The Performance of the Cedar Multistage Switching Network[1]

Josep Torrellas and **Zheng Zhang**
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801

Abstract

While multistage switching networks for vector multiprocessors have been studied extensively, detailed evaluations of their performance are rare. Indeed, analytical models, simulations with pseudo-synthetic loads, studies focused on average-value parameters, and measurements of networks disconnected from the machine, all provide limited information. In this paper, instead, we present an in-depth empirical analysis of a multistage switching network in a realistic setting: we use hardware probes to examine the performance of the omega network of the Cedar shared-memory machine executing real applications. The machine is configured with 16 vector processors.

The analysis suggests that the performance of multistage switching networks is limited by traffic non-uniformities. We identify two major non-uniformities that degrade Cedar's performance and are likely to slow down other networks too. The first one is the contention caused by the return messages in a vector access as they converge from the memories to one processor port. This traffic convergence penalizes vector reads and, more importantly, causes tree saturation. The second non-uniformity is the uneven contention delays induced by even a relatively fair scheme to resolve message collisions. Based on our observations, we argue that intuitive optimizations for multistage switching networks may not be cost-effective. Instead, we suggest changes to increase the network bandwidth at the root of the traffic convergence tree and to delay traffic convergence up until the final stages of the network.

1 Introduction

Few topics in computer systems have attracted more attention than multistage switching networks for multiprocessors [19]. One reason is their impact on machine performance. Indeed, processor communication is a fundamental yet intrinsically expensive activity; networks should support it efficiently. A second reason is the many network variations that are interesting. Each of them satisfies different cost/performance requirements, ranging from those of busses to crossbars. Finally, another reason is technology changes. Technology upgrades in networks have been as fast-paced as in other aspects of computers. The result of all this work is a vast body of knowledge.

It is known that the performance of computer systems is determined by bursty activities that are difficult to predict without detailed empirical observations. While networks are no exception to this rule, measuring such systems is costly and difficult. Therefore, many studies have been based on analytical models [4, 6, 11] or on simulations [5, 14, 15, 17, 18], usually using pseudo-synthetic loads. While these methods supply useful information, they do not provide the complete picture. Other studies, instead, have measured real networks while focusing on high-level average-value parameters like round trip time, bandwidth, or utilization [10]. Some other studies have been performed with the network disconnected from the machine. Again, the information provided is not complete. As a result, researchers still lack a complete picture of how multistage switching networks perform in real systems and how to compare their performance [13].

[1]This work was supported in part by the National Science Foundation under grants NSF RIA MIP 93-08098, NSF MIP 93-07910, and NSF MIP 89-20891; a NASA ICLASS grant; and grant 1-1-28028 from the Univ. of Illinois Research Board.

In this paper, we present an in-depth empirical analysis of a multistage switching network in a vector multiprocessor. We use hardware probes to monitor the omega network [8] of the Cedar shared-memory machine [7] executing real applications. The machine is configured with 16 CPUs. We examine each individual queue, switch element, and link in the network. The analysis suggests that the performance of multistage switching networks is limited by traffic non-uniformities. We identify two major non-uniformities that degrade Cedar's performance and are likely to slow down other networks too. The first one is the contention caused by the return messages in a vector access converging from the memories to one processor port. The traffic convergence leads to slower vector reads and, more importantly, tree saturation of the links. The second non-uniformity is the uneven contention delays induced by even a relatively fair scheme to resolve message collisions. This effect slows synchronization and vector reads. Based on our observations, we then discuss cost-effective optimizations for multistage switching networks.

This paper is organized in three main parts. First, we describe our experimental system (Section 2). Then, we characterize the performance of the network (Sections 3 and 4). Finally, we discuss optimizations (Section 5).

2 The Experimental System

In this section, we briefly describe Cedar, the experimental setup, and the workloads run.

2.1 The Cedar Machine and Its Network

In this research, we use a hardware performance monitor to examine the network of Cedar. Cedar is a shared-memory multiprocessor developed at the Center for Supercomputing Research and Development, University of Illinois [7]. The machine has 32 Alliant FX/8 vector processors. Limited performance monitoring hardware, however, forces us to disable 16 of them. Processors do not stall on writes; however, they stall with two pending scalar reads. Messages in a vector access are pipelined. All processors share 64 Mbytes of memory organized in 32 memory modules interleaved on a 64-bit word basis. In addition, processors have local memory where they store private data like loop index variables. The processors and memories are connected via a forward and a reverse omega network [8] (Figure 1). The cycle time of the networks is 85 ns. For our system, the total bandwidth of the processors and memories is equal: each processor can issue one read message to the network every two cycles, while memory modules have a 4-cycle cycle time.

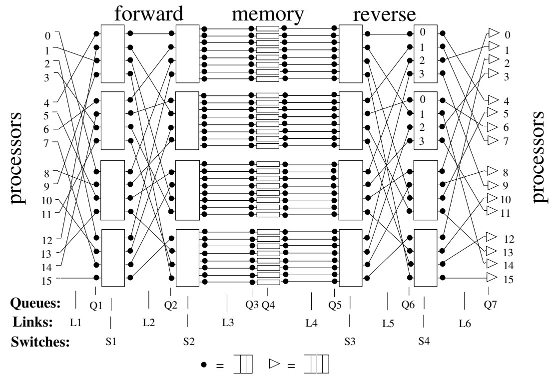

Figure 1: The Cedar network.

Each network in Figure 1 is synchronous, packet-switched and has 64-bit wide data paths. A read consists of one word in the forward network and two in the reverse. A write is two words in the forward network and one in the reverse. Without contention, a round trip takes 14 cycles, of which 4 are spent in memory. Each network has two stages of crossbar switches. Both switches and memory modules have 2-entry queues in the input and output ports (see figure). In addition, the interface between the reverse network and the processors has 3-entry queues. As seen in Figure 1, we refer to the queue stages as *Q1-Q7*, to the link stages as *L1-L6*, and to the switch stages as *S1-S4*. The queuing recorded at the unlabeled queues on the left side of the links is added to the queuing at the queues on the right side of the links. In the figure, queues *Q4* are used only when the memory output queue is full. If *Q4* is used, the corresponding memory module becomes unavailable. Queues *Q7* are needed because the processor interface module may not be able to accept messages every cycle. Every returning read needs to use an internal bus on the interface module for one cycle. However, the same bus is used for one cycle with higher priority by each write and scalar read issued by the processor.

Links *L1*, *L2*, *L5*, and *L6* have not been drawn in the familiar shuffle fashion to emphasize the handling of message collisions. If messages coming to a switch from different inputs request the same output, the following scheme is used. First, the switch records the requesting inputs creating what we call a "snapshot". Then, it services the inputs in the snapshot according to a fixed priority scheme before servicing any new input requesting the same output. In Figure 1, the inputs of the switches have been ordered top to bottom from higher to lower priority of service in a snapshot. As an example, we number the inputs of two switches; 0 denotes highest priority. Note that link connections are rotated so that, in most cases, a given processor-processor path is given different priorities in different switches. This makes the network more fair.

2.2 Experimental Setup

To perform our measurements, we use a trace-collecting hardware performance monitor [1]. The monitor has 16 trace buffers that can store over a million entries each. Each entry contains the address referenced, the time stamp, the type of reference (vector/scalar read or write, or test&set), and other information for a total of 64 bits. The trace buffers are controlled by a workstation and can be connected to different places in Cedar. We initially planned to connect them to the network links, but the high density of wires in the network prevented it. Instead, we connected the trace buffers to the 16 network interfaces, so that the buffers intercepted the messages as they enter the network. The resulting 16-processor traces are then fed to a cycle-by-cycle simulator of the network and memory. The simulator was designed based on the hardware schematics of the network and memory, which were built at CSRD. Since the network is deterministic, the simulation models exactly the behavior of the real machine.

Since the processors fill up the trace buffers in a few hundred ms, we built a feedback connection between the performance monitor and Cedar. When any of the buffers has less than 16,000 entries left, the performance monitor sends a non-maskable interrupt to the processors of Cedar. The interrupt forces all the processors to execute a trap handler that simply contains the "halt" instruction. This instruction freezes the processors. Then, the workstation controlling the performance monitor reads and post-processes the traces in the buffers, resets the buffers, and deactivates the interrupt. At that point, the Cedar processors leave the trap handler and restart execution of the application exactly where they left it. This sequence is repeated until the program being traced finishes. The only inaccuracies in the system are the several tens of references issued by the trap handler when it is invoked and the loss of network state when the program is suspended. Both are negligible. The latter is negligible because the network needs only 1.5 μs to be fully loaded with application traces again.

In Section 5 we evaluate the effect of changes to the Cedar network. To make the analysis accurate, we modify the Cedar traces as follows. When we run the simulator of the Cedar network, we compute, for each pair of consecutive references by a processor, the time the processor was computing as opposed to waiting for a reply. Such thinking times are dumped into a file as the simulation proceeds and reused later when we simulate the new network. For the new network, the latencies will be different, of course, but the thinking time will be the same. Given that our applications exploit loop-level parallelism, their parallel threads execute fairly deterministic paths. In addition, we also take into account the synchronization points. Therefore, the simulations in Section 5 are accurate.

Table 1: Applications traced. All applications operate on double precision data.

Applic.	Subrout. Traced	Vector Refs. (%)	Notes on Vector Accesses
Flo52	Psmoo	96.4	Length 8 often, rest 32. Only 50% with stride 1.
Trfd	Olda	97.6	Length varies from 1 to 32. Stride 1 and 40.
Ocean	Ftrvmt	91.6	Length 32 and 129. Stride 1.
Bdna	Actfor	95.7	Length 32 mostly, rest 4, 640. Stride 1 mostly.

2.3 Workloads

For our experiments, we use four real applications that are frequently run on the machine. The applications, shown in Table 1, are hand-parallelized versions of four Perfect

Table 2: Performance of the Cedar network. The figures are the average of 16 processors.

Application	Time Stalled while Request is in Net+Mem./ Total Exec. Time (%)	Time Avg. Global Memory Busy/ Total Exec. Time (%)	Time Avg. *L1* Link Transfers Data/ Total Exec. Time (%)	Network Latency (Cycles)	Contention Delay/ Network Latency (%)
Flo52	22.4	13.5	11.7	17.2	18.7
Trfd	17.6	24.3	20.0	18.3	23.3
Ocean	14.6	19.4	12.7	21.5	35.0
Bdna	10.3	19.2	12.0	20.5	31.8

Club benchmarks [2] running on 16 processors. The versions used are highly vectorized and are close to the best that a parallelizing compiler can perform [3]. All the data are allocated in Cedar's shared memory, except for private loop index variables and other local variables, which are allocated in a private, faster memory closer to the processors.

Practical reasons limit the length of the code section that we can trace. Therefore, we profiled the execution of each application and isolated the routine that accounts for the largest fraction of the execution time. The name of the routine and some characteristics of its vector accesses are shown in Table 1. We then traced the 16-processor execution of the routine for over 1 second of real time. This time does not include the overhead of the stops and restarts that our setup forces on the applications. The period traced is long enough to exercise the Cedar network extensively.

3 General Characterization of the Network Performance

We start the analysis of the network's performance by examining some high-level parameters. In Table 2, the second column shows that processors are stalled for 10-22% of the execution time waiting for requests traversing the network and memory. While this is a significant amount of time, it does not include the majority of the cycles: execution time is also affected by accesses to Cedar's local memory and inefficiencies in the processor-network interface. The purpose of this paper, however, is to characterize and optimize this 10-22% of the execution time only.

The table also shows the network utilization. Column 3 shows that the average global memory module is busy only 14-24% of the cycles. Furthermore, the average link in the first stage of the network transfers data in only 12-20% of the cycles (Column 4). While these two figures suggest a lightly-loaded network, traffic burstiness changes the picture. Indeed, of the 17-22 cycles taken by the round trip of a message in the network and memory (Column 5), as much as 19-35% are the result of contention (Column 6). In the next section, we investigate the sources of these contention cycles.

4 Detailed Characterization of the Performance Bottlenecks

An analysis of the contention cycles reveals that the performance of the Cedar network is limited by two traffic non-uniformities: the contention that the messages in a vector access cause in the reverse network as they converge to one processor port, and the uneven contention delays induced by even a relatively fair scheme to resolve message collisions. Both problems are likely to be present in many other networks. In the following, we look at each problem and its consequences in turn.

4.1 Traffic Convergence in the Reverse Network Is a Bottleneck

To identify this problem, we examine two differences between the forward and reverse network traffic: message collisions and message queuing.

Messages in the Reverse Network Collide More

Experimental measurements show that messages in the reverse network collide more frequently than in the forward network. Figure 2 shows, for each switch output port, the fraction of messages that collide while accessing the port. Output ports for each switch stage *Si* are numbered from the top of Figure 1 to bottom. Therefore, all stages have 16 output ports except *S2*, which has 32. Figure 2 shows the network in perspective looking from the lower right corner of Figure 1. Interestingly, we see that messages cross the switches in the forward network with little challenge: they collide 5-15% of the times in *S1* and rarely collide in *S2*. In the reverse network, however, collisions are frequent: in stage *S3*, up to 20-40% of the messages collide.

In addition, messages in the reverse network collide in groups. For instance, it can be shown that for *Ocean*, 0.3% of the snapshots in *S1* have more than 2 participating inputs. In *S4*, however, it can be shown that 3.8% and 3.1% of the snapshots have 3 and 4 participating inputs respectively. This means that messages from different links arrive at switches in the reverse network more bunched-up.

This effect is fundamental to vector traffic. In a vector access, a processor issues a set of time-clustered memory requests. These requests diverge in the forward network until they reach the many memory modules. At that point, they need to converge to a single processor port. However, they are no longer as evenly spaced as they were at issue time. Therefore, they collide with each other in the reverse network switches. To see more evidence of this effect, we now examine the queues.

Queuing in the Reverse Network Is Quantitatively and Qualitatively Different

We have classified the cycles wasted to contention in network switches. As shown in Figure 3, contention cycles are due either to conflict or queuing. In the former, the message has already been selected for service and therefore it is in the current snapshot for its destination output port; in the latter, it has not. For a given message, conflict cycles are classified depending on what is keeping the output port busy: a message that belongs to the same processor (*Confl_Same*), one that belongs to a different processor (*Confl_Other*), or simply the fact that the output port queue is full (*Confl_Block*). Queuing cycles are classified

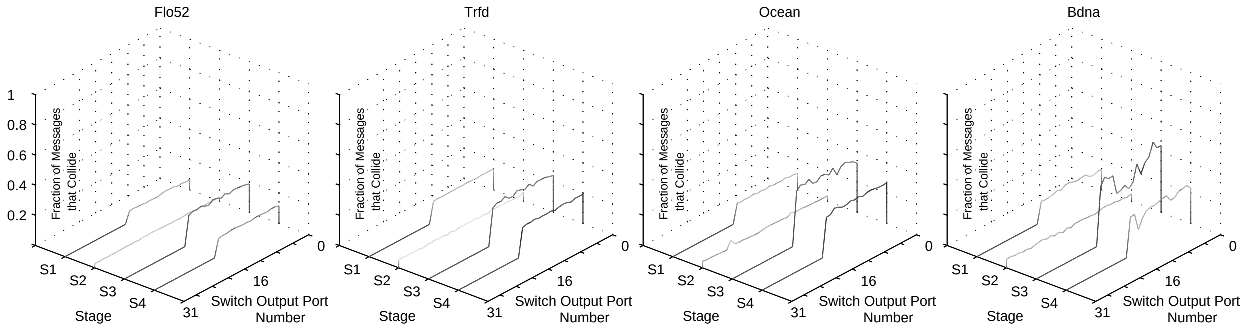

Figure 2: Frequency of message collisions. For each switch output port, the charts show the fraction of messages that collide while accessing the port.

according to the position occupied by the message in the queue: first, second, or third and beyond. Messages in the first position are queued either because they arrived late for the snapshot (*Que1_NoBlock*) or because the output port they wait for has its queue full (*Que1_Block*). Cycles wasted by messages in the second position are classified depending on whether the message goes to the same output port as the message at the head of the queue (*Que2_Need* because we need to queue) or not (*Que2_NoNeed* because, ideally, we do not need to queue). The cycles wasted by messages queued beyond the second position are called *Que3*.

Using this classification, Figure 4 classifies the cycles that the average message wastes to contention in each queue stage. Of course, for stages *Q3*, *Q4*, and *Q7*, contention cycles can only occur because of queuing, not conflict; for this reason we label them *Regular Queue* and do not divide them further. Not surprisingly, we see that messages suffer most of the contention delay in the reverse network. The stages responsible for the largest delays are *Q5* (1.4-3.7 cycles) and, to a lesser degree, *Q6* and *Q7*. By contrast, few cycles are wasted to contention in the forward network. Likewise, contention in *Q3* to access the slow memory modules is low: it costs 0.75 cycles at most.

Interestingly, most of the contention cycles in the reverse network are caused by interaction among messages from the same processor. Indeed, this interaction causes all the cycles in *Q7*; all the cycles in *Q6* except *Que2_NoNeed* and part of *Que3*; and all the cycles in *Q5* except *Confl_Other*, *Que2_NoNeed*, and part of *Que1_NoBlock* and *Que3*. Note, in particular, that *Confl_Block* and *Que1_Block* in *Q5* are caused by messages issued by the same processor as they collide in switch *S4*. Overall, it can be shown that practically all this interaction happens within, not across, vector accesses. Vector element convergence, therefore, causes the large majority of the contention cycles.

Discussion

The contention caused by the elements of a vector converging to one port is the bottleneck in Cedar's network. We believe that this problem is common in networks for medium-scale vector multiprocessors that have adequate memory bandwidth. The problem is general: it results from time-clustered messages returning from many memories to one processor port. The problem is worse if, as in Cedar's network, consecutive addresses reside in memories connected to common switches in the beginning of the reverse network. If so, the frequent stride-one vectors have to converge faster.

4.1.1 Traffic Convergence Causes Tree Saturation

The main effect of the traffic convergence is tree saturation in the reverse network. Recall that up to four 2-word messages can converge at a switch in *S4* at the same time and contend for the same output link. The output link, however, can only take one word. In fact, as indicated in Section 2.1, it may be blocked and not take any if the reverse network interface hardware is not available (this occurs 0.4-1.0% of the cycles in the applications). In any case, the bursty arrival of messages fills up the queues, which in turn block the links further back. This effect propagates back forming a saturation tree.

Figure 5 shows a 1000-cycle section of *Ocean* where tree saturation occurs. Charts (a)-(d) show, as a function of time, the cycles when links block the incoming message because they have no queue space. Each chart has the addition of these blocked cycles for all the input or output links of a switch (see Chart (e)). In addition, the charts indicate in letters the number of blocked cycles. Interestingly, as we move from (d) to (c), and from (b) to (a), the number of blocked cycles grows. Indeed, the output links of the bottom switch of stage *S4* are blocked only 76 cycles. The input links of the same switch, however, are blocked as much as 444 cycles. The same occurs for the bottom switch of stage *S3*: while the output links are blocked 303 cycles, its input links are blocked 801 cycles. Finally, we note that as we move from switch outputs to inputs, the shape of the histograms is somewhat similar. One difference is that in the inputs, the peaks are taller and occur slightly later in time. Such patterns are clearly caused by output-initiated saturation.

Tree saturation causes a significant fraction of the cycles lost in the reverse network. Indeed, it causes the blocking cycles (*Confl_Block* and *Que1_Block*) and a fraction of the remaining queuing cycles approximately equal to *Confl_Block*/(all conflict cycles). Overall, the saturation effect that we measured is similar to the one caused by a hot spot [12]. However, the solution often proposed for hot spots, namely message combining, does not work for this kind of saturation.

4.1.2 Vector Reads Suffer the Most from the Traffic Convergence

A second consequence of the traffic convergence problem is that vector reads are slowed down relative to other messages. To see why, note that, since most collisions occur among the messages of a vector access, reads tend to collide

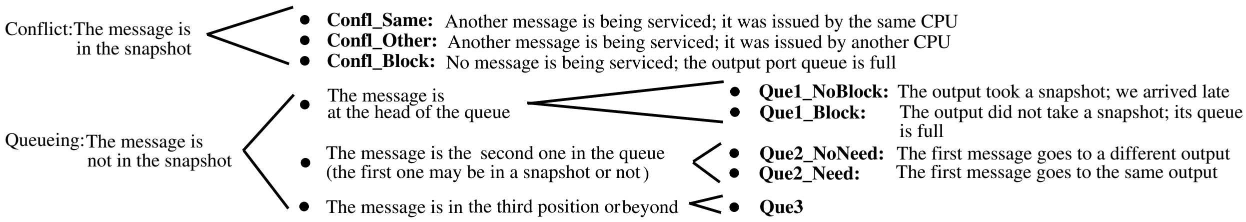

Figure 3: Classification of the cycles wasted to contention in network switches.

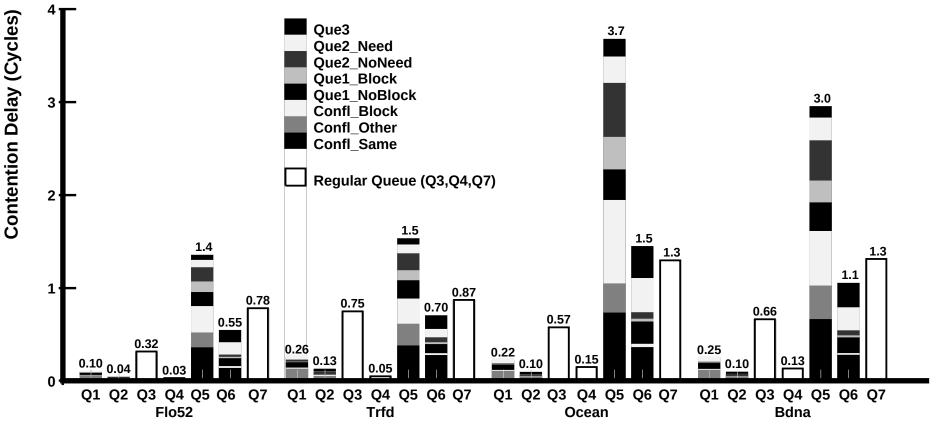

Figure 4: Classification of the cycles that the average message wastes to contention in each stage.

with reads and writes with writes. This makes a difference for two reasons. First, read messages are two words long in the reverse network while writes are only one. Therefore, reads keep switches busy longer, slowing down other reads. Second, in the reverse network, messages in a vector read travel more clustered than in a vector write: if there is no contention, the messages in a vector read return with no idle cycles in between them; in a vector write, they return with one free cycle between every two words. This is because writes do not carry data. As a result, read messages collide more easily. While neither of these two properties is necessarily present in all networks, they are not uncommon in existing designs. Our experiments show that contention does delay a message in a vector read 1.6 cycles more or 70% more than one in a vector write [16]. This is unfortunate, given that low read delays are more critical to performance than low write delays.

4.2 Collision Resolution Should Be Completely Fair

The second non-uniformity that limits the performance of the Cedar network is the uneven distribution of contention delays observed in a given queue stage. The distribution of contention delays for all the queues in the network is shown in Figure 6. In the figure, the queues for a given stage Qi are numbered from top of Figure 1 to bottom. From the figure we see that, in a given queue stage, the contention delay varies dramatically depending on the queue number. For example, in stage *Q5*, this delay ranges from 0.5 cycles for queue 0 to 2-9 cycles for queue 7.

This unevenness is caused by the scheme used in the Cedar network to resolve message collisions. As indicated in Section 2.1, when an output port is requested by several inputs at the same time, a snapshot is taken. The switch records the requesting inputs and then services them before any other snapshot is taken. Such service is done according to a fixed priority scheme. Clearly, this method of handling collisions is not too unfair: it is guaranteed that no input can starve forever. However, under the bursty Cedar traffic, this scheme causes large differences in delays. In the figure, we see that the amount of contention delay follows exactly the priority assignment: queues in high priority inputs have low contention delays and vice-versa. This correlation is particularly obvious in *Q5*.

The reason why this correlation breaks down for *Bdna* is tree saturation. In *Bdna*, the majority of the vector accesses touch memory module 14 first and then 15, 16, etc. Since the first few messages of the vector avoid the bulk of the tree saturation, messages in queue 14 and a few other queues beyond 14 suffer less contention. This is seen in *Q5* of *Bdna*. This same effect explains the dip in Figure 2 for *Bdna*.

In Cedar, the outputs of a given switch are connected to inputs with different priorities in the next stage. (This is why the links in Figure 1 do not look like the familiar shuffle). This partially randomizes the priorities of processor-processor paths and therefore makes the network somewhat fair. There is, however, no way to connect the links in this network to ensure complete fairness. Therefore, two effects occur. First, a given processor accesses some memories faster than others (up to 15% difference for *Ocean*). Second, different processors access the same memory with different speed, which causes even larger differences in memory access times (up to 17% difference for *Ocean*).

These two effects degrade performance. The first one slows down vector reads. Indeed, because of the different latency to access different memories, the elements of a vector arrive out of order. Since they need to be processed

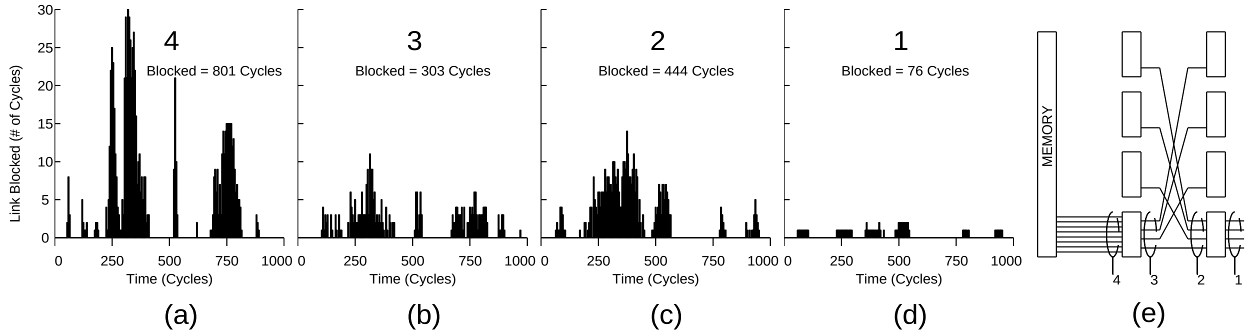

Figure 5: Link blocking as a function of time for a segment of *Ocean*. Charts (a)-(d) present the blocked cycles of all the input or output links of the switches shown in Chart (e).

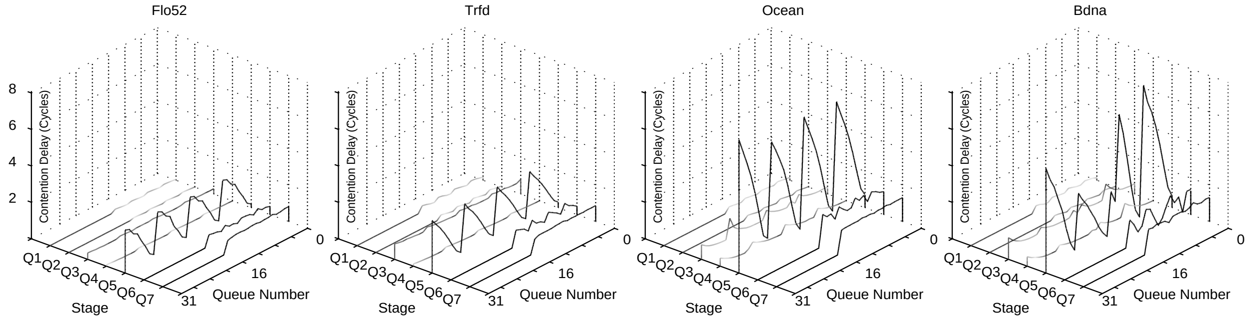

Figure 6: Number of cycles wasted by the average message to contention in each queue.

in order, the processor stalls. For *Ocean*, this stall can be 4 cycles per vector. The second effect slows down barrier synchronization. For example, in *Ocean*, if all processors read a memory element before synchronizing in a barrier, the fastest processor finishes 4 cycles before the slowest one. All these numbers are average values; peak values can be much larger.

In the past, it has been known that completely unfair fixed-priority schemes degrade performance. For example, Noakes *et al.* pointed out the back pressure on the low-priority links of the J-machine [10]. In this section, we have shown that even with a relatively fair scheme to resolve collisions like the one in Cedar, performance suffers.

4.2.1 Ripple Saturation and Memory Hot Spots

A consequence of the uneven contention delays is that in a given network stage, some links saturate more frequently than others. This saturation then propagates to previous stages, inducing similar uneven delays. This can be seen in Figure 6 for *Ocean* and *Bdna*, where the delays in *Q3* and *Q4* have the same ripple shape as in *Q5* even though neither *Q3* nor *Q4* are connected to an unfair switch. Furthermore, it can be shown that all links transfer approximately the same number of messages. This selective slow down of messages several stages before an unfair switch we call ripple saturation because it looks like a propagating wave.

Before finishing the discussion on the network's performance problems, we consider memory hot spots [12]. In our system, we did not find hot spots for three reasons. First, scheduling overheads and random delays prevent processors in a parallel Do loop from accessing memory in lockstep. Second, while vector accesses with the wrong stride can produce hot spots, the programmers of the Perfect Club codes have avoided bad strides. Finally, scalar accesses are not the best candidates either: for reads, each processor has at most 2 accesses pending at a time; for writes, processors rarely write many times to the same memory module without reading. Memory hot spots, therefore, may be relatively rare for machines with few processors like ours. We present some evidence in [16].

5 Optimizing the Performance of the Network

While in the previous section we have exposed the performance problems of the Cedar network, in this section we optimize the network cost-effectively. The goal is to reduce the processor stall time induced by messages traversing the network and memory. However, for vector architectures, the problem is not as intuitive as it appears. While the stall due to scalar reads is directly proportional to their contention delays in the network, the stall due to vector reads is not. Since the messages in a vector read are consumed by the CPU in order, the stall time is determined by the access time of the slowest element of the vector only. Therefore, we address the problem in two steps. First, we try to reduce the contention in the network. This minimizes the stall time for scalar and, partially, for vector accesses. Second, we improve vector element ordering.

To guide our analysis of the first part, we use our classification of contention cycles. Figure 7 shows the time processors stall waiting for messages traversing the network and memory. The leftmost bar in each application, labeled *Baseline* and normalized to 1, corresponds to the

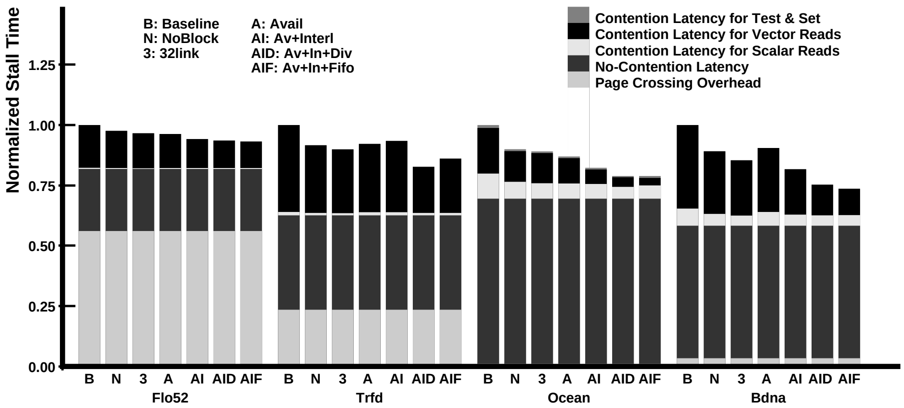

Figure 7: Decomposition of the time processors stall while messages are traversing the network and memory for different network designs. For a given application, the bars are normalized to *Baseline*.

Cedar network. The figure shows the components of the stall time. The first one, "page crossing overhead", is unavoidable overhead caused by the fact that the Cedar vector hardware uses physical addresses. This stall occurs when a vector read crosses a page boundary. The pipelining of reads stops at the page boundary until the processor consumes the data and issues the first request for the next page. At that point, the virtual page is translated and the pipeline restarted. This overhead dominates in applications with long vector strides like *Flo52* and can only be eliminated with better hardware.

The second category, "no-contention latency", includes the no-contention latency of scalar reads, test&sets, and the first element of vector reads. Since vector elements return pipelined, the no-contention latency of the remaining elements of vector reads does not stall the processor. All writes are also overlapped with computation. The remaining three categories show the effect of contention. One of them, "contention latency for vector reads" takes into account the timing and order of arrival of the vector elements. Overall, we focus on optimizations that target only the three categories caused by contention.

Since we showed that the forward network performs well, we focus on the reverse network only. Furthermore, we concentrate on the main problem, namely traffic convergence and the resulting tree saturation. Our analysis suggests (i) increasing the bandwidth of the network at the root of the traffic convergence tree and (ii) delaying the traffic convergence up until the final stages of the network. In the following, we present optimizations that accomplish this, together with other, intuitive yet less cost-effective optimizations.

5.1 Intuitive Optimizations May Not Be Cost-Effective

Two intuitive optimizations to improve network performance are to use advanced switches and to increase the bandwidth of the network. For the first optimization, we use a switch proposed in [14]. Each switch input is fanned out to as many input buffers as there are outputs in the switch (Figure 8-(a)). Therefore, switches in *S3* and *S4* have 8 and 4 2-entry buffers per input respectively. With these buffers, called non-blocking buffers, messages in one input do not block each other if they go to different outputs. For the second optimization, we double the number of links in *L5* to 32 (Figure 8-(b)). With the change, each switch in *S4* is logically partitioned into two since it is addressed with one bit only.

In Figure 7, the bars labeled *NoBlock* and *32Link* show the impact of the non-blocking buffers and the bandwidth increase respectively. Both optimizations have a noticeable impact. While the decrease in stall time looks moderate compared to the overall stall time, it is respectable compared to the categories with contention cycles, the only ones that we can reduce. Indeed, these categories decrease by 13-35%. Both scalars and vectors are sped up.

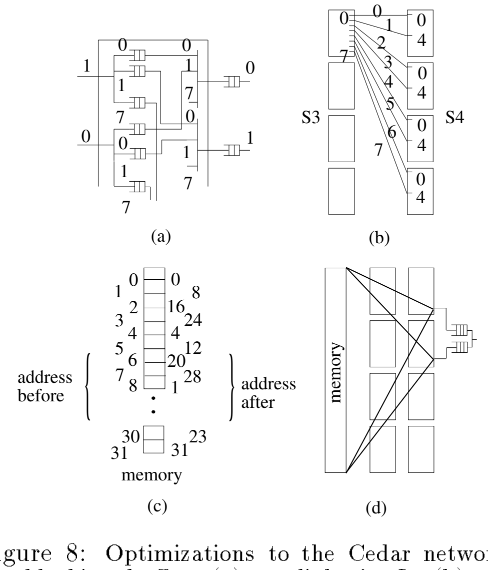

Figure 8: Optimizations to the Cedar network: non-blocking buffers (a), 32 links in *L5* (b), assignment of addresses to memories for better traffic convergence (c), and dual output ports (d).

The hardware costs, however, are high. Compared to plain buffers, 8-input non-blocking buffers in Cedar need about 4 times as much area and chips. With *NoBlock*, therefore, the reverse network costs 2-4 times more. With *32Link*, we double the number of pins, cables, and chips for one stage, therefore doubling its cost. The cost of the short reverse network of Cedar increases by 33%; the cost of a longer network where each stage is modified, however, doubles.

The reason why these optimizations are hardly cost-effective is that they do not attack the problem of traffic convergence and its major effect, the tree saturation, directly. To see the effect of *NoBlock* and *32Link* consider Figure 9. The figure classifies the contention cycles in the reverse network queues. The forward network is ignored because of its small impact.

NoBlock prevents messages from wasting cycles queuing behind other messages going to different switch outputs. Therefore, as shown in Figure 9, *NoBlock* removes the *Que2_NoNeed* cycles from *Q5* and *Q6* (*Ocean* is the easiest to follow). Other types of cycles decrease as well. However, since *Que2_NoNeed* cycles are not too common, we reduce queuing in the reverse network by only 11% on average.

With *32Link*, we increase the bandwidth past switch *S3*, but not for messages that go to the same processor. The latter have to share the same link with or without the optimization. Therefore, as seen in Figure 9, *32Link* decreases some conflict cycles (*Confl_Block* and *Confl_Other*) and queuing cycles (*Que1_NoBlock* and *Que1_Block*) for *Q5*. However, it does not reduce much the most frequent conflicts, namely those with messages in the same vector access (*Confl_Same*). For *Q6*, *32Link* does not help much either, since the main effect of the more links is to remove *Que2_NoNeed* cycles. Overall, the average reduction is only 15%.

5.2 Increasing the Network Bandwidth at the Root of the Traffic Convergence Tree is Effective

An important optimization is to increase the network bandwidth at the root of the traffic convergence tree. In our network, the root is the reverse network-to-processor interface. Therefore, we want this interface to have high bandwidth. As indicated in Section 2.1, each Cedar processor has a network interface that is unavailable to returning messages for very short intervals. When this happens, the message queues up in *Q7*. For 0.4, 0.5, 1.0, and 0.9% of the cycles in *Flo52*, *Trfd*, *Ocean*, and *Bdna* respectively, a message arriving at link *L6* finds it unavailable due to a full *Q7* queue. To avoid this, we redesign the interface to be able to take one returning word per cycle.

In Figure 7, the *Avail* bars (for available) show the new processor stall time. This change reduces the contention-induced stall by 20-43%. Compared to the more expensive *NoBlock* and *32Link*, it performs better for *Flo52* and *Ocean* and worse for *Trfd* and *Bdna*. Not surprisingly, the reduction is high in the applications where the congestion caused by traffic convergence was high (*Ocean*). Furthermore, traffic burstiness plays a major role: the large reduction in *Ocean* is achieved by optimizing links that were blocked a very small fraction of the cycles on average.

While this optimization does not reduce traffic convergence, it is effective because it tackles the effect of traffic convergence: link saturation. It lessens the saturation of the links. In Figure 9, we see that *Avail* reduces the *Confl_Block* and *Que1_Block* cycles in *Q6* and especially in *Q5*. These cycles are caused by link saturation. In addition, it also reduces other types of cycles and, by definition, eliminates *Q7* queuing. Overall, reverse network queuing decreases by 40-60%. The advantage of *Avail* is that it is cheaper than *NoBlock* or *32Link*: the contention between the messages accessing the network and the ones returning can be eliminated with relatively simple changes in the hardware of the interface module.

While the partial unavailability of the network-processor interface may be a design defect of Cedar, the problem of traffic convergence and tree saturation is general. Our experiment proves that a small enhancement at the root of the convergence tree brings a very significant improvement.

5.3 Delaying Traffic Convergence Produces Gains with Low Cost

To further reduce delays, we consider two optimizations that delay traffic convergence up until the final stages of the network. The goal is to have the messages of a stride-one vector access converge along paths separated as much as possible in space and time. The first optimization is to change the relative position of the memory modules. For a word-interleaved memory, the sooner the links out of two memories converge, the larger the difference between addresses in the two memories should be. For Cedar, we want to prevent 8 memories with 8 consecutive addresses from sharing the same switch in *S3*. Therefore, we change the routing logic in the forward network so that the numbering of the 64 memories changes from $m_4 m_3 m_2 m_1 m_0$ to $m_1 m_0 m_2 m_4 m_3$ (Figure 8-(c)). Hence, any 4-element stride-one vector access accesses memories sharing no switch in *S3*. This address assignment is similar to Cray Y/MP's [14] and has negligible cost.

The processor stall time with this new interleaving and the optimized network interface discussed above is shown in the *Av+Interl* bars of Figure 7 (*Av+Interl* stands for available with new interleaving). We see that the new interleaving makes a difference. Overall, the contention-induced stall is reduced by 44% and 59% of the *Baseline*'s in *Bdna* and *Ocean* respectively. As expected, the contention latency of vectors decreases the most. Interestingly, however, *Trfd* performs worse. The reason is that *Trfd* has many stride-40 vector accesses. With the new address assignment, a stride-40 vector accesses consecutive memory modules, suffering the problem avoided by stride-1 vectors. Overall, however, since stride-1 vectors are dominant in most applications, this optimization is good.

Av+Interl is effective because it pushes traffic convergence as late as possible in the reverse network. Therefore, it restricts its effect to a much smaller part of the system. Indeed, Figure 9 shows that, except in *Trfd*, *Av+Interl* removes all *Confl_Same* cycles in *Q5* and greatly reduces other types of cycles. Of course, *Q6* swells because collisions now occur at the end of the reverse network. However, total queuing decreases. Overall, contention delays in the reverse network decrease by 45% of the *Baseline* for *Trfd* and 63-76% for the other applications. In summary, this experiment shows that traffic convergence should be delayed up until the final stages of the network. In this light, the shuffle-exchange topology is not the best one because links from consecutive memories tend to converge early.

The second optimization to delay traffic convergence is to double the number of ports per processor in the reverse network. If a given port is in one *S4* switch, its dual will be placed in a different, albeit close-by switch. Then, messages sent by even-numbered memories to a given processor are sent to the regular port; messages sent by odd-numbered memories to the same processor are sent to its dual port. In this way, the convergence tree caused by a stride-one vector access is decoupled into two trees with

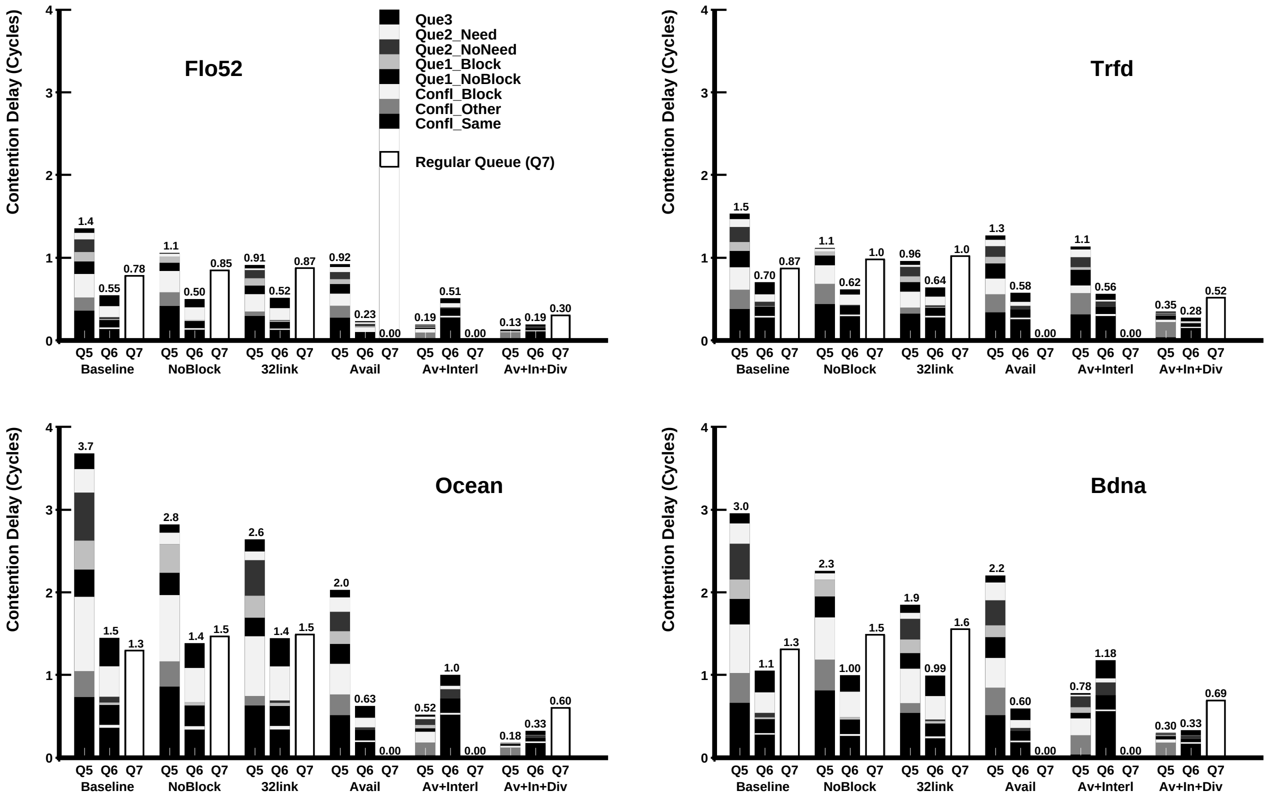

Figure 9: Decomposition of the cycles that the average message wastes to contention in the reverse network. Each set of three bars corresponds to a different network design.

different roots (Figure 8-(d)). Therefore, the tree saturation effect is reduced. With this change, the last network stage costs twice as much; the rest of the network, however, costs the same.

If we add this optimization to *Av+Interl*, we get the *Av+In+Div* bars in Figure 7 (*Div* stands for diverging). As expected, the stall time in *Trfd* drops. Moreover, the other applications also improve. As expected, most of the reductions occur in the contention for vectors category. Overall, the stall time due to contention is now 31-65% of the value in *Baseline*. In addition, as seen in Figure 9, although there is some queuing in *Q7*, messages suffer almost no contention in the reverse network. We are left with only 10-33% of the contention cycles of *Baseline*. To summarize, the two techniques presented in this subsection are effective because they tackle the main problem, namely tree saturation resulting from traffic convergence. The common approach has been to delay the convergence up until the end stages of the network. While *Div* is somewhat expensive, *Interl* is very cheap.

5.4 Improving Even a Relatively Fair Collision Resolution Scheme Helps

As pointed out in Section 4.2, under the traffic considered, even a relatively fair collision resolution scheme causes significant vector element disorder. This worsens traffic convergence. Therefore, we augment the most cost-effective network, namely *Av+Interl* by adding a global timestamp to all messages and forcing the switches to give priority to the messages with lower timestamp in case of collision. Since we use the unused header bits of a Cedar message for the timestamp, messages do not increase in size. Hence, the main cost is simply to change the switches to compare the timestamps.

The resulting stall time is shown in Figure 7 under *Av+In+Fifo*. We see that this simple change significantly reduces the stall caused by contention in vector reads. Overall, the stall time due to contention is now 30-63% of the value in *Baseline*. We see that adding fair switches performs as well or better (except for *Trfd*) than the more expensive addition of double ports per processor (*Div* optimization). Therefore, the latter is not as cost-effective. Similarly, a combined scheme (*Av+In+Div+Fifo*) can be shown not to be as cost-effective either. The conclusion, therefore, is that under this bursty traffic, it is worth having a completely fair collision resolution scheme.

6 Related Work

Many authors have used analytical models to predict the performance of multistage switching networks [4, 6, 11]. Turner and Veidenbaum [18] used simulations to study vector machines of up to 512 CPUs, focusing on average-value parameters like speedup, efficiency, or latency. Turner [17] simulated architectural variations like different memory speeds or amount of buffering. Partly because of the relatively low memory bandwidth of his system, the best optimization was large memory input buffers. Smith and Taylor [14] pointed out that non-blocking buffers substantially increase performance. Our results agree. Smith and Taylor [15] further discussed how vector disorder degrades performance. In this paper, we showed that, under bursty traffic, relatively fair switches create such disorder. Gottlieb *et al.* [4] described a network that supports combining. Pfister and Norton [12] analyzed memory hot spots and tree saturation in the forward network. Noakes *et al.* [10] presented measurements of average-value parameters in a real machine. We use a hardware monitor to go beyond averages and record irregularities. Based on analytical models, Granston *et al.* [5] and McAuliffe [9] point out that the fan-in of read bursts causes performance degradation in the reverse network. However, no detailed characterization or solutions are provided.

7 Conclusion

A realistic analysis of a multistage switching network suggests that the performance of these networks is limited by traffic non-uniformities. We identify two major non-uniformities that degrade Cedar's performance and are likely to slow down other networks too. The first one is the contention caused by the return messages in a vector access as they converge from the memories to one processor port. This traffic convergence penalizes vector reads and, most importantly, causes tree saturation. We believe that this effect is common in medium-scale vector machines with adequate memory bandwidth. The second non-uniformity is the uneven contention delays induced by even a relatively fair scheme to resolve message collisions.

Based on these observations and further analysis, we argue that intuitive optimizations for multistage switching networks may not be cost-effective. Instead, network optimizations should focus on cheap changes to (i) increase the bandwidth of the network at the root of the traffic convergence tree and (ii) delay the traffic convergence up until the final stages in the network. Finally, we also show that, under the bursty traffic present in vector machines, collision resolution schemes must be completely fair.

Acknowledgments

We thank Pen-Chung Yew, Alex Veidenbaum, Thomas Murphy, John Andrews, Ding-Kai Chen, Dan Lavery, Russ Daigle, Steve Turner, and the referees for their feedback. We also thank Alex Veidenbaum, Carl Beckmann, Sunil Kim, and Elana Granston for letting us use their simulator.

References

[1] J. B. Andrews. A Hardware Tracing Facility for a Multiprocessing Supercomputer. Technical Report 1009, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, May 1990.

[2] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[3] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Center for Supercomputing Research and Development, October 1992.

[4] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. In *IEEE Trans. on Computers*, pages 175–189, February 1983.

[5] E. D. Granston, S. W. Turner, and A. V. Veidenbaum. Design and Analysis of a Scalable Shared-memory System with Support for Burst Traffic. 16th ISCA Workshop on Cache and Interconnect Architectures in Multiprocessors, Kluwer Academic Publishers, Boston, MA, 1989.

[6] C. Kruskal and M. Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. In *IEEE Trans. on Computers*, pages 1091–98, December 1983.

[7] D. Kuck et al. The Cedar System and an Initial Performance Study. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 213–224, May 1993.

[8] D. H. Lawrie. Access and Alignment of Data in an Array Processor. In *IEEE Transactions on Computers*, pages 1145–1155, December 1975.

[9] K. McAuliffe. Analysis of Cache Memories in Highly Parallel Systems. Technical Report 269, Department of Computer Science, New York University, May 1986.

[10] M. Noakes, D. Wallach, and W. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[11] J. H. Patel. Performance of Processor-Memory Interconnections for Multiprocessors. *IEEE Transactions on Computers*, 30(10):771–780, October 1981.

[12] G. Pfister and A. Norton. 'Hot Spot' Contention and Combining in Multistage Interconnection Networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 790–797, August 1985.

[13] Panel Session. Benchmarking Interconnects. In *Hot Interconnects I Symposium*, August 1993.

[14] J. E. Smith and W. R. Taylor. Accurate Modeling of Interconnection Networks in Vector Supercomputers. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 264–273, June 1991.

[15] J. E. Smith and W. R. Taylor. Characterizing Memory Performance in Vector Multiprocessors. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 35–44, July 1992.

[16] J. Torrellas and Z. Zhang. The Performance of the Cedar Multistage Switching Network. Technical report, Center for Supercomputing Research and Development, 1994.

[17] S. Turner. Shared Memory and Interconnection Network Performance for Vector Multiprocessors. M.S. thesis. Technical Report 876, Center for Supercomputing Research and Development, May 1989.

[18] S. Turner and A. Veidenbaum. Performance of a Shared-Memory System for Vector Multiprocessors. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 315–325, July 1988.

[19] C. Wu and T. Feng. *Tutorial: Interconnection Networks for Parallel and Distributed Processing.* The Computer Society of the IEEE, 1984.