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
torrella,zzhang@csrd.uiuc.edu

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 a relatively fair scheme to resolve message collisions. Based on our observations, we argue that intuitive optimizations for multistage switching networks may not be the most cost-effective ones. 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.

Keywords: Multistage Switching Networks, Vector Multiprocessors, Performance Evaluation, Experimental Analysis, Address Tracing.

1 Introduction

Few topics in computer systems have attracted more attention than multistage switching networks for multiprocessors [23]. 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, 7, 11, 13, 15, 22, 24] or on simulations [5, 18, 19, 20, 21], 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

[1]A shorter version of this paper was presented at Supercomputing 1994. This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436, RIA MIP 93-08098, MIP 93-07910, and MIP 89-20891; NASA Contract No. NAG-1-613; and a grant from the University of Illinois Research Board.

average-value parameters like round trip time, bandwidth, or utilization [14]. Some other studies have been performed with the network disconnected from the machine. Again, the information provided is limited. As a result, researchers still lack a complete picture of the performance of multistage switching networks in real systems and how to compare their performance [17].

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 [9] of the Cedar shared-memory machine [8] executing real applications. The machine is configured with 16 CPUs. The analysis of queues, switch elements, and links 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. 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 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 (CSRD), University of Illinois [8]. 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. Consecutive 64-bit words are allocated in consecutive memory modules. 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 [9] (Figure 1). The cycle time of the networks is 85 ns. Running Cedar with 16 processors makes the system more balanced. Indeed, the total bandwidth of the processors and memories is now equal: each processor can issue one (64-bit word) read message to the network every two cycles, while memory modules have a 4-cycle cycle time. The reason why a processor can issue only one network access every two cycles is that processors cycle at half the speed of the network.

Each network in Figure 1 is synchronous, packet-switched and has 64-bit wide data paths. A read consists of one 64-bit 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

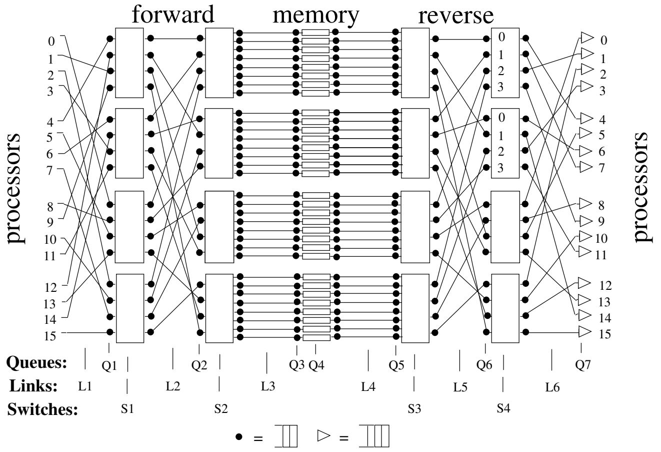

Figure 1: The Cedar network.

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 incoming messages every cycle. This is because every returning read needs to use an internal bus in 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.

The network topology is a shuffle-exchange. However, 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 all the inputs requesting that output port by creating what we call a "snapshot". Then, it services the inputs in the snapshot according to a fixed priority scheme. Only when this is done, it takes the next snapshot for the output port. 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 fairer.

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 corresponds to a data access and 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 timings of the Cedar network are simple and deterministic, there is no need to perform any validation of the simulator: the simulator models exactly the behavior of the network.

Since the processors fill up the trace buffers in a few hundred ms, we built a feedback connection between the performance monitor and Cedar (Figure 2). 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 contains the "halt" instruction. This instruction freezes the processors. Then, the workstation controlling the performance monitor reads the traces from the buffers and either postprocesses them on the fly or dumps them in memory. Then, it 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 few 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 about 2 μs to be fully loaded with application traces again, while it runs for a few hundred ms between interrupts.

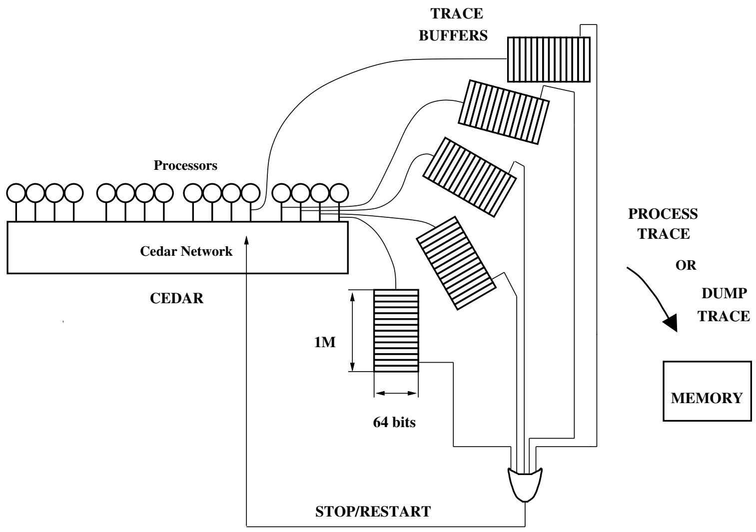

Figure 2: Setup used to generate address traces.

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 reasonably accurate.

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 compiler- and hand-parallelized versions of four Perfect 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 is 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

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.

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.

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 hierarchy and inefficiencies in the processor-network interface. The purpose of this paper, however, is to examine this 10-22% of the execution time only.

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

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 suffer in the reverse network as they converge to one processor port, and the uneven contention delays induced by the relatively fair scheme used 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 data shows that messages in the reverse network collide more frequently than in the forward network. Figure 3 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 3 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.

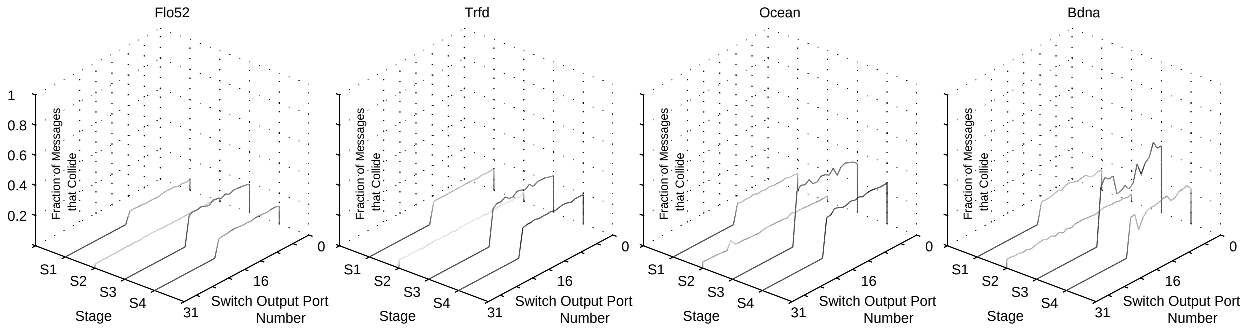

Figure 3: Frequency of message collisions. For each switch output port, the charts show the fraction of messages that collide while accessing the port.

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

As expected, most of the queuing occurs in the reverse network. Figure 4 shows the average length of the queues in the network in words. In the figure, the queues for a given stage *Qi* are numbered from top of Figure 1 to bottom. While we see marked variations among the queues in the same stage, we postpone any discussion of it until we examine collision resolution issues. Instead, we focus on average values only. Clearly, the queues in the forward network and memory are short: on average only 0.1-0.2 words. However, the queues in the reverse network are long: up to 0.65 words. The reason is the traffic convergence in the switches of the reverse network.

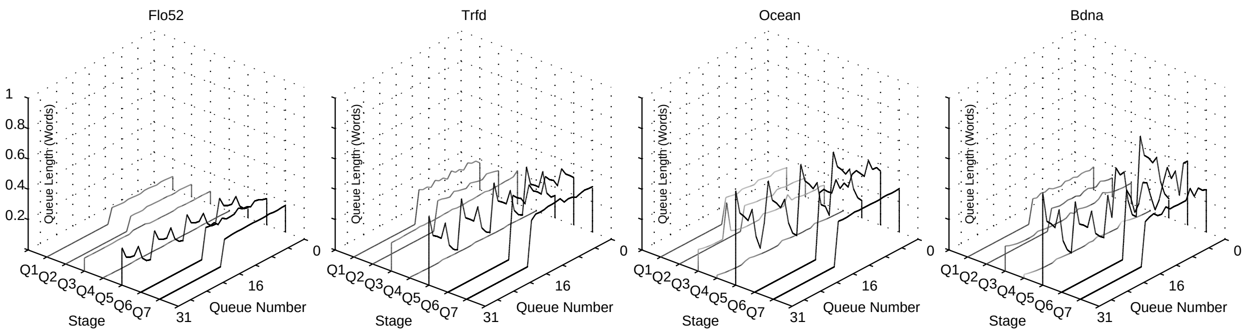

Figure 4: Average number of words queuing at each queue in stages *Q1-Q7*.

We are now ready to analyze the cycles wasted to contention in network switches. Table 3 shows the different classes of such cycles. In the *Confl_Same* and *Confl_Other* cycles, the message has already been selected for service and therefore it is in the current snapshot for its destination output port. However, the message is not being serviced because the requested switch output port is being used by another message issued by the same processor (*Confl_Same*) or by a different processor (*Confl_Other*). *Blocking* cycles are those that are lost when the message is at the head of its input port queue but the requested switch output port is unable to service any message because its queue is full. In this case, we say that corresponding output link is saturated. *Que_NoNeed* are cycles lost by a message while queuing in a switch input queue behind a message that goes to a different output port. We call these cycles "NoNeed" because, ideally, the message would not need to queue since it goes to a different output port. Finally, *Rest_Queuing* are cycles lost to the other instances of message queuing.

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

Class	Description
Confl_Same	The message is in the snapshot. However, the requested switch output port is being used by another message issued by the same processor.
Confl_Other	The message is in the snapshot. However, the requested switch output port is being used by a message issued by another processor.
Blocking	The message is at the head of its input port queue but the requested switch output port is unable to take any message: its queue is full.
Que_NoNeed	The message is queuing behind a message that goes to a different output port.
Rest_Queuing	Other instances of message queuing.

Using this classification, Figure 5 shows the cycles that, on average, a message wastes to contention in each queue stage. Of course, for stages *Q3*, *Q4*, and *Q7*, contention cycles can only occur because of plain queuing; for this reason we label them *Regular Queue* and do not divide them further. Overall, we see that, not surprisingly, messages suffer most of the contention delay in the

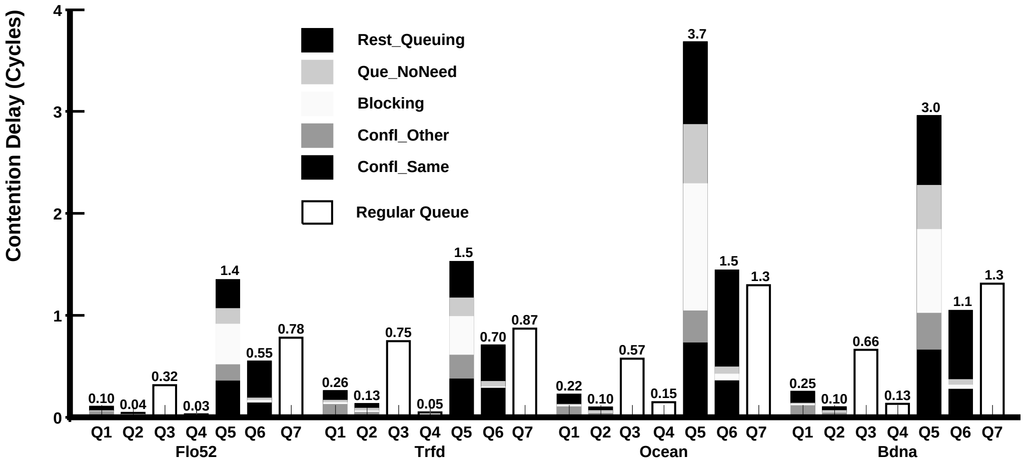

Figure 5: Classification of the cycles that, on average, a message wastes to contention in each stage.

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 collisions among messages from the same processor. Indeed, these collisions cause all the cycles in *Q7*. They also cause all the cycles in *Q6* except *Que_NoNeed*. Finally, they also cause the *Confl_Same*, *Blocking*, and part of the *Rest_Queuing* cycles in *Q5*. Note that the *Blocking* cycles in *Q5* are caused by messages issued by the same processor as they collide in switch *S4*. In our experiments, we did not measure what fraction of the *Rest_Queuing* cycles is due to collision among messages issued by the same processor and what fraction is not. However, even if none of it is, contention cycles due to collision among messages from the same processor still are a majority in *Q5*. In reality, a large fraction of the *Rest_Queuing* cycles probably belongs to this class.

In addition, it can be shown that practically all these collisions happen 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 enough 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. Tree saturation occurs in the following situation. It is possible that, due to random message collisions, several messages directed to the same processor converge in an *S4* switch at the same time. In the worst case, four 2-word messages for the same processor arrive at the four inputs of an *S4* switch at the same time. All request 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 6 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. This number results from adding up the cycles for the four links. 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. The peaks are taller in the inputs because the blocking involves more messages, including messages that go to different outputs. Such patterns are clearly caused by output-initiated saturation.

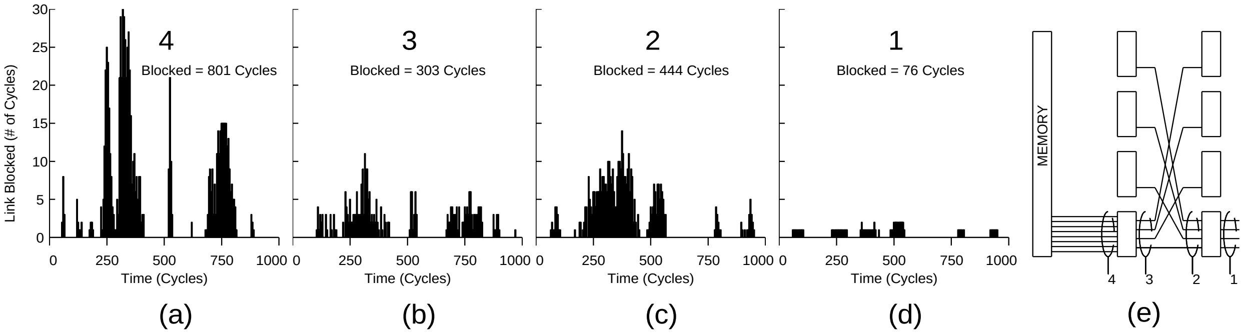

Figure 6: Link blocking as a function of time for a segment of *Ocean*. Charts (a)-(d) present, as a function of time, the addition of the blocked cycles for the input or output links of the switches shown in Chart (e).

Tree saturation causes a significant fraction of the cycles lost in the reverse network. Indeed, it causes the *Blocking* cycles. In addition, it causes some of the *Que_NoNeed* and *Rest_Queuing* cycles, namely those cycles that messages spend queuing behind others that are stalled in a blocked link. Overall, the saturation effect that we measured has some similarities with the one caused by a memory hot spot [16]. 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 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 messages. 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.

To measure the higher latency of vector reads, we can not use the Cedar network. As indicated in Section 2.1, reads contend for a bus in the reverse network interface while writes do not. This contention slows reads a bit and would confuse the results. Instead, we use a simulated network with a better interface that can always take returning reads or writes (*Avail* network in Section 5). For such a network, Table 4 shows the cycles wasted to contention by the average vector/scalar read and write, and test&set. To compute the values for vector accesses, we average out all component messages. From the table, we see that contention does delay a message in a vector read 1.6 cycles more or 70% more than one in a vector write. This is unfortunate, given that low read delays are more critical to performance than low write delays.

Table 4: Cycles wasted to contention by different message types in the *Avail* network (Section 5). A contention-free round trip takes 14 cycles.

Application	Vector Read		Vector Write		Scalar Read		Scalar Write		Test&Set		Average
	# of Reqs (%)	Cont. Delay (Cyc.)	# of Reqs (%)	Cont. Delay (Cyc.)	# of Reqs (%)	Cont. Delay (Cyc.)	# of Reqs (%)	Cont. Delay (Cyc.)	# of Reqs (%)	Cont. Delay (Cyc.)	Cont. Delay (Cyc.)
Flo52	72.8	2.00	23.6	1.13	2.4	0.51	1.2	0.50	0.0	0.44	1.74
Trfd	64.6	3.74	33.0	2.37	1.6	1.20	0.8	1.17	0.0	0.99	3.23
Ocean	58.4	4.77	33.2	2.79	5.0	2.03	2.5	2.01	0.9	1.22	3.87
Bdna	68.7	4.84	27.0	2.77	2.9	2.29	1.4	2.25	0.0	0.48	4.17
AVERAGE	66.1	3.84	29.2	2.26	3.0	1.51	1.5	1.48	0.2	0.78	3.25

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 7. 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, while this method of handling collisions is not totally fair, it is not completely unfair either: 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 3 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 (up to 17% difference for *Ocean*).

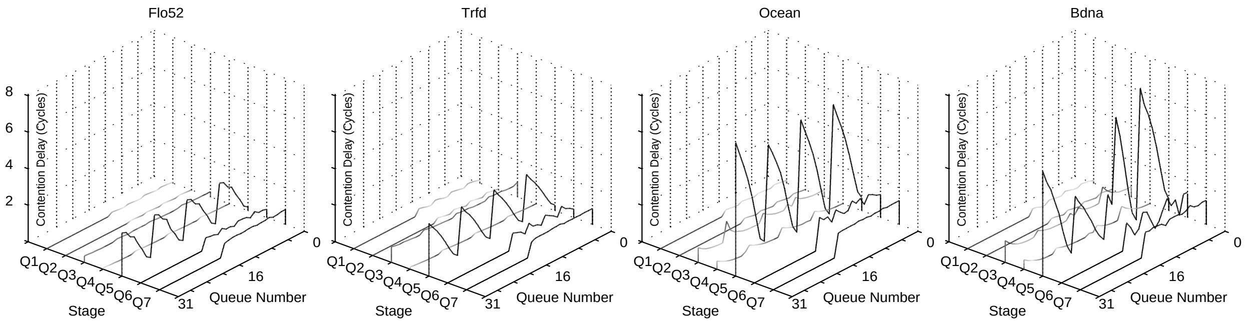

Figure 7: Average number of cycles wasted by a message to contention in each queue.

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 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 [14]. 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 7 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.

To get insight into the problem, we examine the fraction of cycles that links transfer data in *Ocean* and *Bdna* (Figure 8). The format of the figure is similar to that of Figure 7. Note that the *L3* and *L4* links have half the utilization of the others because these stages have 32 links instead of 16. In addition, because messages change size in memory, the links in the forward and reverse networks have different utilizations. Clearly, most of the links in *L3* transfer the same number of messages. Therefore, the ripple must be caused by the saturation in *L4*. Furthermore, given the low utilization of *L3* and *L4*, we conclude that, to create the ripple, the traffic must be very bursty. 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 [16]. 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 to cause hot spots 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.

This third argument is illustrated in Figure 8 for *Ocean*. The peak in *L4* shows that the output link of one of the memories is used twice as much as the others. This memory has the control variable of a barrier that is polled by the processors. While this peak suggests a hot spot, Figure 7 shows that the delay in queue 22 of *Q3* increases by less than one cycle. The reason is that, since processors have at most two pending reads, polling does not cause a large contention. Memory hot spots, therefore, may be relatively rare for machines with few processors like ours.

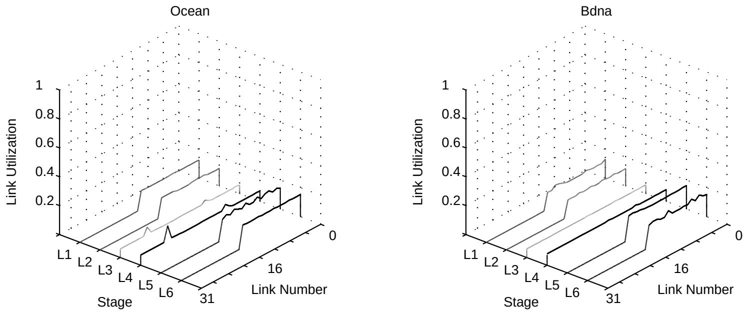

Figure 8: Fraction of the time that links transfer data in *Ocean* and *Bdna*.

5 Optimizing the Performance of the Network

While in the previous section we have discussed the performance problems of the Cedar network, in this section we attempt to optimize the network. The goal is to reduce the processor stall time induced by messages traversing the network and memory. In a machine such as the one considered, we need to handle scalar and vector reads differently. For scalar reads, we need to focus exclusively on reducing the contention delays that the messages encounter in the network. For vector reads, however, the stall time is determined by the latency of the slowest element of the vector only. This is because the messages in a vector read are consumed by the CPU in order. Therefore, we also need to improve vector element ordering.

To assess how much we can speed-up the network accesses, Figure 9 breaks down the time processors stall waiting for messages traversing the network and memory in the Cedar machine. Each bar corresponds to a different application. The different categories in the bars correspond to the components of the stall time. The category at the bottom, "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 the vector element 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 new virtual page is translated and the pipeline restarted. This overhead is large in applications with long vector strides like *Flo52* and, to a lesser extent, *Trfd*. It 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-

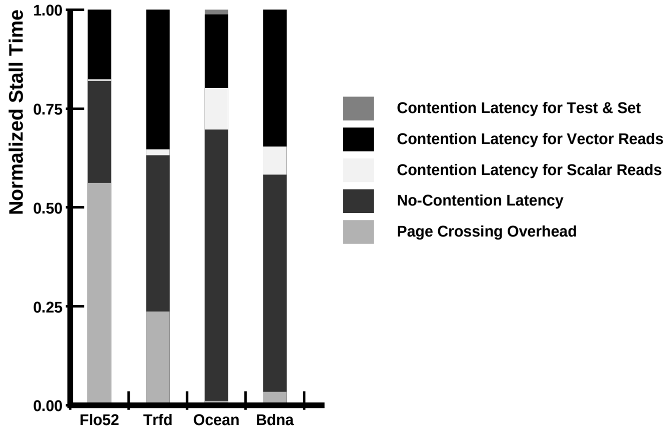

Figure 9: Breakdown of the time processors stall while messages are traversing the network and memory of Cedar.

contention latency of the remaining elements of vector reads does not stall the processor. Furthermore, all writes are overlapped with computation, which means that writes do not affect the stall time directly. The remaining three categories show the effect of contention on scalar reads, vector reads, and test&set respectively. The "contention latency for vector reads" takes into account the latency and order of arrival of the vector elements. Overall, in this paper, we will 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 two problems identified in the previous section, namely the traffic convergence in the reverse network and the unfair collision resolution scheme. Before addressing these two problems, however, we discuss optimizations that are intuitive but not as cost effective.

5.1 Intuitive Optimizations May Not Be the Most Cost-Effective Ones

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 non-blocking switches. Each switch input is fanned out to as many input buffers as there are outputs in the switch (Figure 10-(a)). Therefore, switches in *S3* and *S4* (Figure 1) have 8 and 4 2-entry buffers per input respectively. With these 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 10-(b)). With this change, each switch in *S4* is logically partitioned into two since it is addressed with one bit only.

Figure 11 shows the time that processors stall due to message contention in the network and memory. For each application, we consider different network designs. The bars are normalized to *Baseline*, which corresponds to the Cedar network. The bars are divided into the contributions of the three contention categories in Figure 9. Recall that these three categories are the only ones that we will attempt to optimize. The bars labeled *NoBlock* and *32Link* show the impact of the non-blocking switches and the bandwidth increase respectively. Both optimizations have a noticeable impact. The decrease in stall time is a respectable 12-36%. Both scalars and vectors are sped up.

The hardware costs, however, are high. Compared to plain switches, 8-input non-blocking

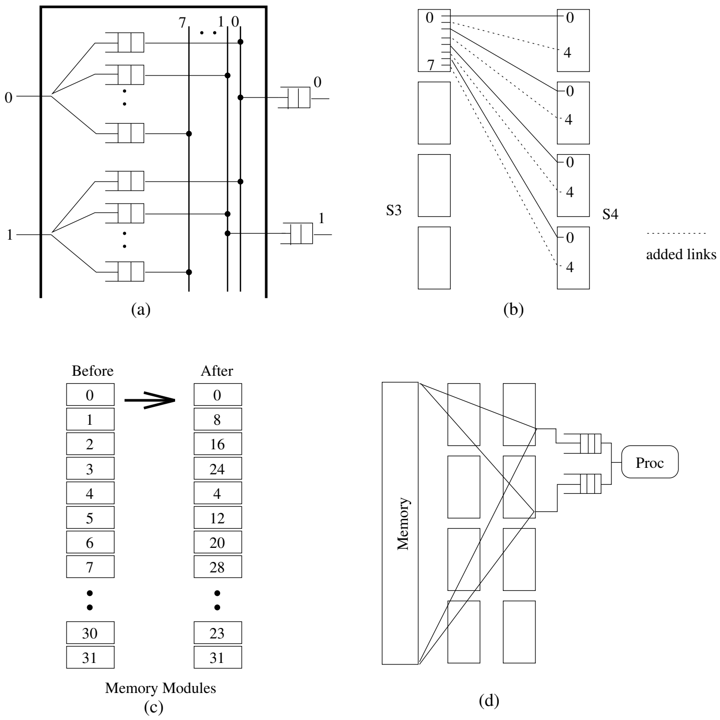

Figure 10: Optimizations to the Cedar network: non-blocking switches (a), 32 links in *L5* (b), assignment of addresses to memory modules for smoother traffic convergence (c), and dual output ports (d).

switches 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 for the modest cost-effectiveness of these optimizations is that they do not directly attack the problem of traffic convergence and its major effect, the tree saturation. To see the effect of *NoBlock* and *32Link* consider Figure 12. The figure classifies the cycles wasted to contention by the average message in the queues of the reverse network for different network configurations. The forward network is ignored because of its small impact. As it was done in Figure 5, the contention cycles in *Q5* and *Q6* are divided into *Confl_Same*, *Confl_Other*, *Blocking*, *Que_NoNeed*, and *Rest_Queuing*. *Q7* is not divided into categories because *Q7* is a regular queue.

NoBlock prevents messages from wasting cycles queuing behind other messages going to different switch outputs. Therefore, as shown in Figure 12, compared to *Baseline*, *NoBlock* removes the *Que_NoNeed* cycles from *Q5* and *Q6*. Other types of cycles decrease as well. However, since *Que_NoNeed* cycles are a minority, queuing in the reverse network decreases 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. As seen in Figure 12, *32Link* decreases, for *Q5*, some of the *Confl_Other*, *Blocking*, and *Rest_Queuing* cycles. However, it does not reduce the most frequent conflicts much, namely those among 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 *Que_NoNeed* cycles. Overall, the average reduction is only 15%. We can see, therefore, that these two optimizations are not very effective.

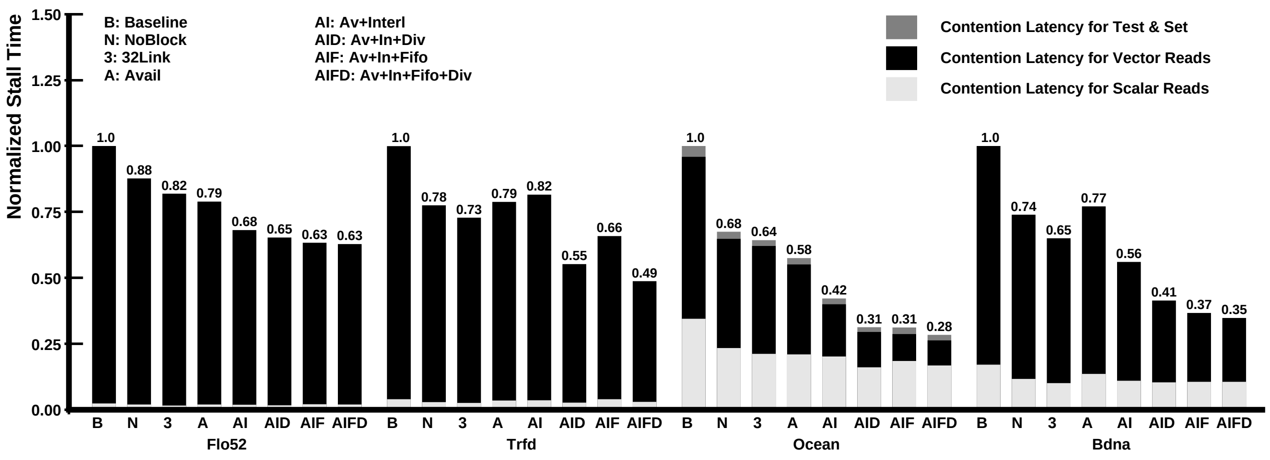

Figure 11: Decomposition of the time that processors stall due to message contention in the network and memory. For each application, we consider different network designs. The bars are normalized to *Baseline*.

5.2 Optimizing the Convergence of Traffic

To optimize the convergence of traffic, we perform two changes to the network. We consider them next.

5.2.1 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. While this optimization does not reduce traffic convergence, it addresses the main effect of traffic convergence: tree saturation. It lessens the saturation of the links.

In our network, the root of the tree is the reverse network-to-processor interface. Therefore, we want this interface to have high bandwidth. As indicated in Section 2.1, however, each Cedar processor has a network interface that is unavailable to returning messages for very short intervals. This is an implementation issue in Cedar. It occurs because every returning read needs to use an internal bus in 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. When the interface is required but it is unavailable, 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 11, the *Avail* bars (for available) show the new processor stall time. This change reduces the contention-induced stall by 21-42%. 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 in stall time is highest in the applications where the congestion caused by traffic convergence was highest (*Ocean*). Furthermore, traffic burstiness plays a major role: the large reduction in *Ocean* is achieved by optimizing links that were blocked a small fraction of the cycles on average.

In Figure 12, we see that *Avail* reduces the *Blocking* cycles in *Q5*. These cycles are caused by link saturation. In addition, it also reduces other types of cycles in *Q5* and *Q6* 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.

5.2.2 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. Roughly speaking, 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. In Cedar, 8 memories with 8 consecutive addresses are sharing the same switch in *S3*. Therefore, we change the routing logic in the forward network so that the numbering of the 32 memory modules changes according to Figure 10-(c). If the old number of a memory module was given by the 5 bits $bit_4bit_3bit_2bit_1bit_0$, the new number is given by $bit_1bit_0bit_2bit_4bit_3$. A simple swap of the bits generates the right memory module number. Now, any 4-element stride-one vector request accesses memory modules sharing no switch in *S3*. This address assignment is similar to Cray Y/MP's [18] 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 11 (*Av+Interl* stands for available with new interleaving). We see that the new interleaving makes a difference: it reduces the contention-induced stall by 14-28% in *Flo52*, *Ocean*, and *Bdna*. As expected, the contention latency of vectors decreases the most. Interestingly, however, *Trfd* performs 4% 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 the effect of traffic convergence to a much smaller part of the system. Indeed, Figure 12 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. Total queuing, however, 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 memory modules tend to converge early.

The second optimization to delay traffic convergence is significantly more expensive. It involves doubling 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 memory modules to a given processor are sent to the regular port; messages sent by odd-numbered memory modules 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 different roots (Figure 10-(d)). Therefore, the tree saturation effect is reduced. With this change, the last network stage costs about twice as much; the rest of the network, however, stays the same.

If we add this optimization to *Av+Interl*, we get the *Av+In+Div* bars in Figure 11 (*Div* stands

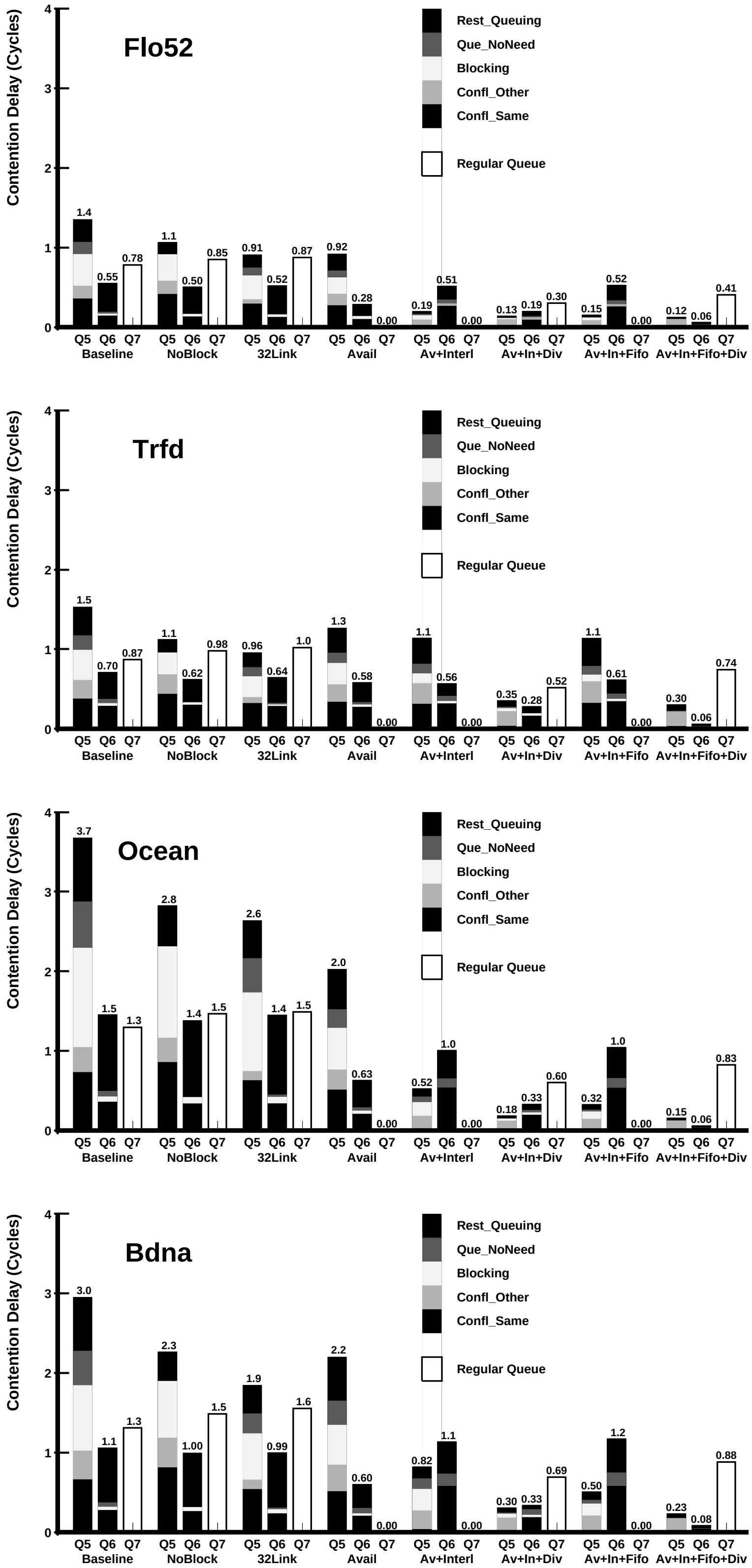

Figure 12: Decomposition of the cycles that, on average, a message wastes to contention in the reverse network. Each set of three bars corresponds to a different network design.

for diverging). In the figure, the stall time in *Trfd* drops by as much as 33%. This is because stride-40 vectors now have a better traffic convergence pattern. Moreover, the other applications also improve. As expected, most of the reductions occur in the vector category. Overall, the stall time due to contention decreases by 26-27% in *Ocean* and *Bdna* and by 4% in *Flo52*. In addition, as seen in Figure 12, although there is some queuing in *Q7*, messages suffer little contention in the reverse network. We are left with only 10-33% of the contention cycles of *Baseline*.

To summarize, we note that the two techniques presented in this subsection are effective because they tackle the main problem, namely traffic convergence. Their common approach has been to delay the convergence up until the end stages of the network. Their cost, however, is different: while *Div* is expensive, *Interl* is very cheap. To see how these techniques compare to the ones presented before, Table 5 shows the type of contention cycles that each technique targets.

Table 5: Type of contention cycles targeted by the optimization techniques described.

Type of Cycle	*NoBlock*	*32Link* *Q5*	*32Link* *Q6*	*Avail*	*Av+Interl* or *Av+In+Div*
Confl_Same					X
Confl_Other		X			
Blocking		X		X	X
Que_NoNeed	X		X		
Rest_Queuing		X			X

5.3 Improving Even a Relatively Fair Collision Resolution Scheme Helps

The second problem that we address is the sub-optimal collision resolution scheme. As pointed out in Section 4.2, under the traffic considered, even a relatively fair collision resolution scheme causes 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. Then, we force 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 to add some logic to generate the timestamps and to change the switches to compare the timestamps.

The resulting stall time is shown in Figure 11 under *Av+In+Fifo*. The figure assumes that the generation and comparison of timestamps does not change the cycle time of the system. From the figure, we see that this change reduces the stall caused by contention in vector reads over *Av+Interl*. The reduction is 20-34% for *Trfd*, *Ocean*, and *Bdna*, and 7% for *Flo52*. 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, *Av+In+Fifo* is likely to be more cost-effective than *Av+In+Div*.

The resulting contention in the reverse network is shown in Figure 12 under *Av+In+Fifo*. The addition of this global order to the processing of the messages should not affect the average latency of the messages in a significant way. The execution time reductions should mainly result from the faster way in which the processor can process the ordered vector elements, not from a reduction in the average latency of the messages. As it is, this optimization reduces the average latency for *Flo52*, *Ocean*, and *Bdna*, while it increases it for *Trfd*. The changes, however, have a fairly small absolute value.

We note that this optimization has a larger impact now than if it were applied to the original

Cedar network. This can be seen in Figure 13. The figure compares the processor stall time due to contention in *Baseline* to that in *Fifo*. *Fifo* is the *Baseline* network plus the optimization described in this section. We note that the difference between *Baseline* and *Fifo* (Figure 13) is smaller than the difference between *Av+Interl* and *Av+In+Fifo* (Figure 11) in both absolute and relative terms. The many sources of inefficiency in the original network prevent the ordering of messages from achieving its potential. It is, therefore, better to start with a relatively-optimized network.

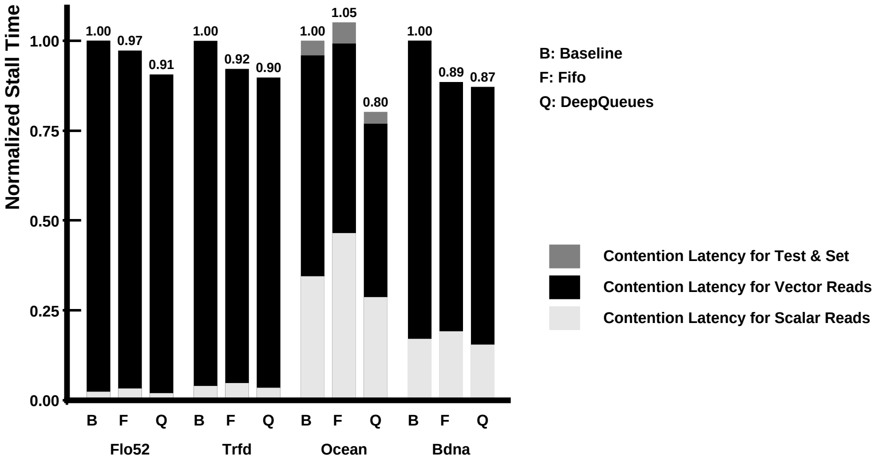

Figure 13: Decomposition of the time that processors stall due to message contention in the network and memory. For each application, we consider different network designs. The bars are normalized to *Baseline*.

Interestingly, we see from Figure 13 that *Ocean* runs slower with the *Fifo* optimization. This effect is caused by the different impact of *Fifo* on vectors and scalars. *Fifo* ensures order among the elements of a vector and therefore, for all applications of Figure 13, reduces the vector stall time. However, a more efficient vector message transfer implies that some scalar messages are slowed down. Indeed, scalar messages that are issued later than vector messages and that enter a switch via its highest priority inputs are now denied an unfair advantage. They have to wait their turn. As a result, for all the applications in Figure 13, the scalar time increases. In *Ocean*, where there are many scalars, the net result is a slowdown.

5.3.1 Combining All Techniques

Ensuring fairness in the processing of the messages in a switch has not succeeded in speeding up *Trfd* as much as *Div* did (Figure 11). This is because *Trfd* has the worst traffic convergence pattern: many of the vector accesses have a stride of 40, which forces consecutive messages to converge in the first stage of the reverse network. To see the impact of a very aggressive network that combines all the techniques studied, we now add the *Div* optimization to the latest network. The resulting network we call *Av+In+Fifo+Div*.

Div and *Fifo* should combine well since, as indicated before, they target different problems. Bars *Av+In+Fifo+Div* in Figure 11 show the processor stall time under the resulting network. We can see that, for our applications, the processor stall time decreases an average of 10%. In particular, we note that the decrease is 26% for *Trfd*. On average, the resulting stall time of the processors in the applications is now only 44% of *Baseline*.

To gain insight into the reason for the speedups, we now examine the resulting contention in the reverse network. We can see from the bars labeled *Av+In+Fifo+Div* in Figure 12 that, compared to *Av+In+Fifo*, the most obvious change is that the bulk of contention cycles shifts from the queues at the head or center of the network (*Q5* and *Q6*) to the queue at the tail (*Q7*). Very little contention now occurs in *Q5* and *Q6*: the traffic has been organized such that it flows well in the network and the messages collide only at the end. Note, in particular, that all the *Blocking* cycles present in *Q5* for *Av+In+Fifo* disappear. Overall, the remaining contention cycles are mostly queuing in *Q7* and *Confl_Other* in *Q5*. The former is caused by the periods of intensive traffic, which fill the *Q7* queues, while the later cannot be eliminated easily.

While this change successfully complements the message ordering optimization, especially for *Trfd*, the resulting network is more expensive. Most of the programs use stride one and, therefore, would not benefit as much as *Trfd* from the optimization. Since *Av+In+Fifo* is cheap and already quite effective, adding *Div* support in the network is probably not cost effective.

5.4 Deep Queues Help under Bursty Traffic

Before finishing this section, we consider the impact of a popular optimization, namely adding deep queues to the network [20]. Specifically, we add four extra entries to all queues. In the absence of contention, this extra buffering does not add more latency to the messages traversing the network.

The effect of this optimization is to smooth out the traffic fluctuations. When a burst of traffic occurs and messages converge to one port in the reverse network, the extra buffering absorbs some of this traffic. As a result, there is less chance of saturating the links and propagating the saturation backwards in the form of a saturation tree. The elimination of the saturation tree speeds up messages that are directed to other ports and that would otherwise get stuck in the tree. For this reason, this optimization can reduce the processor stall time.

The addition of deep queues to the *Av+In+Fifo* or *Av+In+Fifo+Div* networks does not change the execution time. Indeed, the new bars are not shown in Figure 11 because they are no different from the old ones. This is due to the already highly-optimized traffic transfers: the extra buffering is not needed that much in well-optimized networks. However, the addition of deep queues to the original Cedar network reduces processor stall time. This can be seen in bars *DeepQueues* in Figure 13. The reductions are in the 9-20% range. In the *DeepQueues* network, the queuing in *Q7* increases, while the queuing in *Q5* and *Q6* decreases. This is because the queues at the tail of the network are the first ones to fill up and, therefore, messages spend most of their contention cycles there. Overall, this optimization, while probably unnecessary for highly-optimized networks, is otherwise useful for less optimized networks.

6 Related Work

Many authors have used analytical models to predict the performance of multistage switching networks [4, 6, 7, 11, 13, 15, 22, 24]. These models are based on many different mathematical tools, including probability theorems, queuing models, markov chains, and mean value analysis. They can hardly model the complex nature of the traffic patterns revealed here.

Other authors based their work on detailed simulations of networks, often driven by synthetic loads. For example, Turner and Veidenbaum [21] used simulations to study vector machines of up to 512 CPUs, focusing on average-value parameters like speedup, efficiency, or latency. Turner [20] simulated architectural variations like different memory speeds or amount of buffering. While his work has a flavor similar to ours, he uses a synthetic workload instead of address traces of programs running on the machine. Furthermore, he does not model an existing machine as closely as we do. Partly because of the relatively low memory bandwidth of his system, the optimization that he identified as best was large memory input buffers. In this paper, we have shown the bottlenecks for networks in machines with relatively high memory bandwidth.

Lee *et al.* [10] studied the *Consecutive Request* traffic pattern, where each processor issues a sequence of requests to successive memory modules. This pattern is similar to the most important pattern discussed in this paper. They proposed the "Bit-Reverse Mapping", which is similar to the *Interl* optimization proposed here. Smith and Taylor [18] pointed out that non-blocking buffers substantially increase performance. Our results agree. Smith and Taylor [19] further discussed how vector disorder degrades performance. In this paper, we showed that, under bursty traffic, even relatively fair switches create such disorder. Gottlieb *et al.* [4] described a network that supports combining. Pfister and Norton [16] analyzed memory hot spots and tree saturation in the forward network. In this paper, we showed evidence for a different type of saturation, namely the one resulting from traffic convergence in the reverse network.

Noakes *et al.* [14] presented measurements of average-value parameters in a real machine. In this study, we use a hardware monitor to capture the full variation of the parameters instead of averages.

Finally, Granston *et al.* [5] and McAuliffe [12] use analytical models to point out that the fan-in of read bursts causes performance degradation in the reverse network. However, no detailed characterization is provided. In this paper, we have produced experimental evidence that tree saturation resulting from traffic convergence in the reverse network constitutes the bottleneck in the network studied.

7 Conclusion

While multistage switching networks have been studied extensively, detailed evaluations of their performance are rare. Indeed, analytical models, simulations with pseudo-synthetic loads, studies focused on high-level average parameters, and measurements of networks disconnected from the machine, all provide limited information. Consequently, the performance of these systems is less known than that of other aspects of computers that have been the object of many realistic empirical analyses. In this paper, we perform one such analysis by monitoring the omega network of the Cedar vector multiprocessor using hardware probes.

The analysis suggests that the performance of this network 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 enough 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 the most cost-effective ones. 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. Josep Torrellas is supported in part by an NSF Young Investigator Award.

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] H. Jiang and L. Bhuyan.
MVAMIN: Mean Value Analysis Algorithms for Multistage Interconnection Networks.
Journal of Parallel and Distributed Computing, pages 189–201, July 1991.

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

[8] 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.

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

[10] Y. Lee, S. Cheung, and J. Peir.
Consecutive Requests Traffic Model in Multistage Interconnection Networks.
In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 534–541, August 1991.

[11] T. Lin and L. Kleinrock.
Performance Analysis of Finite-Buffered Multistage Interconnection Networks with a General Traffic Pattern.
In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 68–78, May 1991.

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

[13] P. Mohapatra and C. Das.
A Queuing Model for Finite-Buffered Multistage Interconnection Networks.
In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I210–I213, August 1993.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] D. L. Willick and D. L. Eager.
An Analytical Model of Multistage Interconnection Networks.
In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 192–202, May 1990.

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

[24] H. Yoon, K. Y. Lee, and M. T.Liu.
Performance Analysis of Multibuffered Packet-Switching Networks in Multiprocessor Systems.
IEEE Transactions on Computers, 39(3):319–327, March 1990.