A Cost-Effective Combining Structure for Large-Scale Shared-Memory Multiprocessors

Nian-Feng Tzeng, *Senior Member, IEEE*

Abstract—**In a large-scale shared-memory multiprocessor, concurrent requests to a shared location made by many processors can create a serious problem, known as *hot-spot contention*, which drastically degrades access to *all* memory locations. A combining network has been proposed to alleviate this problem by combining requests destined for the same location into a single request. Although it can effectively eliminate hot-spot contention, the combining network suggested for the IBM RP3 and the NYU Ultracomputer tends to be very expensive for large-scale systems.**

A cost-effective combining structure is proposed for large-scale multiprocessors in this paper. The key idea of the new combining structure lies in that the combining function is separated from the routing function, so that a binary tree-based combining configuration becomes feasible. The combining element for realizing the new structure has only one wait buffer and one combining logic, involving less hardware than the element used in earlier combining networks. More importantly, the number of constituent combining elements in such a structure with size N is reduced to $O(1/\log_2 N)$ of that required in the previous design. The proposed combining structure in conjunction with a regular multistage interconnection network can remove hot-spot contention effectively in any sized system at a considerably lower cost, and appears readily suitable for use in some applications.

Index Terms—**Access contention, binary trees, hot-spot locations, mean latency, multiprocessors, multistage interconnection networks.**

I. Introduction

DUE to the advances in VLSI technology, we have an opportunity to build a cost-effective multiprocessor system composed of hundreds or even thousands of processors and memory modules. Such a large-scale multiprocessor can be interconnected by a multistage interconnection network (MIN), like the Omega network [1] or its variations [2], to provide processors (in one end) with communication paths to shared memory modules (in the other end). Notable shared-memory multiprocessors include, among others, the CHoPP of Columbia University [3], the PASM of Purdue University [4], the Cedar of University of Illinois [5], the Ultracomputer of NYU [6], the Butterfly of BBN [7], and the RP3 of IBM [8].

In a large-scale shared-memory multiprocessor, requests for the synchronization or resource management purposes may head for the same memory location simultaneously, leading to serious access conflicts known as *hot-spot contention* [9]. Hot-spot contention significantly degrades all memory access, not just access to the shared location. It has been shown that potential degradation due to even modest hot-spot traffic can be very remarkable and the degree of degradation extends as the system size grows [9]–[11]. To reduce the adverse impacts of hot-spot contention on network performance, requests simultaneously directed to the same location are combined into a single one before being forwarded to the shared location [9], [12], [13], [28]. The number of hot-spot accesses to the shared location is reduced drastically this way, and ideally it may become independent of the system size. (In fact, the combining capability was first introduced in the Columbia CHoPP to deal with general memory traffic [3].) The resulting network in which every switching element has the combining and routing capabilities is called a *combining network*. The combining network has been suggested for the Ultracomputer of NYU [9], [12] and the RP3 of IBM [13]. It is clear that with provision of the combining capability in a network, hot-spot contention can be eliminated effectively.

A switching element with the combining capability tends to be very complicated, as those combining switches designed for the Ultracomputer and the RP3 [9], [12], [13], [18]. Because of its high complexity, such a switch is likely to be fabricated by high-density NMOS or CMOS technology rather than by high-speed TTL or ECL technology [8], [26]. One may use the complicated switches to construct a MIN for both regular and hot-spot requests. Although straightforward, this MIN imposes unnecessarily long delays on regular requests due to slowing the network clock down. In light of this, the IBM researchers decided to employ two separate copies of multistage interconnection networks in RP3, with one for regular requests and the other for hot-spot requests [8], [9]. The size of the combining network may be reduced by utilizing multiplexers to make several processors share one network input. This seems to be a good design choice because regular requests now experience no extra delay, while the complexity of the combining network is kept lower. Nevertheless, the overall network cost is still high.

A close observation on the previous design reveals that the combining network, though expensive, has low utilization because only a small portion of the switches in the network evokes the combining capability during any period, although all of them are equipped with that capability. This indicates that the complexity of the network can possibly be lowered by constructing a simpler, yet better utilized combining structure. A tree with each node having the com-

Manuscript received October 17, 1990. This work was supported by the National Science Foundation under Grant MIP-8807761. A preliminary version of this paper was presented at the 1989 International Conference on Parallel Processing.

The author is with the The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

IEEE Log Number 9200209.

bining capability, referred to as a *combining tree*, appears sufficient to carry out the combining function and reduce hot-spot contention [25], [26]. A combining tree is dedicated to combining requests only, and it does not perform any routing function.

In this paper, a novel combining structure based on the combining tree is investigated. Being a tree structure, the combining tree involves much fewer combining elements than the earlier combining network. In addition, each combining element has lower hardware complexity. The overall structure consists of a combining tree and a regular multistage interconnection network. Combined requests, on leaving the combining tree, are routed to their proper destinations through the regular multistage interconnection network, which is also used for delivering nonhot-spot requests. Because it can effectively remove hot-spot contention in any sized network at a considerably lower cost, the new combining structure seems readily suited for use in practice.

For the proposed combining structure to work successfully, hot-spot traffic must be differentiated from regular traffic at run time so that it can be directed to the combining tree. This is not always possible but could often be achievable, because we believe that the most serious and frequent source of hot-spot traffic is due to synchronization and resource management purposes (see, for example, [20]–[23]) employing a set of primitive instructions which can be detected by examing (a part of) the op-codes at run time. It is found from the simulation results that a setup of this structure which gives rise to low latency of hot-spot requests can be identified for parallel program executions where the hot-spot rate or the hot-spot number varies over a reasonably limited range. This implies that the proposed structure is likely to reduce hot-spot contention effectively even without changing its configuration during the course of executing a parallel program where the hot-spot rate or the hot-spot number does not fluctuate excessively, a typical situation.

This paper is organized as follows. In Section II, the impact of hot-spot access on network performance is briefly reviewed, followed by the description of a previous solution. The new combining structure is presented in Section III and is then discussed in Section IV. To examine the effectiveness of the proposed combining structure, simulation study is adopted and its results are provided in Section V. Concluding remarks appear in Section VI.

II. Hot-Spot Contention and Previous Solution

Concurrent access to a shared location can cause severe congestion in the interconnection network of a large-scale multiprocessor. The shared location could be a lock for process synchronization [20], [21], a loop index for parallel loops [22], a task queue for self-scheduling [23], etc. Although access to such a shared location (called *hot-spot requests*) often accounts for a small fraction of total requests through the interconnection network, it can lead to a phenomenon termed *tree saturation* [9], which once developed, would seriously degrade network effective bandwidth and mean latency.

To have an insight into this problem, a brief explanation of hot-spot contention is given next, followed by an overview of the previous combining switch design. A detailed analysis and discussion about this phenomenon can be found in [9]–[11].

A. Hot-Spot Contention

Consider a square network of size N, where all inputs are involved in hot-spot access. Suppose that there is only one hot-spot output location (the case of multiple hot-spots would have lighter impacts on network performance [10], [14]). Let us assume that each input (i.e., processor) during a cycle generates r, $0 \leq r < 1$, requests, out of which h percent is hot-spot access and the remaining portion is evenly distributed over all outputs, including the hot-spot one. The hot-spot output now receives Nrh hot-spot requests and $r(1-h)$ regular requests during a cycle. If an output is assumed to be able to take at most one request during a cycle, then, every input is saturated at the input rate of $r = 1/(1 + h(N-1))$, yielding the maximum effective network bandwidth equal to $N/(1 + h(N-1))$.

This result explains what happens in MIN's with queues to hold conflicting requests [24]. If every input generates requests at the rate of $r = 1/(1 + h(N-1))$, the queue associated with the output port connected directly to the hot-spot location becomes full first. The two queues immediately before the filled queue then become full, and in turn the four prior queues are filled, and so on. This effect propagates all the way back to the input side and saturates all queues in a binary tree of switches rooted at the hot spot and extending to all the inputs. This situation is known as tree saturation [9], which blocks hot-spot requests and regular requests alike. Once developed, tree saturation leads to severely degraded network performance. For example, consider the case of a network sized 1000, where each input contributes a hot-spot rate of 1% (i.e., $h = 0.01$). From the above result, we find that effective network bandwidth is now limited to less than 100, i.e., 10% efficiency.

It should be noticed that the above result is due to two simultaneous assumptions: 1) all inputs are involved in hot-spot contention and 2) blocked requests are held in queues (rather than dropped). In the case where the two assumptions do not hold simultaneously, access to the shared hot-spot location would lead to lighter performance degradation. If only a portion of the inputs contributes to hot-spot access, for example, the degree of potential degradation is reduced, as illustrated in [10], [15]. This is because a saturated tree now has less impacts on regular requests issued from inputs which are not involved in hot-spot access. Similarly, if there is no queue in the network and blocked requests are simply dropped, a better result can be expected because no tree saturation would then arise, as shown in the study of [15]. However, a MIN without queues tends to have fairly low bandwidth and may suffer from considerable overhead due to retransmission [24]. In this paper, we focus only on situations where the two mentioned assumptions are simultaneously present, as they lead to the worst possible scenario.

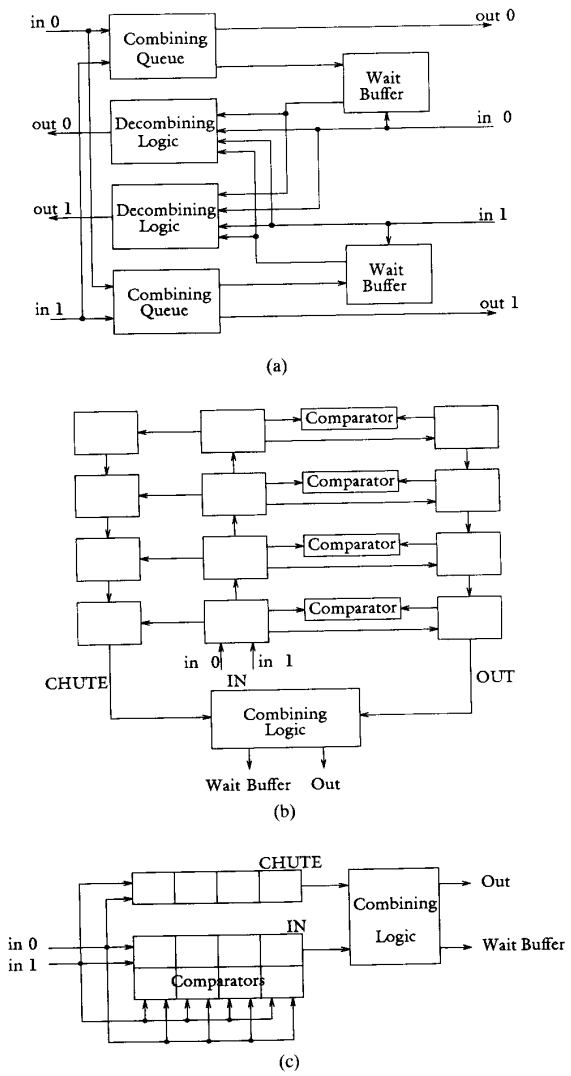


Fig. 1. (a) The block diagram of 2 × 2 combining switches. (b) A systolic design of the combining queue. (c) Another design of the combining queue.

B. Previous Solution

To prevent hot-spot access from causing tree saturation, networks with the combining capability have been suggested [9], [12], [13]. As hot-spot requests (frequently due to concurrent loads, stores, or fetch-and-op's [6]) directed to the same memory location meet at a switch, they are combined to reduce the number of concurrent references to the shared location. Combined requests themselves can be combined again in successive stages. The block diagram of a typical combining switch designed for the IBM RP3 and the NYU Ultracomputer is shown in Fig. 1(a) [13], [18]. There are two combining queues, two wait buffers, and two de-combining logics for a 2 × 2 switch.

The combining queue may follow either a systolic design where an incoming request is compared to one request in the queue at a time, as depicted in Fig. 1(b) [18], or a design where an incoming request is compared simultaneously to all requests held in the queue, as given in Fig. 1(c) [13]. Both designs contain an IN column and a CHUTE column (while the systolic design contains an additional column, the OUT column). Requests enter the IN column sequentially, and on entering the column, a request is compared to the request in the adjacent slot in the OUT column (if the slot is filled) for the systolic design, or compared to all the requests held in the IN column for the other design. If a match is found, the newly entered request is shunted over to the CHUTE column, where it proceeds downwards synchronously with the corresponding (i.e., matched) request; the two requests will leave the two columns at the same time.

A pair of matched requests are sent to the combining logic, where the two requests are combined into a single one (which is then forwarded to the output port), and where appropriate information to allow later de-combining is generated and forwarded to the wait buffer. When the reply of a request is back, the wait buffer associated with the port at which it arrives is searched. If there is a match, de-combining is performed to produce the reply of the request previously being stored in the wait buffer. Like the combining logic, the de-combining logic may be very complicated.

III. Proposed Combining Structure

The principal attribute of the proposed combining scheme lies in separating the combining function from the routing function. The combining function is accomplished by a combining tree, as shown in Fig. 2(a). Each element in the combining tree has two inputs (or one input) and one output (or two outputs) along the forward (or return) direction. The block diagram of combining elements is illustrated in Fig. 2(b): there is one combining queue, one wait buffer, and one de-combining logic. The "component count" in this combining element is reduced by one half when compared with that given in Fig. 1(a). Furthermore, the total number of combining elements in the combining tree with size N is $O(1/\log_2 N)$ of that in the previous combining network of the same size.

There are two major components in the novel combining structure, namely, a combining tree (called the combining section, or CS for short) and a regular MIN (called the routing section, or RS for short). In addition to delivering nonhot-spot requests, the regular MIN also serves to group hot-spot requests before combining and to route them to their destinations after combining. Fig. 3 gives the abstract model of the overall combining structure for hot-spot requests. In Fig. 3(a), hot-spot traffic is directed to CS before entering RS; while in Fig. 3(b), it is directed to CS after traversing i stages in RS. Hot-spot requests, after leaving CS, are sent back to RS over which they are directed in the same way as regular requests to reach their destinations. It is assumed that regular requests and hot-spot requests can be differentiated by examining (a part of) their op-codes. Unlike hot-spot requests, regular requests would never enter CS and travel merely through RS.

Provision is made in the regular MIN to allow hot-spot requests to be taken out for combining at *any* stage of the

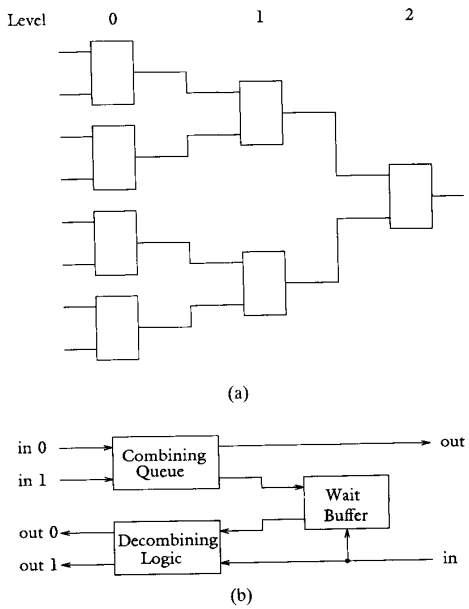


Fig. 2. (a) A combining tree with size 8. (b) The block diagram of combining elements.

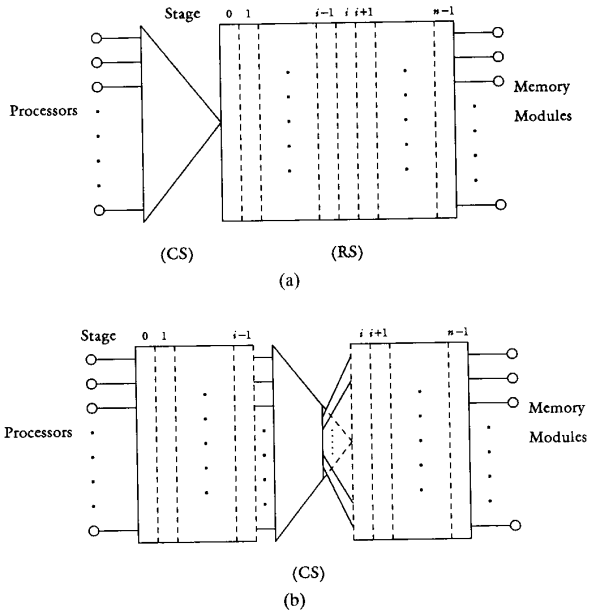


Fig. 3. The abstract model of the overall combining structure for hot-spot requests.

RS and to be injected back to RS from *any* level of CS. To this end, simple control switches, multiplexers/demultiplexers, and 1×2 demultiplexers are added to couple RS and CS together, as sketched in Fig. 4. Specifically, a simple control switch is incorporated in each input link of RS and a 1×2 multiplexer/demultiplexer is placed at each output link of CS, with a column of multiplexers/demultiplexers connecting RS and CS. The column of multiplexers (or demultiplexers) is

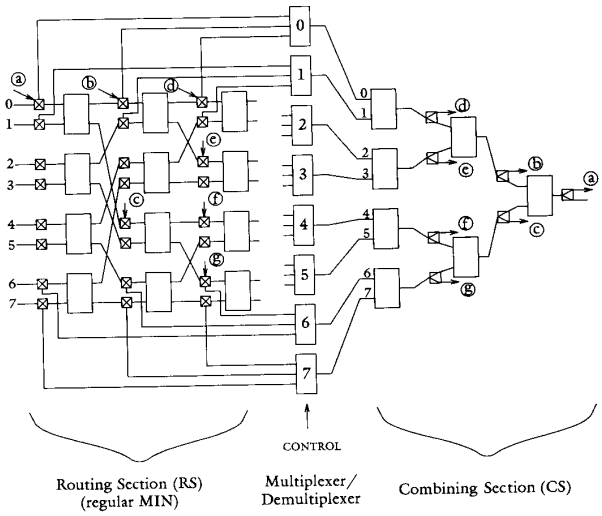


Fig. 4. The overall combining structure.

employed for the forward (or return) traffic flow, i.e., from the processor side to the memory side (or the opposite direction).

To facilitate the following explanation, let us assume that RS is a Baseline network [2]. The use of another network topology as RS can be explained similarly. A network of size $N(= 2^n)$ has n stages, which are numbered from 0 to $(n-1)$ starting with the leftmost stage. Input links connected to switching elements in a stage are numbered from 0 to $(N-1)$, with the topmost being 0. If all the input links to stage k in the Baseline network are partitioned into 2^k equally sized groups, numbered from 0 to $(2^k - 1)$, starting with the topmost one, then, requests arriving at an input port of group g must have addresses ranging from $g \times (N/2^k)$ to $(g+1) \times (N/2^k) - 1$ [2]. Let the levels of the combining tree be numbered from 0 to $(n-1)$, starting with the input end; and the output links of level l be numbered from 0 to $2^{(n-1-l)}$, starting with the topmost one.

A column of control switches in RS is controlled by one signal, so is a column of demultiplexers in CS. These control switches and demultiplexers make it possible to reconfigure this combining structure, according to the number of *concurrent hot-spots* (which refer to hot-spot locations that exist simultaneously), such that hot-spot requests are directed to the combining tree for combining after traversing, say k $(0 \leq k < n)$, stages in RS (guided by the k high-order tag bits [1]), and then the combined requests are injected back into RS from stage k after traversing $(n-k)$ levels in CS (as explained in the next paragraph). After being injected back to RS, combined requests are routed toward their destined hot-spot locations under the control of the remaining $(n-k)$ low-order tag bits.

Hot-spot requests in RS have been split into 2^k groups according to their addresses after passing through k stages; so a request in a group has the opportunity to combine only with other requests in the *same* group, not in another group. Thus, hot-spot requests which are taken out of RS after traversing k stages, $0 \leq k < n$, need to utilize only the first $(n-k)$

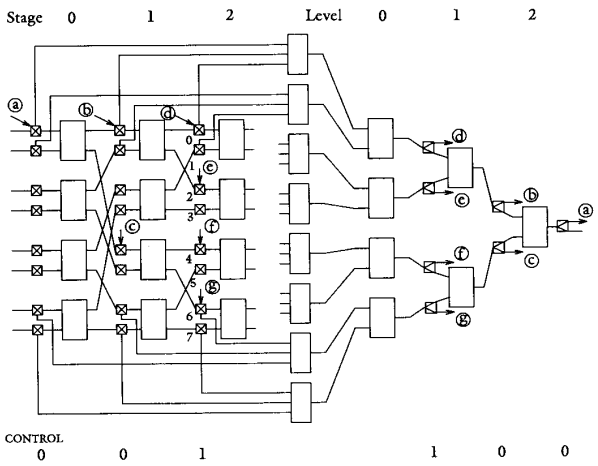


Fig. 5. A set of control signals for RS to let traffic flow from RS to CS via control switches 0, 2, 4, and 6 lying between stage 1 and stage 2. Traffic is fed back to RS from CS through d, e, f, and g.

levels of CS for combining. The combining tree now can be envisioned as an entity consisting of 2^k independent subtrees, each starting from level 0 to level $(n-1-k)$. Hot-spot requests are fed to these 2^k independent combining subtrees, one for a group, for combining.

Hot-spot requests are taken out of RS for combining at a stage determined by the number of concurrent hot-spot locations (more about this will be discussed in the next section). If we attempt to direct hot-spot traffic out of RS after it traverses k stages, then, among the n signals required for control switches in RS, all but the kth signal are set to make hot-spot traffic flow directly to the next stage, with the kth signal controlling hot-spot traffic to go to the combining tree. It should be noted that regular requests are never directed away from RS, as they are distinguishable from hot-spot ones.

The combining tree also needs n control signals. If hot-spot requests are taken out from RS after traversing k stages, traffic is controlled to flow back to RS after traversing $(n-k)$ levels in the combining tree. Suppose that control signal "1" directs hot-spot requests off RS or CS, then, control signals for the case where hot-spot traffic is directed to the combining tree after passing through two stages of RS are demonstrated in Fig. 5. The stream of combined requests from the pth output link of level l, $0 \leq l \leq (n-1)$, in CS can be put back to RS through an *arbitrary* input link of group p in stage $(n-1-l)$. An example of control signals for connections from CS to RS with $l = 0$ is given in Fig. 5, where connections d, e, f, and g are enabled. A combined request, after being put back to RS, proceeds toward its desired destination through the path established by making use of the remaining unused tag bits.

The design of control switches is simple. A $1 \times n$ demultiplexer in fact contains n AND gates; whereas an $n \times 1$ multiplexer can be either an OR gate or implemented by "wiring" the n inputs together. Every $1 \times n$ demultiplexer residing between RS and CS shares the same set of control signals for CS. Control signals for reconfiguring the combining structure may be fed to control switches and demultiplexers from an entity, e.g., a scheduling processor (which, according to the number of concurrent hot-spot locations, decides the control setting).

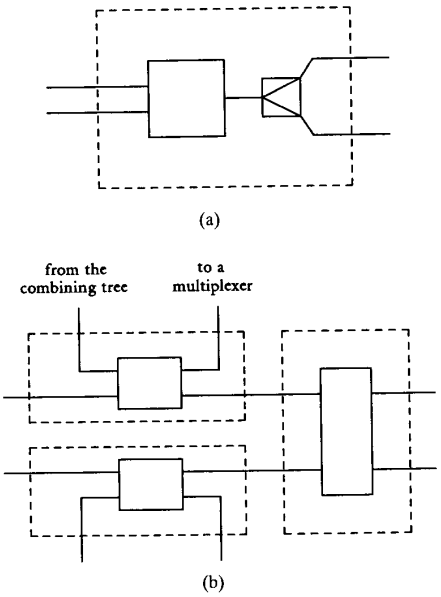


Fig. 6. Possible packages for components.

Notice that different applications generate different communication patterns, and may give rise to different numbers of concurrent hot-spots. The combining structure can be reconfigured to best suit various applications. If the number of concurrent hot-spots does not vary considerably (say, within; ξ and 10ξ for an integer ξ) during the course of execution (practically, this is often the case), then the configuration of the combining structure can be kept unchanged, else the structure has to be reconfigured.

The demultiplexer placed at the output of a combining switch in CS is regarded as a part of the combining switch, as illustrated in Fig. 6(a). This way of packaging components eliminates the off-chip delay between the demultiplexer and the switch (the pin count of a package is about equivalent to that of a 2×2 combining switch). On the other hand, the upper control switch in front of each switching element in RS is packaged separately from the switching element itself (so as not to increase the pin count much), as shown in Fig. 6(b). (Notice that if the wire "from the combining tree" terminates at the lower control switch instead, we then package the lower control switch separately.) Although the nonnegligible off-chip delay tends to slow down the operating speed of RS, the overall speed of the whole structure would probably be unaffected because its speed limit lies in CS not in RS.

IV. Discussion of the Structure

In this section, we first show that the proposed combining structure actually has *no limit on the number of allowable concurrent hot-spots*; it can effectively handle as many concurrent hot-spots as an application might have. We then discuss the limitations of the combining structure and explain how

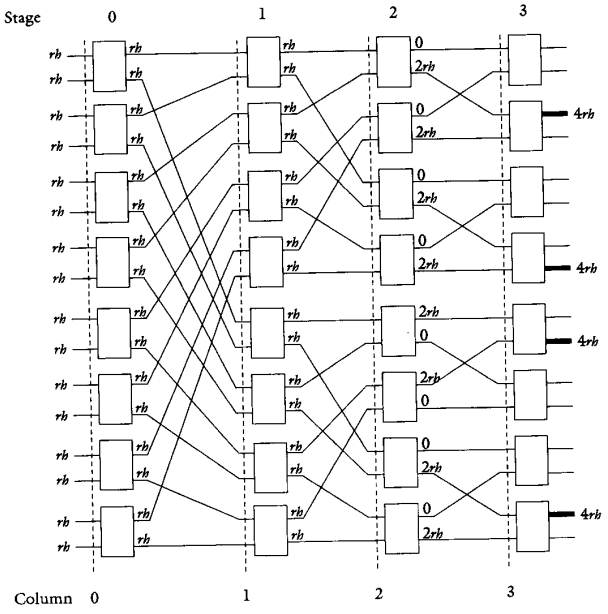


Fig. 7. The request rate on every link of a MIN where traffic is heading for four hot-spots.

the mean latency of hot-spot requests is affected by the setup of the structure. Finding a setup that yields low mean latency is essential and is one of our major concerns, because a low latency indicates that the combining structure can effectively manage hot-spot contention, keeping the overall system execution flow fluent.

Consider a combining structure with size $N(= 2^n)$. Every column of control switches in RS is labeled by a number i, $0 \leq i \leq n-1$, with the leftmost column being 0; so is every column of multiplexers/demultiplexers in CS. Let k be the column (in RS) from which hot-spot traffic is directed away and m be the number of concurrent hot-spot locations. To simplify our explanation, we assume m to be a power of 2, i.e., $m = 2^\tau$, where $0 \leq \tau \leq n$.

Suppose that a hot-spot request has an *equal probability* to address any one of the m concurrent hot-spot locations. Let us also assume that hot-spot locations are *evenly distributed* over memory modules, namely, one hot location is placed at each group of N/m consecutive memory modules (see Fig. 7 for an example, where a bold output denotes a hot-spot). This may be achieved (to a close degree) with the help of compilers and operating systems. Requests heading for a full queue cannot proceed and remain in their original locations.

A. Allowable Concurrent Hot-Spots

Consider the case that hot-spot traffic is directed away from RS at control column $k \leq \tau$. The request rate at a link (in RS) which lies before control column k can be shown to be always no larger than the input hot-spot rate (which is less than 1). This is because requests from an input of a switching element lying before the column head for the two outputs with an equal probability, since hot-spot locations are evenly distributed over memory modules. As depicted in Fig. 7, where there are four concurrent hot-spots $((0010)_2, (0111)_2, (1001)_2$, and $(1110)_2)$, the request rate on every link lying before control column 2 is the same and is equal to the input hot-spot rate.

Thus, if $k \leq \tau$, it is impossible to develop tree saturation because no link in RS would become saturated. After traversing k stages in RS, hot-spot requests are sorted into 2^k groups before being combined. For an application where m is large, we can choose a big k such that requests within each group still address one or a limited number of concurrent hot-spot locations, suggesting that this combining structure can effectively handle an arbitrary number of concurrent hot-spots.

B. Limitations to the Structure

A basic assumption for this structure to achieve satisfactory performance is that hot-spot requests can be differentiated from regular requests at run time. It is recognized that the sources of hot-spot traffic vary widely, including heavy access of specific variables for synchronization or resource management purposes, massive references of a certain location temporarily (due to, for example, fetching a constant during initialization), unusual array reference patterns resulting in unproportionally frequent access to several locations within a memory module, etc. Among these sources, the first one is perhaps the most critical because it causes high references to some fixed locations relatively persistently. These references are often made by utilizing a set of specific primitive instructions (e.g., fetch-and-op's [6] and others [21]). They may be told dynamically by examining (a part of) the instruction op-code or tag (if a specific tag exists in the instruction). The use of this combining structure is limited to eliminating hot-spot contention due to this source, and the structure does not alleviate contention caused by other sources which give rise to references that cannot be identified at run time. It should be noted that the earlier combining network may also fail to avoid hot-spot contention, when hot-spot references are made to several locations (rather than a single location) within a memory module.

For programs that are to be executed simultaneously on the processors interconnected by this structure, they had better involve roughly the same order of concurrent hot-spot numbers (say, between ξ and 10ξ for an integer ξ, as mentioned earlier); otherwise, the reconfigured combining structure would possibly deliver disappointed performance. This implies that the proposed combining structure may reduce the freedom of task scheduling or lose some parallelism during program execution. A compiler is responsible for estimating the number of concurrent hot-spots associated with each program. It is likely that a reasonably accurate estimate of the number due to synchronization or resource management references can be made. For example, a parallel loop requires barrier synchronization whose implementation may use two counters, both of which become hot locations.

When new jobs arrive, the combining structure may need to be reconfigured so that good performance can be ascertained. From the implementation standpoint, complex (and somewhat

difficult) wiring between RS and CS as well as additional control lines to all system components for the setup purpose tend to increase the cost of this proposed structure, lessening the potential gain obtainable from the use of the simple combining tree. While it cannot directly replace the combining network suggested for the IBM RP3 and the NYU Ultracomputer, this combining structure could be profitable in some applications.

C. Structure Setup

In order to keep the latency of hot-spot requests low, we assume that a dedicated queue with high priority for hot-spot requests is introduced in every output port of RS, namely, each output port is equipped with two queues [19]—one for regular requests and the other (with high priority) for hot-spot requests. Hot-spot requests thus have priority over regular ones in RS, and they compete only with hot-spot requests. Under this buffering organization, we need to consider only hot-spot requests, since hot-spot requests are then free from the interference of regular requests. It should be noted that the combining structure without the added dedicated queues would also work appropriately, but its performance may be somewhat degraded. To validate this point, the simulation results of the combining structure where only a single queue is incorporated in each output port are also included and discussed in the next section.

We consider the relationship between the structure setup and the mean latency of hot-spot requests. Let RS and CS have the same clock rate. During a cycle, a conflict-free request can move from one stage (or level) to the next stage (or level). The total latency (L) of a hot-spot request from the input end to the output end of the combining structure comprises three components (when traffic is directed away from RS for combining at control column k): the latency to traverse the first k stages of RS, the latency in CS, and the latency to traverse the remaining $(n-k)$ stages of RS, which are denoted respectively by L_1, L_2, and L_3. The latency is measured in terms of cycles. Since combined requests never experience extra delays in traversing the remaining $(n-k)$ stages of RS, L_3 is $(n-k)$. The number of levels traversed by a hot-spot request in CS is $(n-k)$. The mean latency of requests at every level in CS is different and is dependent upon the mean length of the combining queue as well as the number of concurrent hot-spots in a group.

As k increases from 0 to τ (where $2^\tau = m$ is the total number of concurrent hot-spots), each group involves fewer and fewer concurrent hot-spots, implying that hot-spot requests experience less and less delay in CS. Thus, L_2 gets reduced, as does L_3, when k increases. On the other hand, L_1 becomes larger as k grows. The growth rate in L_1 is well offset by the reduction in L_2 and L_3 combined, if k is less than or equal to τ. A net reduction in latency still results from further increasing k, provided that tree saturation is not developed in RS and the increment in L_1 is slower than the decrement in L_2 plus L_3 (a too large k leads to tree saturation and, thus, a dramatic increase in L_1). Theoretically, it is possible to derive an optimal k that yields the least mean latency, for any given hot-spot rate and concurrent hot-spot number, as provided in [25]. In practical situations, the hot-spot rate and concurrent hot-spot number for a set of jobs to be executed simultaneously may not be exactly the same, indicating that an optimal setup is not always achievable. Fortunately, the structure with an appropriately chosen k can offer mean latency close to the lowest value for a reasonable range of hot-spot rates and concurrent hot-spot numbers, as will be seen in our simulation results given in the next section. This means that in practice, a setup which supports multiple jobs with different hot-spot rates and concurrent hot-spot numbers can be found, and such a setup is desirable in multiprogramming environments.

V. Simulation Results

We ran simulations to examine the performance of the proposed combining structure with different sets of control signals under a variety of hot-spot rates (rh) and concurrent hot-spot numbers (m). The simulated structure operates in a packet switching mode, with every request involving only one packet. The performance measure of interest is the mean latency of hot-spot requests in the forward direction, L (in terms of cycles). If a setup of the structure can effectively handle hot-spot access, L is small; otherwise, L becomes unacceptably large. The latency of a request is counted from the time when the request is generated (rather than when it is admitted to the structure) until it leaves the structure. The following assumptions are made in our simulation:

1) RS and CS operate at the same speed, i.e., during a cycle, a request, without conflicting with other requests, can travel from one stage (or level) to another in RS (or CS);
2) A queue capable of accommodating 2 requests is associated with each output port of switching elements in RS;
3) A combining queue of depth 4 [which means that it can hold up to 4 requests in the IN column, see Fig. 1(c)] is incorporated in a combining element;
4) A request, on entering an IN column, is compared simultaneously with all requests held in the column;
5) Before leaving a combining element, a combined request can combine with another combinable request in the same element;
6) A hot-spot request has the same probability of addressing any one of all the concurrent hot-spot locations, in a random and independent manner. Each hot-spot location is placed at a separate group of memory modules;
7) An output queue in RS can accept up to 2 requests during a cycle; so can a combining queue in CS. If a request heads for a full queue, it cannot proceed and remains in its original location. In case a conflict arises and only one slot is left in a queue, a randomly chosen request proceeds to the queue, with the other one kept at its original location; and
8) During a cycle, one request in a queue at the last stage of RS is removed from the structure.

First, a high priority queue dedicated to hot-spot requests is assumed to exist in every output port of RS, and the simulation results are provided in Figs. 8, 9, and 10. Then, hot-spot

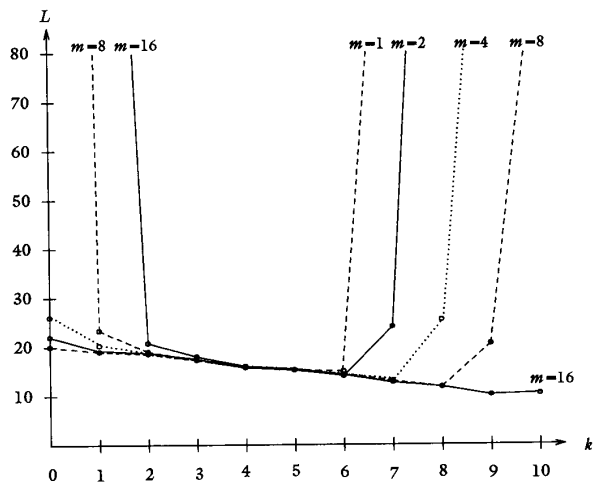


Fig. 8. Mean latency (L) versus the column (k) where traffic is directed away from RS under various concurrent hot-spot numbers (m). ($N = 1024, rh = 1\%$).

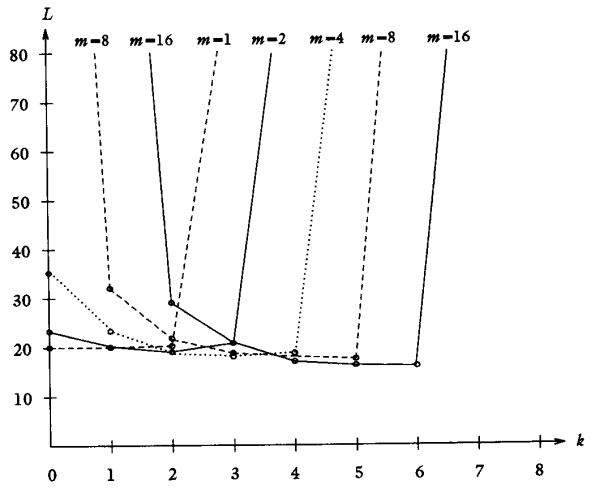


Fig. 9. Mean latency (L) versus the column (k) where traffic is directed away from RS under various concurrent hot-spot numbers (m). ($N = 1024, rh = 16\%$).

requests are assumed to share a single queue with regular requests at every output port of RS, with the simulation results depicted in Figs. 11 and 12. All the points given in the figures represent steady state results.

In Fig. 8, the mean latency (L) versus k is plotted for concurrent hot-spot numbers ranging from 1 to 16 under the case of N (system size) = 1024 and rh (hot-spot rate) = 1%. It can be seen that L is unacceptably high for $k < 2$ as m is 16, because in that case, CS is congested. On the other hand, as k is larger than 6 for the case of $m = 1$, L starts to increase drastically since RS then gets saturated. Each m has a range of k where L is low.

It is interesting to notice the case of $k = 0$. L is low only when m is no larger than 4. This is because the combining queue is assumed to have capacity 4. The curve for $m = 16$ goes lower for a larger k and reaches its lowest point when $k = 10$. This suggests that when m is greater than or equal to 16, there is no memory module really "hot" and the combining tree is thus unnecessary, under the situation of $N = 1024$ and $rh = 1\%$.

Fig. 9 is for the case of a high hot-spot rate ($rh = 16\%$). The k value which yields the lowest L for each m is different from that in Fig. 8, but every curve still has a range of k where L stays low.

Interestingly, we may also observe from Figs. 8 and 9 that, for any given hot-spot rate, an appropriate value of k can be determined that leads to reasonably good performance over several different concurrent hot-spot numbers, m. When rh is 1%, for example, choosing $k = 5$ or 6 would give rise to the lowest or near-lowest mean latency, for m ranging from 1 to 16 (and more) in a system with size 1024. Likewise, setting $k = 2$ guarantees good performance when m is between 1 and 8, for $rh = 16\%$. As a result, it is possible to select a proper k even when the number of concurrent hot-spots is not fixed but varies within a reasonable range. For practical applications (when rh is 10% or less), the range is probably wide enough to accommodate the variation of m during the course of typical parallel program execution, indicating that the combining structure could maintain satisfactory performance even without changing its configuration during the entire period of program execution. The selected proper k usually is the value which yields the least L (called k_{opt}) for such a concurrent hot-spot number that is at the lower side of the range of variation. For example, given $1 \leq m \leq 16$ and $rh = 1\%$, k is selected to be k_{opt} for $m = 1$ or 2, which is 5 or 6.

L as a function of k for various hot-spot rates is depicted in Fig. 10, where the network size is 1024 and the number of concurrent hot-spots is 4. It can be seen that the network starts to saturate earlier (i.e., at a smaller k), as the hot-spot rate grows. Every curve value is bounded to an acceptable L at k = 0, implying that the combining tree can effectively remove hot-spot contention when the capacity of combining queues equals the number of concurrent hot-spots. An appropriate k may be identified for a given m and a range of rh. According to Fig. 10, for example, picking k = 3 results in the least or near-least L when rh lies between 1% and 16%.

The above simulation results are for systems where a dedicated higher-priority queue for hot-spot requests exists at every output port of RS. In order to understand the degree of possible performance degradation without having such dedicated queues, we also simulate the combining structure where each RS output port has only one single queue, which is shared by hot-spot and regular requests. The simulated mean latency of hot-spot requests for $rh = 1\%$ and $N = 1024$ under the arrival rate of $r = 0.4$, a moderate traffic situation, is illustrated in Fig. 11; whereas the mean latency of regular requests is given in Fig. 12.

Because of the sophisticated interference between regular and hot-spot requests, the curves in Fig. 11 no longer stay in an unmixed order dictated by rh, as in Figs. 8–10. While the regular requests exhibit little degraded mean latency as

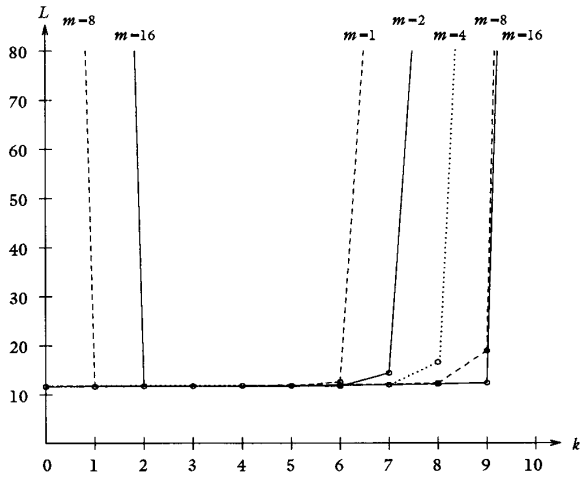


Fig. 10. Mean latency (L) versus the column (k) where traffic is directed away from RS under various concurrent hot-spot rates (rh). ($N = 1024, m = 4$).

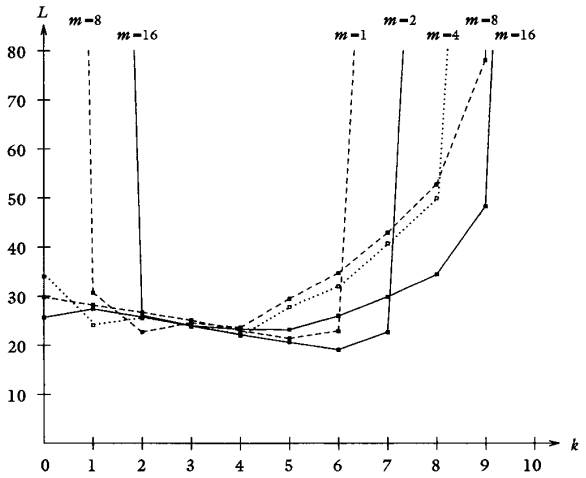


Fig. 11. Mean latency (L) of hot-spot requests versus the column (k) where traffic is directed away from RS under various hot-spot numbers (m). ($N = 1024, r = 0.4, rh = 1\%$).

a result of this simple buffer design (unless saturated trees have developed), the hot-spot requests consistently suffer from noticeable performance degradation. This is because hot-spot requests now compete with regular requests in RS, and when a combined request experiences one additional unit delay, each request represented by the combined request (stored in the wait buffer) also experiences so, making the mean latency of hot-spot requests increased noticeably. On the other hand, only a small fraction of regular requests (in this case, less than $h = 2.5\%$ of them) encounters combined requests on the way to their destinations and may suffer from extra delays, leading to a negligible increase in the mean latency of all regular requests, provided that tree saturation does not exist. It is expected that the degradation of hot-spot requests would be less under a lighter traffic situation, and becomes significant

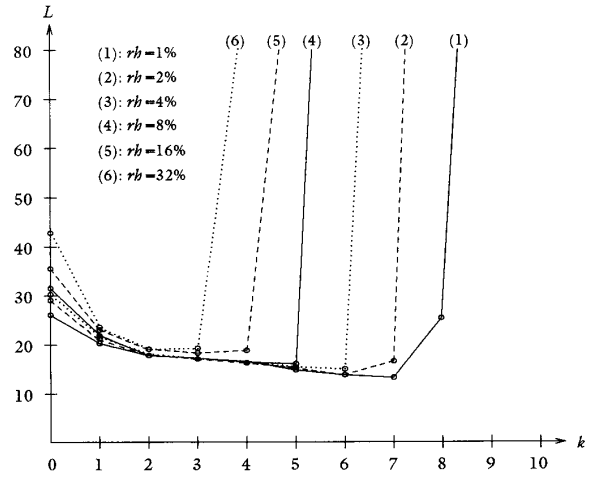


Fig. 12. Mean latency (L) of regular requests versus the column (k) where traffic is directed away from RS under various hot-spot numbers (m). ($N = 1024, r = 0.4, rh = 1\%$).

when traffic approaches the queueing knee (which is near half saturation). Under a moderate to heavy traffic load, it appears clearly advantageous to incorporate a separate high priority queue in every output port for hot-spot requests so that the mean latency of hot-spot requests can be kept low.

From the simulation results, we are sure that the proposed combining structure can prevent hot-spot contention as effectively as a combining network suggested for the IBM RP3 and the NYU Ultracomputer under practical situations, despite its lower hardware complexity.

VI. Concluding Remarks

A cost-effective combining structure has been proposed for eliminating hot-spot contention caused by concurrent access to shared locations in multiprocessors. The key concept of this design involves separating the combining function from the routing function, making it possible to use a binary tree-based combining configuration. As a result, the constituent combining element is simple, and the total number of required elements is reduced to $O(1/\log_2 N)$ of that in the previously known combining network, where N is the network size. The proposed combining structure can handle any arbitrary number of concurrent hot-spots, but removes the adverse impacts only due to hot-spot references which are distinguishable at run time. While it does not directly replace the earlier combining network, the proposed structure could profitably apply to certain applications.

Simulation results of the proposed structure under various setups are provided. It is discovered from the results that with appropriate setups, the structure can effectively eliminate hot-spot contention. More importantly, the simulation results indicate that a near-optimal reconfigured structure can be determined even when m is not fixed but varies over a reasonable range during the course of executing typical parallel programs, for a given rh. Similarly, it is also possible to identify a suitable reconfiguration that achieves good perfor-

mance as rh fluctuates over a practical range, for any m. Consequently, the proposed combining structure may be able to deliver satisfactory performance even without changing its configuration for the entire course of program executions.

A slightly simpler structure may be used for applications where the number of concurrent hot-spot locations is small. In this simplified structure, a separate "demultiplexer tree" serves as RS to perform the routing function, and its input is connected to the output of the combining tree. Hot-spot requests are directed to the combining tree immediately after they are generated, and combined requests are sent to the "demultiplexer tree" after leaving the combining tree in order to reach their proper destinations; hot-spot requests would never enter or use the regular MIN (which is then devoted to regular requests only). This simplified combining structure performs well when the number of concurrent hot-spots is less than a certain limit (say, 3 or 4), but its performance degrades significantly when the number exceeds that limit.

Since the combining element is more complex than a regular switching element, it is essential to incorporate fault-tolerance into the combining tree in order to meet the overall reliability requirement. Being a tree structure, the new combining design can be made fault-tolerant at a little additional cost by utilizing approaches similar to X-tree [29], SOFT [30], or others. It would be interesting to investigate an effective fault-tolerant technique for the proposed combining structure.

REFERENCES

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

[2] 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.

[3] H. Sullivan, T. Bashkow, and D. Klappholtz, "A large scale homogeneous, fully distributed parallel machine," in *Proc. 4th Symp. Comput. Architecture*, June 1977, pp. 105–124.

[4] H. J. Siegel *et al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934–947, Dec. 1981.

[5] D. D. Gajski *et al.*, "Cedar—A large scale multiprocessor," in *Proc. 1983 Int. Conf. Parallel Processing*, Aug. 1983, pp. 524–529.

[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] W. Crowther *et al.*, "Performance measurements on a 128-node butterfly parallel processor," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 531–540.

[8] 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.

[9] 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.

[10] R. Lee, "On hot spot contention," *ACM SIG Computer Architecture News*, vol. 13, pp. 15–20, Dec. 1985.

[11] N. M. Patel and P. G. Harrison, "On hot-spot contention in interconnection networks," in *Proc. 1988 ACM SIGMETRICS Conf.*, May 1988, pp. 114–123.

[12] S. Dickey, R. Kenner, and M. Snir, "An implementation of a combining network for the NYU Ultracomputer," New York Univ., Ultracomputer Research Laboratory, Ultracomputer Note #93, Jan. 1986.

[13] M. C. Wong, "A combining omega network: Performance vs implementation," IBM Res. Rep., RC 11977, June 1986.

[14] 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.

[15] R. H. Thomas, "Behavior of the butterfly parallel processor in the presence of memory hot spots," in *Proc. 1986 Int. Conf. Parallel Processing*, Aug. 1986, pp. 46–50.

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

[17] G. Lee, "A performance bound of multistage combining networks," *IEEE Trans. Comput.*, vol. C-38, pp. 1387–1395, Oct. 1989.

[18] S. Dickey *et al.*, "A VLSI combining network for the NYU Ultracomputer," in *Proc. Int. Conf. Comput. Design*, Oct. 1985, pp. 110–113.

[19] Y. Tamir and G. L. Frazier, "Support for high-priority traffic in VLSI communication switches," in *Proc. 9th Symp. Real-Time Syst.*, Dec. 1988, pp. 191–200.

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

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

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

[23] B. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Proc. SPIE, Real Time Processing IV*, 1981, pp. 241–248.

[24] 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.

[25] N.-F. Tzeng, "Design of a novel combining structure for shared-memory multiprocessors," in *Proc. 1989 Int. Conf. Parallel Processing*, Aug. 1989, pp. I 1–8.

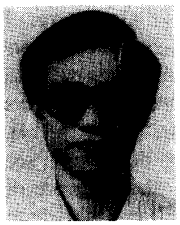
[26] G. J. Lipovski and P. Vaughan, "A fetch-and-op implementation for parallel computers," in *Proc. 15th Symp. Comput. Architecture*, May 1988, pp. 384–392.

[27] L. J. Guibas and F. M. Liang, "Systolic stacks, queues, and counters," in *Proc. Conf. Advanced Research VLSI*, Jan. 1982.

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

[29] A. M. Despain and D. A. Patterson, "X-Tree: A tree structured multiprocessor computer architecture," in *Proc. 5th Annu. Symp. Comput. Architecture*, Apr. 1978, pp. 144–150.

[30] M. B. Lowrie and W. K. Fuchs, "Reconfigurable tree architectures using Subtree Oriented Fault Tolerance," *IEEE Trans. Comput.*, vol. C-37, pp. 1172–1182, Oct. 1987.



Nian-Feng Tzeng (S'85–M'86–SM'92) 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 at Urbana–Champaign in 1978, 1980, and 1986, respectively.

He is currently an Associate Professor in the Center for Advanced Computer Studies at the University of Southwestern Louisiana, Lafayette. From 1986 to 1987, he was a Member of Technical Staff, AT&T Bell Laboratories, Columbus, OH. He was involved in the Cedar supercomputer project at the Center for Supercomputing Research and Development, University of Illinois, Urbana for more than two years. His current research interest is in the areas of parallel and distributed processing, fault-tolerant computing, and VLSI/WSI-based systems.

Dr. Tzeng is a member of Tau Beta Pi, a member of the Association for Computing Machinery, and the recipient of the outstanding paper award of the 10th International Conference on Distributed Computing Systems, May 1990.