, R is also the maximum request rate allowed for a processor. Therefore, we can consider R to be an absolute upper bound on the bandwidth per processor.

The value of R depends on the network buffer size, the length of a request, and the network switch size, etc. [13]. However, as h increases, the request rate to the hot memory module, i.e., $r(1 - h) + rhk$, will increase from R to 1. Tree saturation will occur when the request rate to the hot memory module approaches 1, and the maximum processor request rate r will decrease. Hence, we have

$$R \leq r(1-h) + rhk \leq 1.$$

By rearranging the above equation, we have the following:

$$R/(1+h(k-1)) \leq r \leq 1/(1+h(k-1)).$$

$1/1 + h(k - 1))$ is equal to 1 when h is 0. Since the absolute upper bound is $R(R \leq 1)$ as discussed before, we can have a tighter upper bound by using R, i.e.,

$$R/(1+h(k-1)) \leq r \leq R \qquad \text{(C)}$$

Notice that (C) also shows a lower bound for the maximum processor request rate r when a software combining tree is used with a fan-in of k, and η is unlimited, i.e., even when η is unlimited, the maximum bandwidth cannot be worse than $NR/(1 + h(k - 1))$.

We obtained R from simulations, and in Fig. 3 we plot lower bounds for various system sizes with h varying from 0–32 percent. Notice that those curves are in a very narrow range, i.e., the lower bound in (C) seems to be tight at least for systems up to size 1024. The top dotted line in Fig. 3 shows R, the maximum bandwidth we can get when there are no hot-spot requests.

The degradation factor in (C) is $1 + h(k - 1)$. This degradation factor is independent of the system size and reaches a minimum when $k = 2$. Given unlimited hot-spot requests, i.e., $\eta \geq 1$, the optimal software combining tree for maximum memory bandwidth has a minimum fan-in of 2.

B. Single Hot-Spot Request Per Processor

If the hot-spot request rate is limited ($\eta = 1$), then there cannot be more than N hot-spot requests in the system at any time. For systems with instruction look-ahead or with data prefetching capability, regular requests still may be issued while a hot-spot request is pending. However, this case is not different from that of unlimited hot-spot requests with a very small h; when h is very small, it is unlikely that there will be more than one hot-spot request pending at any time.

Hence, when $\eta = 1$, we will only consider the case where no additional requests, hot or regular, are issued by the processor when there is a pending hot-spot request. Thus, the bandwidth depends on the delay of the hot-spot requests. The request rate from the processor is further restricted by any increased delay. If a software combining tree is used to

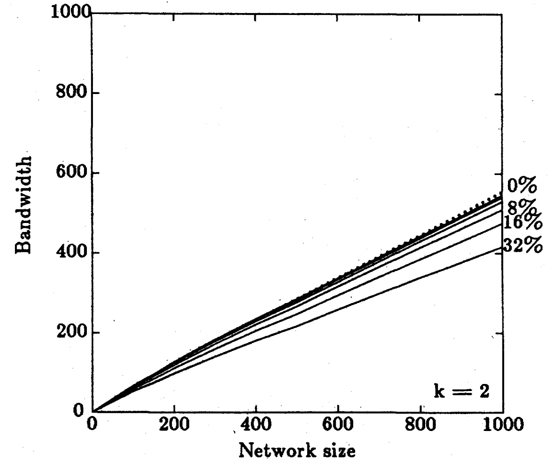


Fig. 3. Lower bounds on network bandwidth for various hot-spot rates (η = unlimited).

eliminate the memory contention caused by the hot-spot requests, the limiting factor for the memory bandwidth will only be η; the inherent nature of the hot-spot that prohibits further processor requests.

During a long period of time T, there will be rT requests from a processor, among which rhT requests are hot-spot requests. The processor will be barred from issuing any request for a total period of $rhTC$ where C is the average round-trip delay for a hot-spot request. The processor can issue a request only for a total period of $T - rhTC$. Within that period, $r(1 - h)T$ regular requests are issued. Hence, the real issuing rate for regular requests is $r(1 - h)T/(T - rhTC)$. This rate cannot be greater than 1, i.e.,

$$r(1-h)T/(T-rhTC) \leq 1.$$

This equation can be rearranged to obtain an upper bound for r

$$r \leq 1/(1-h+hC). \qquad \text{(D)}$$

As expected, the maximum rate of r is greatly dependent on the hot-spot delay C. This bound gets tighter as the hot-spot rate h gets larger. When $h = 1$, the equality in (D) will hold. Fig. 4 shows this upper bound for various hot-spot rates h with minimum hot-spot delay of $C = 2 \log_2 N$. For $N = 1000$ and $h = 8$ percent, the upper bound will be around 40 percent of the total bandwidth. Notice that *the upper bound in* (D) *is valid even for a hardware combining network* because it is a bound imposed by the inherent nature of the hot-spot request (i.e., $\eta = 1$).

V. Simulation Results

To study the effectiveness of a software combining tree, we performed several simulations for $N = 256$, with h varying from 0–32 percent. These simulations are based on the system models described in Section III-A. Fig. 5 shows the delay and maximum bandwidth when neither a software combining tree nor a hardware combining network is used. Following each curve from left to right, each point represents a larger value of r. As shown in [6], while r increases, bandwidth increases

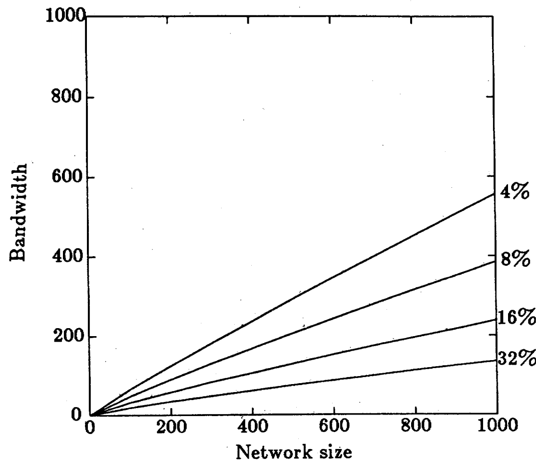
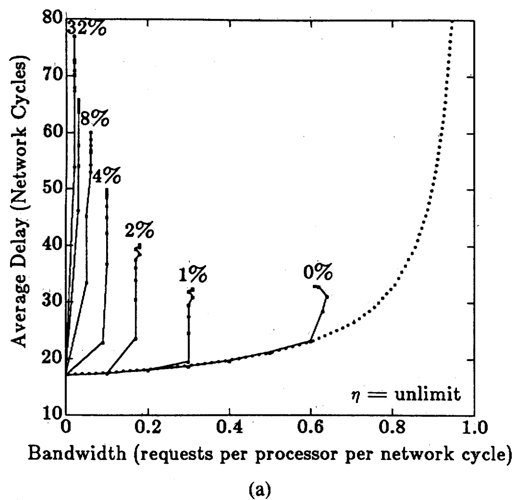
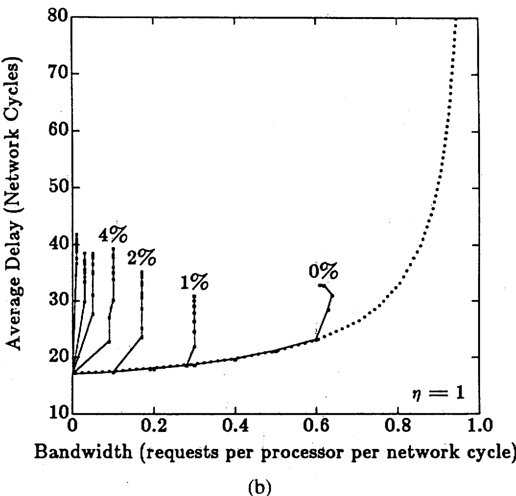


Fig. 4. Upperbounds on network bandwidth for various hot-spot rates ($\eta = 1$).



(a)



(b)

Fig. 5. Average network delay versus bandwidth for a network of size 256 (h varies from 0 to 32 percent).

while delay stays relatively constant up to a point of saturation. After the saturation point, bandwidth ceases to increase while delay gets worse. The results clearly show low bandwidth and increased average network delay results. The maximum bandwidth of $0.63N$ is achieved when $h = 0$.

Fig. 6 represents *fixed*-type access patterns with unlimited η, and shows the use of a software combining tree to reduce hot-spot contention. The fan-in's for the software combining trees are varied from $k = 16$ to 2. The improvement is quite significant compared to the result in Fig. 5(a). According to (C), the minimum degradation factor for the bandwidth can be obtained when the software combining tree has the minimum fan-in. In Fig. 6 we can see that when $k = 2$, the degradation is indeed the smallest.

As presented in Section III-B, the hot-spot traffic can be nearly doubled by the extra hot-spot traffic created by the software combining tree with the minimum fan-in $k = 2$. In Fig. 6, h is indicated as the original hot-spot request rate; the results shown there already include all extra hot-spot traffic. This shows that with an original hot-spot request rate of 16 percent, the degradation remains small. The elimination of the hot-spot contention, indeed, more than offsets the results of increased traffic.

We also simulate some cases for *fixed*-type access patterns with $\eta = 1$ (Fig. 7). If we take into account the fact that busy-waiting is not required in this kind of access pattern, we can see that the results are quite similar to those from our simulations of *variable*-type access patterns discussed above. In fact, the average hot-spot request delay, i.e., C in (D), is smaller in this case. Also, as shown in (D), we can expect an improved maximum rate r.

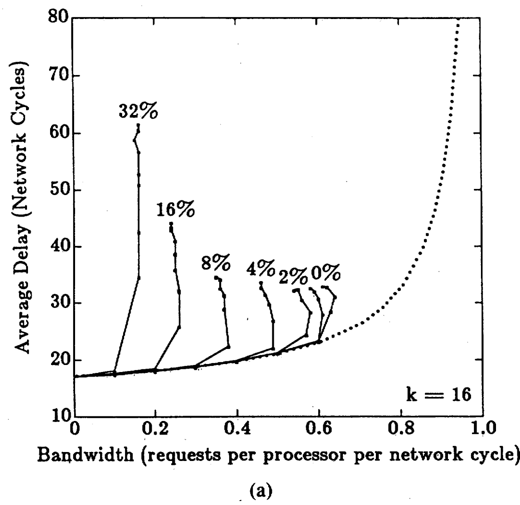
Fig. 8 represents *limited-variable* access patterns, wherein no additional requests are issued by a processor while it has a hot-spot request pending, but the total number of requests allowed over time is not fixed. The upper bound on the bandwidth given in (D) will depend on C, the average delay of the hot-spot requests. The value of delay C includes the overhead from traversing the software tree, busy waiting in the intermediate nodes, and the possible memory contention. From these figures, we can see that the optimal fan-in k for the software tree is no longer $k = 2$, but rather at around $k = 4$. The increased fan-in k allows for a lesser number of levels of nodes in the tree, thus reducing the time required for requests to traverse the tree.

Furthermore, when h increases, the upper bound in (D) becomes tighter. There is less traffic in the network due to the restriction that no more requests will be issued when a hot-spot request is pending. In this case, the turnaround time for a request can actually improve as Fig. 9 shows.

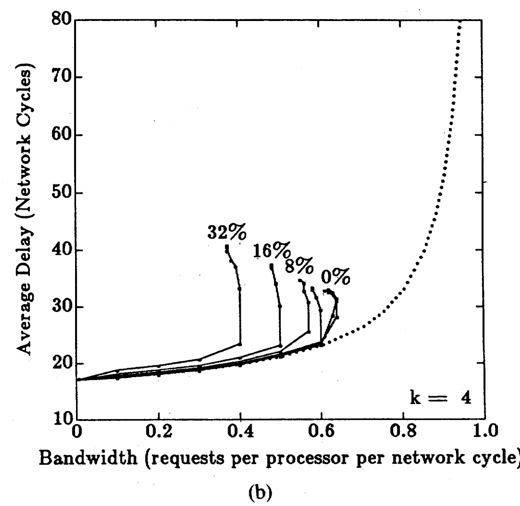
The lower dotted lines in Fig. 5 through Fig. 9 are the average delay of a request through the network assuming the buffer size in each switching element is unlimited. These values are calculated based on the analytical model in [16].

VI. Discussion

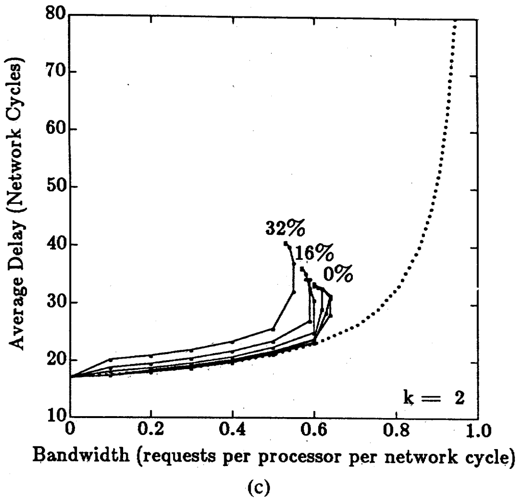
Our simulations show that the software combining tree effectively eliminates tree saturation caused by hot-spot contention. However, the main purpose of the software



(a)

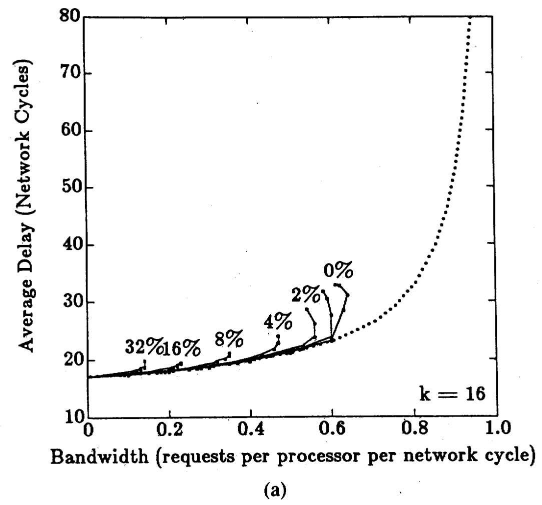


(b)

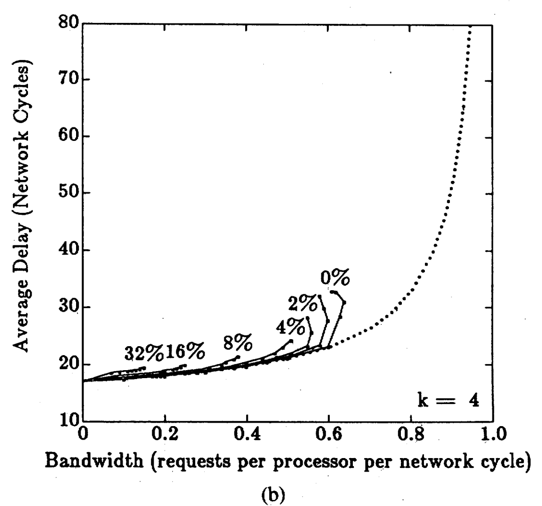


(c)

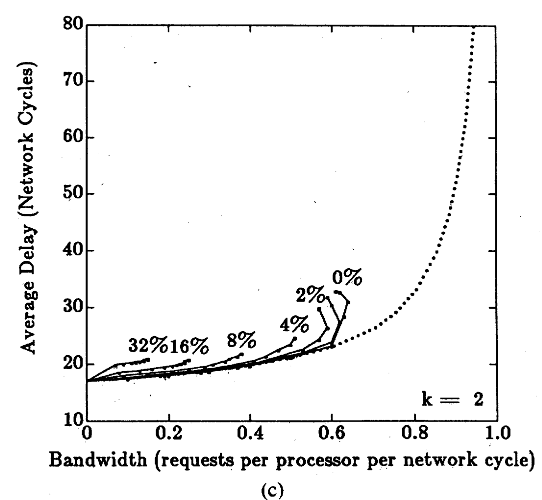
Fig. 6. Average network delay versus bandwidth for unlimited-fixed access patterns (N = 256, h varies from 0 to 32 percent).



(a)



(b)



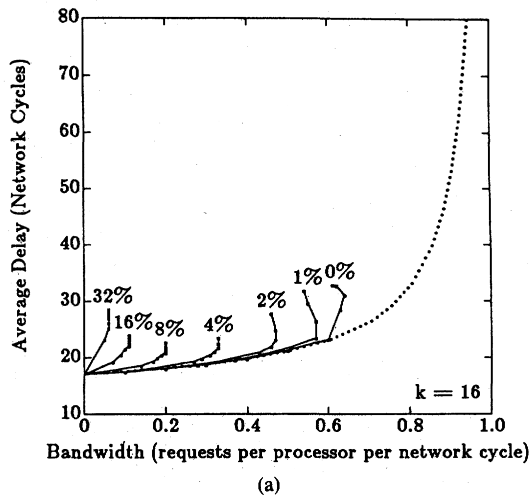
(c)

Fig. 7. Average network delay versus bandwidth for limited-fixed access patterns (N = 256, h varies from 0 to 32 percent).

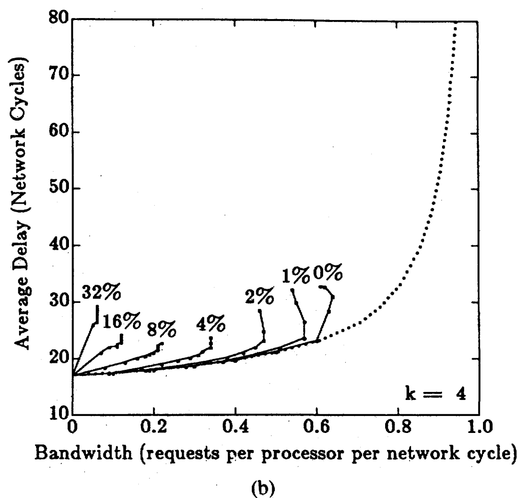
combining tree differs slightly from the original purpose of the hardware combining networks [2], [6].

Hardware combining networks were originally proposed to speedup hot-spot requests by combining those requests in the interconnection network and in this way eliminate memory contention at the hot memory module. Because such memory contention creates the serious side effect of tree saturation that can adversely affect even regular requests [6], such requests must also be processed through the hardware combining network. Although it alleviates the problem of tree saturation, hardware combining can cause extra delay in processing regular requests.

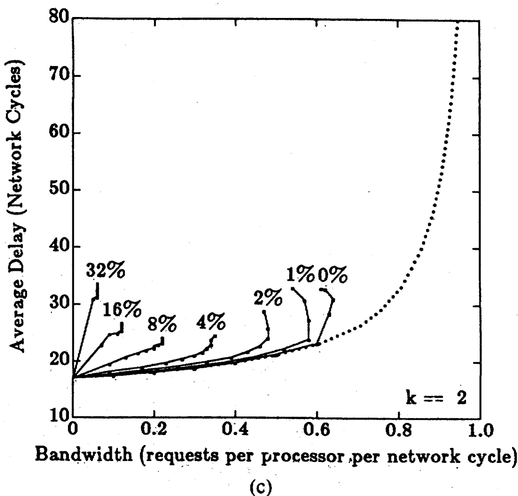
Software combining trees seem to effectively relieve regular requests from the side effect of tree saturation, without the



(a)



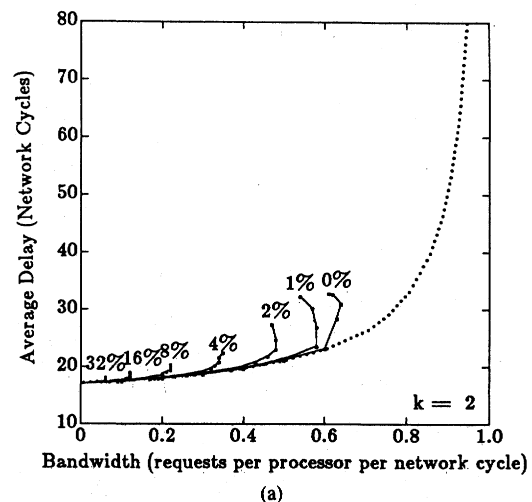
(b)



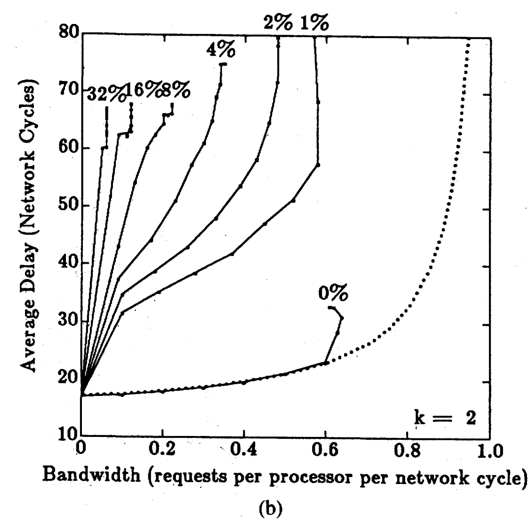
(c)

Fig. 8. Average network delay versus bandwidth for limited-variable access patterns (N = 256, h varies from 0 to 32 percent).

expense of hardware combining networks. The beneficial result from this scheme is that the service time of hot-spot requests decreases. Theoretically, this improvement cannot be as good as a hardware combining network with unrestricted combining capability: In a software combining tree, a hot-spot request must traverse the interconnection network $\log_k N$ times, whereas in a hardware combining network the request



(a)



(b)

Fig. 9. (a) Average delay of regular requests versus bandwidth for limited-variable access patterns (N = 256, h varies from 0 to 32 percent). (b) Average delay of hot spot requests versus bandwidth for limited-variable access patterns (N = 256, h varies from 0 to 32 percent).

must traverse the network only once. However, in a real implementation, unrestricted combining in a switch is impossible due to the complexity of the switches in a hardware combining network. This will inevitably hamper the effectiveness of a combining network [8], and also introduces increased delay due to the extra hardware. It is difficult to determine the system size requirements necessary to prove the hardware combining network to be the optimal method of speeding up hot-spot requests. The effect of the somewhat slower hot-spot requests on total system performance, if the rate is very small, also remains to be seen.

VII. Conclusions

When a large number of processors try to access a common variable, referred to as *hot-spot* accesses in [6], not only can the resulting memory contention seriously degrade performance, but it can also cause *tree saturation* in the interconnection network which blocks both hot and regular requests alike. It is shown in [6] that even if only a small percentage of all requests are to a hot spot, these requests can cause very serious performance problems, and networks that do the

necessary combining of requests are suggested to keep the interconnection network and memory contention from becoming a bottleneck.

Instead we propose a software combining tree, and we show that it is effective in decreasing memory contention and preventing tree saturation because it distributes hot-spot accesses over a software tree whose nodes can be dispersed over many memory modules. Thus, it is an inexpensive alternative to expensive combining networks.

References

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

[2] 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.

[3] G. F. Pfister *et al.*, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug. 1985, pp. 764–771.

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

[5] C. L. Wu and T.-Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, pp. 694–702, Aug. 1980.

[6] 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.

[7] H. Sullivan, T. Bashkow, and D. Klappholtz, "A large scale homogeneous, fully distributed parallel machine," in *Proc. Fourth Ann. Symp. Comput. Architect.*, June 1977, pp. 105–124.

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

[9] D. M. Dias and J. R. Jump, "Analysis and simulation of buffered delta networks," *IEEE Trans. Comput.*, vol. C-30, pp. 273–282, Apr. 1981.

[10] C. Q. Zhu and P. C. Yew, "A synchronization scheme and its applications for larger multiprocessor systems," in *Proc. 4th Int. Conf. Distrib. Comput. Syst.*, May 1984, pp. 486–493.

[11] P. Tang and P. C. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *Proc. 1986 Int. Conf. Parallel Processing,* Aug. 1986, pp. 528–535.

[12] E. L. Lusk and R. A. Overbeek, "Implementation of monitors with macros: A programming aid for the HEP and other parallel processors," Argonne Nat. Lab., Argonne, IL, Tech. Rep. ANL-83-97, Dec. 1983.

[13] P. Y. Chen, "Multiprocessor systems: Interconnection networks, memory hierarchy, modeling and simulations," Dep. Comput. Sci., Univ. Illinois, Urbana, Rep. UIUCDCS-R-82-1083, Jan. 1982.

[14] R. Lee, "On hot spot contention," *ACM SIG Comput. Architect.*, vol. 13, pp. 15–20, Dec. 1985.

[15] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. Ass. Comput. Mach.*, vol. 8, pp. 569–569, Sept. 1965.

[16] C. P. Kruskal and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Trans. Comput.*, vol. C-32, pp. 1091–1098, Dec. 1983.

[17] P. Tang and P. C. Yew, "Algorithms for distributing hot-spot addressing in large multiprocessor systems," Center for Supercomputing R.&D., Univ. Illinois, Urbana, Cedar doc. 617, Dec. 1986.

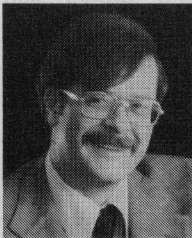
Pen-Chung Yew (S'76–S'78–M'80–S'80–M'81) received the BSEE degree from National Taiwan University, Taiwan, in 1972, the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst, 1977, and the Ph.D. degree in computer science from the University of Illinois, Urbana, in 1981.

He is currently an Assistant Professor in the Department of Computer Science and a Senior Computer Engineer in the Center for Supercomputing Research and Development, University of Illinois, Urbana. He has been working on the architecture and hardware design for the Cedar supercomputer since 1984. His current research interests are parallel processing, computer architecture, high-performance multiprocessor systems, and performance evaluation.

Nian-Feng Tzeng (S'85–M'87) was born on November 22, 1956 in Taichung, Taiwan, Republic of China. He received the B.S. degree in computer science from National Chiao Tung University, Taiwan, the M.S. degree in electrical engineering from National Taiwan University, Taiwan, and the Ph.D. degree in computer science from the University of Illinois, Urbana, in 1978, 1980, and 1986, respectively.

Currently, he is a member of the Technical Staff with AT&T Bell Laboratories, Columbus, OH. Prior to joining Bell Laboratories, he was a Research Assistant with the Center for Supercomputing Research and Development, University of Illinois, Urbana, where he had been involved in the Cedar supercomputer project for more than two years. His research interests include interconnection networks, fault-tolerant computings, distributed and parallel processings, and computer architectures.

Dr. Tzeng is a member of Tau Beta Pi.



Duncan H. Lawrie (S'66–M'73–SM'81–F'84) is currently Professor of Computer Science, Professor of Electrical and Computer Engineering, and Associate Director of the Center for Supercomputing Research and Development at the University of Illinois, Urbana. He has contributed to the design of several large computers including the Illiac IV where he designed and implemented Glypnir, the first high-level language for that machine, and the Burroughs Scientific Processor where he was a Principal Architect, specializing in the array memory system. He is currently a Principal Architect of the Cedar large-scale multiprocessor at the University of Illinois, and directs the compiler and operating systems development work for that machine. His main interest is in the area of design and evaluation of computer architecture, with specialization in the areas of high-speed algorithm design, communication networks, virtual memory performance, and the use of mass storage devices. He has been a consultant to industry and government in the areas of computer organization, local networking, and applications studies.

Dr. Lawrie was Chairman of the Symposium on High-Speed Computer and Algorithm Organization, Program Chairman of the Ninth International Conference on Parallel Processing, and General Chairman of the Fourth International Conference on Distributed Computing Systems. He has also served as Editor of the Computer Architecture and Systems Department of the *Communications of the ACM,* and was the Chairman for Conferences and Chairman for Tutorials of the Conferences and Tutorials Board as well as Acting Vice President for Publications of the IEEE Computer Society. He is a Member and Secretary of the IEEE Computer Society Governing Board (1986–1987), and is a member of the Association for Computing Machinery.