Debunking then Duplicating Ultracomputer Performance Claims by Debugging the Combining Switches

Eric Freudenthal and Allan Gottlieb
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*freudenthal, gottlieb*}*@nyu.edu*

Abstract

Memory system congestion due to serialization of hot spot accesses can adversely affect the performance of interprocess coordination algorithms. Hardware and software techniques have been proposed to reduce this congestion and thereby provide superior system performance. The combining networks of Gottlieb et al. automatically parallelize concurrent hot spot memory accesses, improving the performance of algorithms that poll a small number of shared variables.

We begin by debunking one of the performance claims made for the NYU Ultracomputer. Specifically, a gap in its simulation coverage hid a design flaw in the combining switches that seriously impacts the performance of busy wait polling in centralized coordination algorithms. We then debug the system by correcting the design and closing the simulation gap, after which we are able to duplicate the original claims of excellent performance on busy wait polling. Specifically our simulations show that, with the revised design, the Ultracomputer readers-writers and barrier algorithms achieve performance comparable to the highly regarded MCS algorithms.

1. Introduction

It is well known that the scalability of interprocess coordination can limit the performance of shared-memory computers. Since the latency required for coordination algorithms such as barriers or readers-writers *increases* with the available parallelism, their impact is especially important for large-scale systems. A common software technique used to minimize this effect is *distributed local-spinning* in which processors repeatedly access variables stored locally (in so-called NUMA systems, the shared memory is physically distributed among the processors).

An less common technique is to utilize special purpose coordination hardware such as the barrier network of [1], the CM5 Control Network [2], or the NYU "combining network" [8] and have the processors reference centralized memory. The idea behind the combining network is that when references to the same memory location meet at a network switch, they are combined into one reference that proceeds to memory. When the response to the combined messages reaches the switch, data held in the "wait buffer" is used to generate the needed second response. Other approaches to combining have been pursued as well, see for example [23] and [13].

The early work at NYU on combining networks showed their great advantage for certain classes of memory traffic, especially those with a significant portion of hot-spot accesses (a disproportionately large percentage of the references to one or a few locations). It is perhaps surprising that this work did not simulate the traffic generated when all the processors engage in busy-wait polling, i.e., 100% hot-spot accesses (but see the comments on [14] in Section 3). When completing studies begun a number of years ago of what we expected to be very fast centralized algorithms for barriers and readers-writers, we were particularly surprised to find that the combining network performed poorly in this situation. While it did not exhibit the disastrous serialization characteristic of accesses to a single

location without combining, the improvement was much less than expected and our algorithms were not nearly competitive with the MCS algorithms based on distributed local-spinning [17], [19]. Further investigation showed that our results were correct and the previous NYU claims were invalid for this important case.

The present paper briefly reviews combining networks and presents the debunking data just mentioned. We then debug the system by offering two fairly simple changes to the combining switches that fix the problem with busy wait polling. The first change is simply to increase the wait-buffer size. The second change is more subtle: The network is output-buffered and a trade-off exists involving the size of the output queues. Large queues are well known to improve performance for random traffic. However, we found that large queues cause poor polling performance. We therefore adapt the queue size to the traffic encountered: as more combined messages are present, the queue capacity is reduced. Together, these two simple changes have a dramatic effect on polling, and our centralized barrier and readers-writers algorithms become competitive with the commonly used MCS local-spin algorithms of Mellor-Crummey and Scott (some of which also benefit from the availability of combining), thereby duplicating the results claimed in the early NYU work.

There is an interesting explanation for the surprising observation that *crippled* hardware (reducing queue length) *improves* performance (of polling). The network switches can not combine three or more requests into one, and thus for many requests to be combined into a single request, it is necessary for combining to occur at many switches. The smaller queues increase "backpressure" and result in queuing at more switches and hence more combining. Modern techniques for caches (MSHRs) in a sense combine more than two requests. We explain in Section 2.2 why the corresponding idea is problematic for combining networks.

2. Background

Large-scale, shared-memory computation requires memory systems with bandwidth that scales with the number of processors. Multi-stage interconnection fabrics and interleaving of memory addresses among multiple memory units can provide scalable memory bandwidth for memory reference patterns whose addresses are uniformly distributed. Many variants of this architecture have been implemented in commercial and other research systems [12], [20], [22]. However, the serialization of memory transactions at each memory unit is problematic for reference patterns whose mapping to memory units is unevenly distributed. An important cause of non-uniform memory access patterns is *hot-spot* memory accesses generated by centralized busy-waiting coordination algorithms. The Ultracomputer architecture includes network switches [24] with logic to reduce this congestion by *combining* into a single request multiple memory transactions (e.g. loads, stores, fetch-and-adds) that reference the same memory address.[1]

The Ultracomputer combining switch design utilizes a variant of cut-through routing [10] that imposes a latency of one clock cycle when there is no contention for an outgoing network link. When there is contention, messages are buffered on queues associated with each output port. Investigations by Dias and Jump [4], Dickey [5], Liu [15], and others indicate that these queues significantly increase network bandwidth for large systems with uniformly distributed memory access patterns.

Systems with high degrees of parallelism can be constructed using these switches: Figure 1 illustrates an eight-processor system with three stages of routing switches interconnected by a shuffle-exchange [25] routing pattern. References to MM_3 are communicated via components drawn in **bold**.

Our simulation parameters are set to agree with the earlier NYU simulations (and the small-scale prototype built).[2] Specifically, our memory modules (MMs) can accept one request every 4 network cycles, whereas the switches can accept one request every 2 cycles on each input and can transmit one request every 2 cycles on each output. When the rate of requests to one MM exceeds its bandwidth, the

[1] Combining occurs only for messages that are buffered when the arrival rate exceeds the acceptance rate of the downstream queue. In particular, messages are *not* delayed solely to enable combining. Also note that a message can combine with any (not necessarily adjacent) enqueued message. Finally, the combining logic does not rely on associative search (however, de-combining does). [7]

[2] We briefly discuss parameter values more appropriate to current technology in Section 3.4.

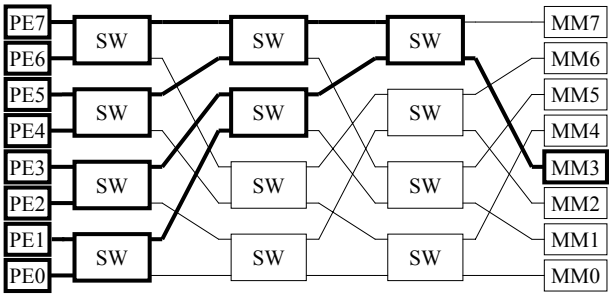

Fig. 1. Eight PE System with Hot-Spot Congestion to MM 3.

switch queues feeding it will fill. Since a switch cannot accept messages when its output buffers are full, a funnel-of-congestion will spread to the network stages that feed the overloaded MM and interfere with transactions destined for other MMs as well.[3] Thus unbalanced memory access patterns, such as hot spot polling of a coordination variable, can generate network congestion. Figure 1 illustrates contention among references to MM_3.

Ultracomputer switches combine pairs of memory requests accessing the same location into a single request to reduce the congestion generated by hot spot memory traffic. When the memory response subsequently arrives at this switch, it is *de-combined* into a pair of responses that are routed to the requesting PEs. To enable this de-combination, the switch uses an internal *wait buffer* to hold information found in the request until it is needed to generate the second response. Since combined messages can themselves be combined, this technique has the potential to reduce hot spot contention by a factor of two at each network stage.

a) Combining of Fetch-and-add: Our fetch-and-add based centralized coordination algorithms poll a small number (typically one) of "hot spot" shared variables whose values are modified using fetch-and-add.[4] Thus, as indicated above, it is crucial, when these algorithms are executed on large numbers of processors, not to serialize this activity. The solution employed is to include adders in the MMs (thus guaranteeing atomicity) and to combine concurrent fetch-and-add operations at the switches.

When two fetch-and-add operations referencing

[3]Pfister and Norton [21] called this funnel *tree saturation* and observed that access patterns containing only 5% hot spot traffic substantially increase memory latency.

[4]Recall that FAA(X,e) is defined to return the old value of X and atomically increment X by the value e.

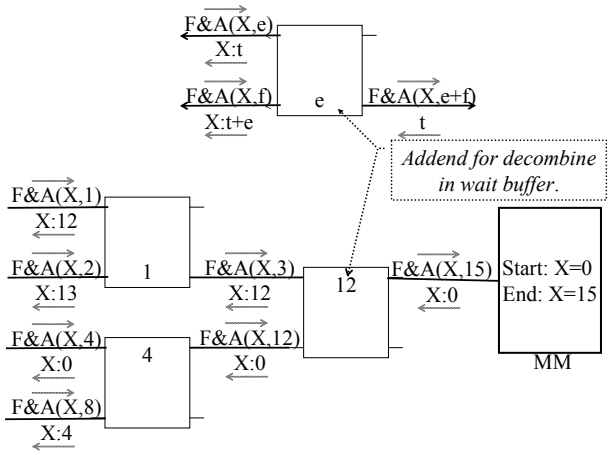

Fig. 2. Combining of Fetch-And-Adds at a single switch (above) and at multiple switches (below).

the same shared variable, say $FAA(X, e)$ and $FAA(X, f)$, meet at switch the combined request ***FAA(X,e+f)*** is transmitted and the value e is stored in the wait buffer. Load are transmitted as fetch-and-adds whose addends are zero and thus are also combinable.

Upon receiving $FAA(X, e+f)$, the MM updates X (to $X + e + f$) and responds with X. When the response arrives at the combining switch the latter transmits X to satisfy the request $FAA(X, e)$ and transmits $X + e$ to satisfy the request $FAA(X, f)$, thus achieving the same effect as if $FAA(X, e)$ was followed immediately by $FAA(X, f)$. This process is illustrated in the upper portion of Figure 2. The cascaded combining of 4 requests at two network stages is illustrated in the lower portion of the same figure.

Figure 3 illustrates an Ultracomputer combining switch. Each switch contains

- Two *Dual-input forward-path combining queues:* Entries are inserted and deleted in a FIFO manner and matching entries are combined, which necessitates an ALU to compute the sum $e + f$.
- Two *Dual-input reverse path queues:* Entries are inserted and deleted in a FIFO manner.
- Two *Wait Buffers:* Entries are inserted and associative searches are performed with matched entries removed. An included ALU computes $X + e$.

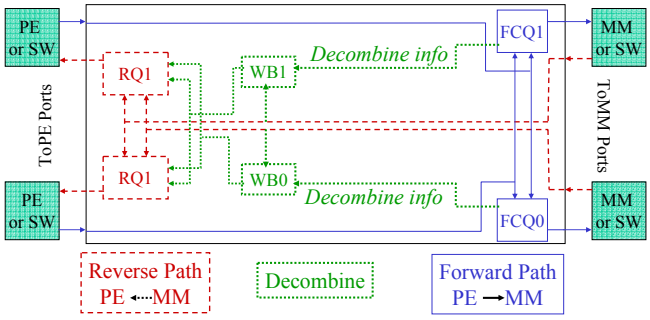

Fig. 3. Block Diagram of Combining 2-by-2 Switch *Notation: RQ: Reverse (ToPE) Queue, WB: Wait Buffer, FCQ: Forward (ToMM) Combining Queue*

2.1. When Combining Can Occur

Network latency is proportional to switch cycle times and grows with queuing delays. VLSI simulations showed that the critical path in a proposed Ultracomputer switch included the adder to form $e + f$ and the output drivers. To reduce cycle time, at the cost of restricting the circumstances in which combining could occur, the chosen design did not combine requests that were at the head of the output queue (and hence might be transmitted the same cycle as combined). This modification reduced the critical path timing to the max of the adder and driver rather than their sum. We call the modified design "decoupled" because the adder and driver are in a sense decoupled, and call the original design "coupled".

Since the head entry cannot be combined, we note that a decoupled queue requires at least three requests for combining to occur. We shall see that this trivial observation is important.

To enable the dual input queues to accept items on each input in one cycle, the queue was constructed from two *independent* single-input queues whose outputs are multiplexed. To achieve the maximum combining rate, we therefore require at least three requests in each of the single-input combining queues, which implies at least six in each dual-input combining queues. A more complicated dual-input decoupled combining queue, dubbed *type A* in Dickey [5] requires only three messages to achieve the maximum combining rate rather than six in the "type B" design we are assuming.

2.2. Combining Multiple Requests

Kroft [11] introduced Miss Status/Handler Registers (MSHRs) to implement lockup-free caches. These registers are also used to merge multiple requests for the same memory line. Similar techniques could be employed for combining switches and, were the degree of combining (i.e., the number of requests that could be merge) large, one might expect that good polling behavior would result. Indeed the early NYU work did consider greater-than-two way combining and sketched a VLSI design for one modest extension. However, this idea cannot be used to solve the polling problem. As observed by [14], if a large number of requests are combined into one, the *decombining* that results degrades performance due to serialization in the response (memory-to-processor) network.

There are several differences between MSHRs and network switches that might explain why the [14] observation does not apply to the former. Recall that MSHRs are located in the (multi-)processor node; whereas, combining switches are located in the network itself. Hence serialization in the MSHRs encountered by a memory response directly affects only one node. As observed by [21], however, delays in network switches (encountered either by requests or responses) can seriously degrade performance of many nodes, even those not referencing the hot-spot memory. The different locations of MSHRs and combining switches has another effect: A single path from a memory module to a processor node passes through multiple switches and thus a memory response might be subjected to multiple serialization delays when passing through a series of switches that must decombine requests.

3. Improving the Performance of Busy-Wait Polling

Figure 4 plots memory latency for two simulated systems of four to 2048 PEs with memory traffic containing 10% and 100% hot spot references. The latter typifies the traffic when processors are engaged in busy-wait polling and the local caches filter out instruction and private data references.

The first simulated system closely models the original Ultracomputer and is referred to as the *baseline* system. Observe that for 10% hot spot references, round-trip latency is only slightly greater than the minimum network transit time (one cycle per stage per direction) plus the simulated memory latency of two cycles.

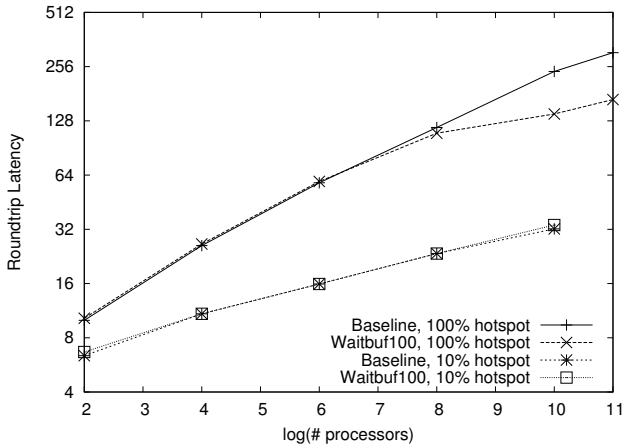

Fig. 4. Memory Latency for Ultraswitches with Wait Buffer Capacities of 8 and 100 messages for 10% and 100% Hotspot Traffic, 1 Outstanding Request/PE.

On larger systems, memory latency is substantially greater for the 100% hot spot load and can exceed 10 times the minimum. Since the combining switches simulated were expected to perform well for this traffic, the results were surprising, especially to the senior author who was heavily involved with the Ultracomputer project throughout its duration. High-performance centralized coordination cannot be achieved using these simulated switches.

The cause of the less than expected performance is two (previously unnoticed) design flaws in the combining switch design. The first is that the wait buffers were too small, the second is that, in a sense to be explained below, the combining queues were too *large*.

3.1. Increasing the Wait Buffer Capacity

The second system shown in Figure 4 contains switches with 100-entry wait buffers (feasible with today's technology). These larger switches reduce the latency for a 2048PE system from 306 to 168 cycles, an improvement of 45%. While this increased capacity helps, the latency of hot spot polling traffic is still seven times the latency of uniformly distributed reference patterns (24 cycles).

3.2. Adaptive Combining Queues

In order to supply high bandwidth for typical uniformly distributed traffic (i.e., 0% hot spot), it is important for the switch queues to be large. However, as observed in [14], busy wait polling (100% hot spot) is poorly served by these large queues, as we now describe.

For busy-wait polling, each processor always has one outstanding request directed at the same location.[5] The expectation was that, with N processors and hence $logN$ stages of switches, pairs would combine at each stage resulting in just one request (or perhaps more realistically, a few requests) reaching memory.

What actually happens is that the queues in switches near memory fill to capacity and the queues in the remainder of the switches are nearly empty. Since combining requires multiple entries to be present, it can only occur near memory. However, a *single* switch cannot combine an unbounded number of requests into one. Those fabricated for the Ultracomputer could combine only pairs so, if, for example, eight requests are queued for the same location, (at least) four requests will depart.[6]

Figure 5 illustrates this effect. Both plots are for busy wait polling and use the large, 100 entry wait buffers. The forward path combining queues in the left plot are modeled after the Ultracompputer design and contain four slots, each of which can hold either a request received by the switch or one formed by combining two received requests. The plot on the right is for the same switches with the queue capacity restricted so that if 2 combined requests are present, the queue is declared full even if empty slots remain. We call these queues adaptive and denote switch with these adaptive queues and large wait buffers as *improved*.

We compare the graphs labeled 10 (representing a system with 1024 PEs) in each plot. In the left plot, we find that combines occur at the maximal rate for the four (out of 10) stages closest to memory, occur at nearly the maximal rate for the fifth stage, and do not occur for the remaining five stages. The improved switches (the right plot) do better, combining at maximal rate for five stages and at 1/4 of the maximum for the sixth stage. In addition, since the queues are effectively smaller, the queuing delay is reduced.

[5]This is not quite correct: When the response arrives it takes a few cycles before the next request is generated. Our simulations accurately account for this delay.

[6]Alternate designs could combine more than two requests into one, but, as observed by [14], when this "combining degree" increases, congestion arises at the point where the single response is decombined into many (see Section 2.2).

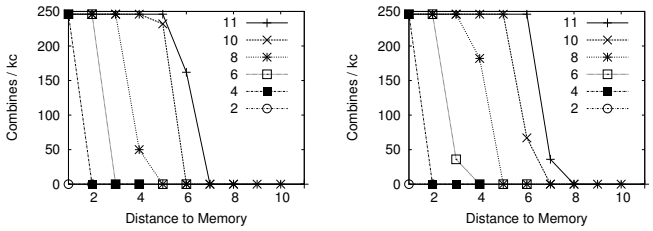

Fig. 5. Combining rate, by stage for simulated polling on systems of 2^2 to 2^{11} PEs. Wait buffers have capacity 100 and combining queues can hold 4 combined or uncombined messages. In the right plot the combining queues are declared full if two combined requests are present.

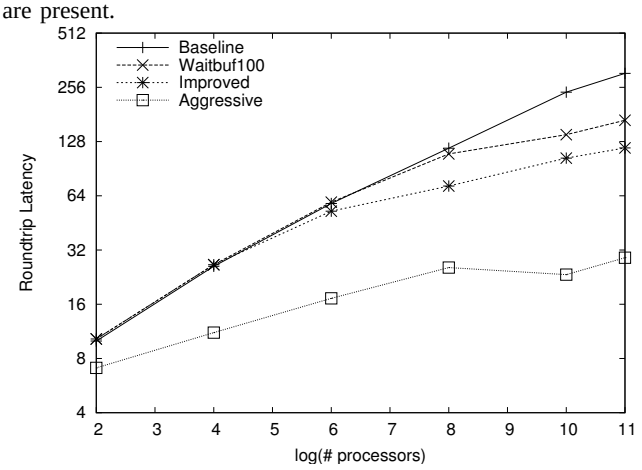

Fig. 6. Memory latency for simulated hot spot polling traffic, 4-2048 PEs.

Note that for uniformly distributed traffic without hot spots, combining will very rarely occur and the artificial limit of 2 combined requests per queue will not be invoked. We call this new combining queue design *adaptive* since the queues are full size for uniformly distributed traffic and adapt to busy wait polling by artificially reducing their size.

We see in Figure 6 that the increased combining rate achieved by the improved switches dramatically lowers the latency experienced during busy wait polling. For a 2048 PE system the reduction is from 168 cycles for a system with large (100 entry) wait buffers and the original queues to 118 cycles (five times the latency of uniform traffic) with the same wait buffers but adaptive combining queues. This is a reduction of over 30% and gives a total reduction of 61% when compared with the 306 cycles needed by the baseline switches. The bottom plot is for a more aggressive switch design described below. In Section 4 we shall see that centralized coordination algorithms executed on systems with adaptive combining queues and large wait buffers are competitive with the best distributed local-spinning alternatives.

Figure 7 compares the latency for 1024 PE systems with various switch designs and a range of accepted loads (i.e., processors can have multiple outstanding requests unlike the situation above for busy wait polling). The figure shows results for 1%, 20%, and 100% hot spot traffic. Similar results (not shown) were obtained for simulations with 0%, 40%, 60%, and 80% hot spot traffic. These results confirm our assertion that adaptive queues have very little effect for low hot spot rates and are a considerable improvement for high rates. Thus the Ultracomputer claims of good performance under a variety of loads are substantiated for the improved switches, but not for the original baseline design.

3.3. More Aggressive Combining Queues

Recall that we have been simulating decoupled type B switches in which combining is disabled for the head entry (to "decouple" the ALU and output drivers) and the dual input combining queues are composed of two independent single input combining queues with multiplexed outputs. We started with a "baseline design", used in the Ultracomputer, and produced what we refer to as the "improved design" having a larger wait buffer and adaptive combining queues. We also applied the same two improvements to type A switches having coupled ALUs and refer to the result as the "aggressive design" or "aggressive switches" For example, the lowest plot in Figure 6 is for aggressive switches. Other experiments not presented here have shown that aggressive switches permit significant rates of combining to occur in network stages near the processors. Also, as we will show in Section 4, the centralized coordination algorithms perform exceptionally well on this architecture, Although aggressive switches are the best performing, we caution the reader that our measurements are given in units of a switch cycle time and, without a more detailed design study, we cannot estimate the degradation in cycle time such aggressive switches might entail.

3.4. Applicability of Results to Modern Systems

The research described above investigates systems whose components have similar speeds, as was typical when this project began. During the intervening decade, however, logic and communication rates have increased by more than two orders of

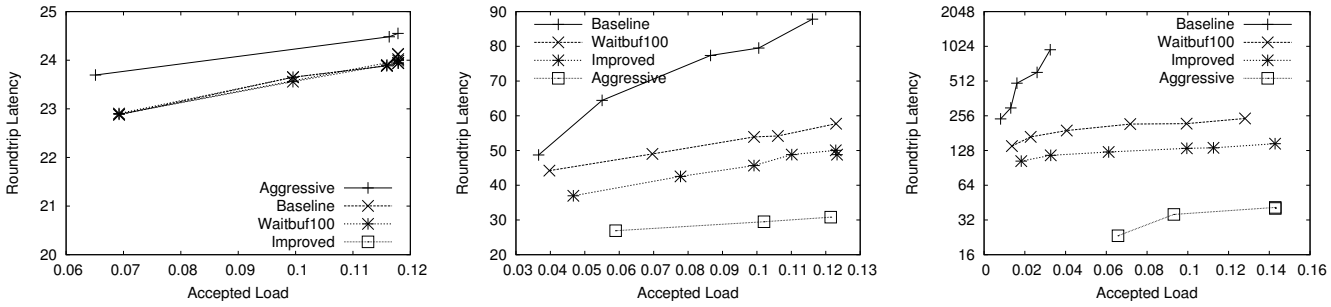

Fig. 7. Simulated Round-trip Latency over a Range of Offered Loads for 1% (left), 20% (middle) and 100% (right) Hot Spot Traffic.

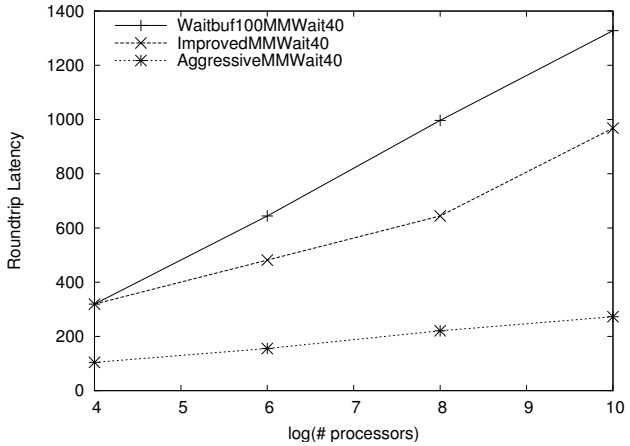

Fig. 8. Memory latency for hot spot polling on systems with MMs that can accept one message every 40 cycles.

magnitude while DRAM latency has improved by less than a factor of two.

In order to better model the performance obtainable with modern hardware, we increased the memory latency from two to thirty-eight cycles, and the interval between accepting requests from four to forty cycles.[7] These results are plotted in Figure 8 and indicate that the advantage achieved by the adaptive switch design is even greater than before.

4. Performance Evaluation of Centralized and MCS Coordination

A series of micro-benchmark experiments were performed to compare the performance of centralized fetch-and-add based coordination with state of the art MCS algorithms of Mellor-Crummey and Scott. The simulated hardware includes combining switches, which improves the performance of some MCS algorithms and is crucial for the centralized algorithms, as well as NUMA memory, which is important for MCS and not exploited by the centralized algorithms. We present results for readers-writers and barrier coordination. These simulations plus others appeared in the junior author's dissertation [6].

4.1. Barrier Synchronization

Barrier coordination is often used in algorithms that require coarse-grain synchronization between asynchronous *supersteps* [26]. Our micro-benchmark study, presented in Figure 9 considers three superstep bodies. The *intense* experiment, in which Superstep bodies are empty, measures the latency of synchronization. To simulate programs where processors execute roughly synchronously, each processor executing our *uniform* experiment issues thirty shared memory references during each Superstep. In contrast, half of the processors executing our *mixed* experiment issue thirty references to shared variables during each Superstep, and the other half issue only fifteen.

A best-of-breed centralized fetch-and-add based algorithm was simulated on four architectures. The one without combining is labeled NoComb in Figure 9. The three with combining use the original Ultracomputer (*Baseline*) switches and the *Improved* and *Agressive* switches described earlier. The MCS *Dissemination* barrier [17], which does not generate hot spot traffic and is intended for NUMA systems, is simulated on a NUMA system without combining.

As expected, the availability of combining substantially decreases superstep latency for the centralized algorithms in all experiments. The improved switches (but not the original design) match the performance of MCS and the aggressive switches exceed it (but may entail a longer cycle time).

4.2. Readers-Writers Coordination

Many algorithms for coordinating readers and writers [3] have appeared. A centralized algorithm is presented in [8] that, on systems capable of

[7] Standard caching and sub-banking techniques can mitigate the effect of slow memory.

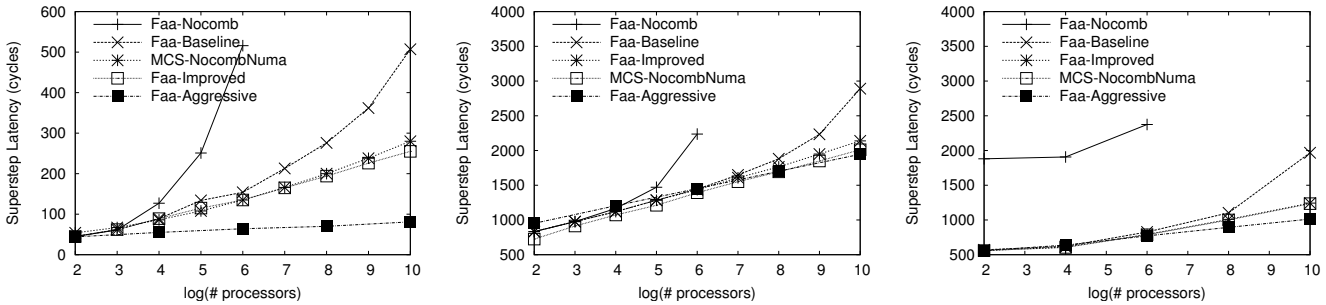

Fig. 9. Superstep latency, in cycles, for *intense* (left), *uniform* (middle), and *mixed* (right) workloads. (lower latency is better)

combining fetch-and-add operations, does not serialize readers in the absence of writers. However, no commercial systems with this hardware capability have been constructed, and in their absence, alternative "distributed local-spin" MCS algorithms have been developed [18]. Although the MCS algorithms do serialize readers, they minimize hot spot traffic by having each processor busy-wait on a shared variable stored in memory co-located with this processor. This NUMA memory organization results in local busy waiting that does not contribute to or encounter network congestion.[8]

The most likely cause of unbounded waiting in any reader-writer algorithm is that a continual stream of readers can starve all writers. The standard technique of giving writers priority eliminates this possibility (but naturally permits writers starving readers). In this section we present a performance comparison of "best of breed" centralized and MCS writer-priority reader-writer algorithms each executed on a simulated system with the architectural features it exploits.

The centralized reader-writer algorithm [6] issues only a single shared-memory reference (a fetch-and-add) when the lock is uncontested. The MCS readers-writers algorithm [18] is commonly used on large SMP systems. This algorithm is a hybrid of centralized and distributed approaches. Central state variables, manipulated with various synchronization primitives, are used to count the number and type of lock granted and to head the lists of waiting processors. NUMA memory is used for busy waiting, which eliminates network contention.

[8]Some authors use the term NUMA to simply mean that the memory access time is non-uniform: certain locations are further away than others. We use it to signify that (at least a portion of) the shared memory is distributed among the processors, with each processor having direct access to the portion stored locally.

4.3. Experimental Results

The algorithms are roughly comparable in performance: The centralized algorithms are superior except when only writers are present. Recall that an ideal reader lock, in the absence of contention, yields linear speedup; whereas an ideal writer exhibits no *slowdown* as parallelism increases. When there are large numbers of readers present, the centralized algorithm, with its complete lack of reader serialization, thus gains an advantage, which is greater for the aggressive architecture.

The scalability of locking primitives is unimportant when they are executed infrequently with low contention. Our experiments consider the more interesting case of systems that frequently request reader and writer locks. For all experiments each process repeatedly:

- Stochastically chooses whether to obtain a reader or writer lock.[9]
- Issues *Work* non-combinable shared memory references distributed among multiple MMs,
- Releases the lock.
- Waits *Delay* cycles.

For simplicity we assume one process per processor. In order for every measurement shown on a single plot to represent equivalent contention from writers, we fix the value of E_W, the expected number of writers, and thus the probability that each process chooses to be a writer is E_W divided by the number of processes.

Each experiment measures the rate that locks are granted over a range of system sizes (higher values are superior). Two classes of experiments were performed: Those classified "I" represent *intense* synchronization in which each process request and release locks at the highest rate possible, $Work =$

[9]The simulated random number generator executes in a single cycle.

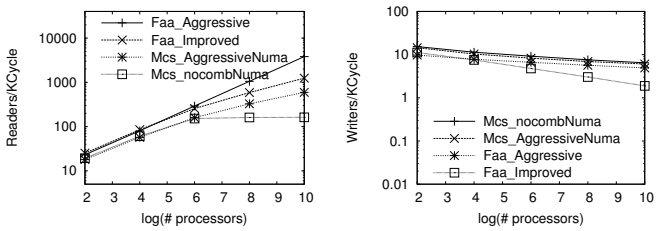

Fig. 10. Experiment R, All Readers (left), All Writers (right)

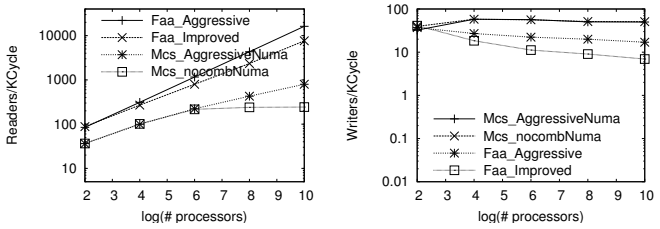

Fig. 11. Experiment I, All Readers (left), All Writers (right)

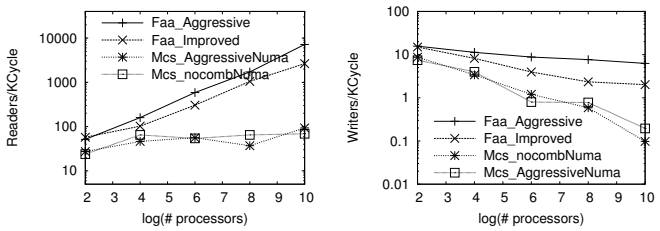

Fig. 12. Experiment I, $E_W = 1$

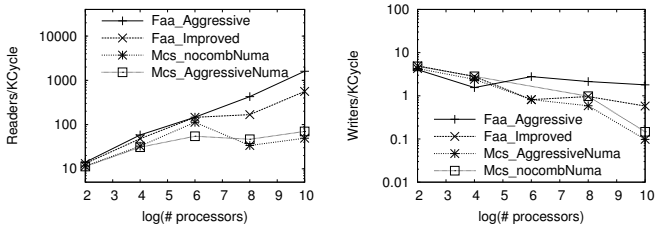

Fig. 13. Experiment R, $E_W = 1$

$Delay = 0$. Those classified "R" are somewhat more *realistic*, $Work = 10$ and $Delay = 100$.

a) All Reader Experiments, $E_W = 0$: The left-side charts in Figures 10 and 11 present results from experiments where all processes are readers. The centralized algorithm requires a single hot-spot memory reference to grant a reader lock in the absence of writers. In contrast, the MCS algorithm generates accesses to centralized state variables and linked lists of requesting readers. Not surprisingly, the centralized algorithm has significantly superior performance in this experiment, and MCS benefits from combining.

b) All-Writer Experiments: The right-side charts in Figures 10 and 11 present results from experiments where all processes are writers, which must serialize and therefore typically spend a substantial period of time busy-waiting. The MCS algorithm has superior performance in these experiments.

Since writers enforce mutual exclusion, no speedup is possible as the system size increases. Indeed one expects a slowdown due to the increased average distance to memory, as described in [18]. The MCS algorithm issues very little hot spot traffic when no readers are present and thus does not benefit from combining in these experiments.

c) Mixed Reader and Writer Experiments: Figures 12 through 15 present results of experiments with both readers and writers. In the first set, $E_W = 1$ (this lock will have substantial contention from writers) and in the second set $E_W = 0.1$ (a somewhat less contended lock).

The rate at which the centralized algorithm grants reader locks increases linearly with system size for all these experiments and, as a result, significantly exceeds the rate granted by MCS for all large system experiments.

5. Open Questions

5.1. Extending the Adaptive Technique

Our adaptive technique sharply reduces queue capacity when a crude detector of hot spot traffic is triggered. While this technique reduces network latency for hot spot polling, it might also be triggered by mixed traffic patterns that would perform better

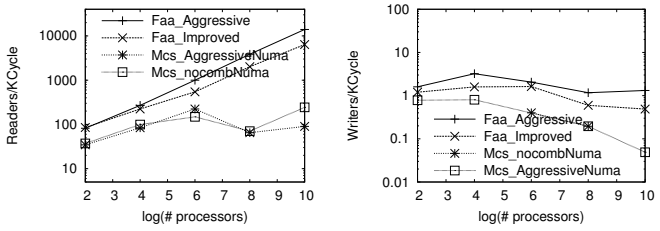

Fig. 14. Experiment I, $E_W = 0.1$

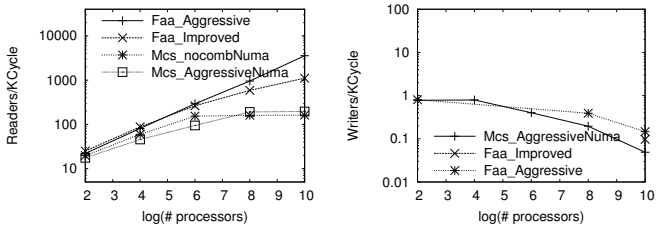

Fig. 15. Experiment R, $E_W = 0.1$

with longer queues. We have neither witnessed nor investigated this effect, which might be mitigated by more gradual adaptive designs that variably adjust queue capacity as a function of a continuously measured rate of combining.

5.2. Generalization of Combining to Internet Traffic

The tree saturation problem due to hot spot access patterns is not unique to shared memory systems. Congestion generated by flood attacks and flash crowds [9] presents similar challenges for Internet Service Providers. In [16] Mahajan et al. propose a technique to limit the disruption generated by hot spot congestion on network traffic with overlapping communication routes. In their scheme, enhanced servers and routers incorporate mechanisms to characterize hot spot reference patterns. As with adaptive combining, upstream routers are instructed to throttle the hot spot traffic in order to reduce downstream congestion.

Hot spot requests do not benefit from this approach, however combining may provide an alternative to throttling. For example, the detection of hot spot congestion, could trigger deployment of proxies near to network entry points, potentially reducing downstream load and increasing the hot spot performance. This type of combining is service-type specific and therefore service-specific strategies must be employed. Dynamic deployment of such edge servers requires protocols for communicating the characteristics of hot spot aggregates to servers, and secure mechanisms to dynamically install and activate upstream proxies.

5.3. Combining and Cache-Coherency

Cache coherence protocols typically manage shared (read-only) and exclusive (read-write) copies of shared variables. Despite the obvious correspondence between cache coherence and the readers-writers coordination problem, coherence protocols typically serialize the transmission of line contents to individual caches. The SCI cache coherence protocol specifies a variant of combining fetch-and-store to efficiently enqueue requests. However, data distribution and line invalidation on network connected systems is strictly serialized. Extensions of combining may be able to parallelize cache fill operations. Challenges for such schemes would include the development of an appropriate scalable directory structure that is amenable to (de)combinable transactions.

6. Conclusions

An investigation of the surprisingly poor performance attained by the Ultracomputer's combining network when presented with 100% hot spot traffic has revealed in a gap in the previous simulations that hid flaws in the combining switch design. Closing the simulation gap debunks the old claims of good performance on busy-waiting coordination. Fortunately the switch design was not hard to debug. The first improvement is to simply increase the size of one of the buffers present. The more surprising second improvement is to artificially *decrease* the capacity of combining queues during periods of heavy combining. These adaptive combining queues better distribute the memory requests across the stages of the network, thereby increasing the overall combining rates and lowering the memory latency.

Using the debugged switches, we then compared the performance of centralized algorithms for the readers writers and barrier synchronization problems with those of the widely used MCS algorithms. The latter algorithms reduce hot spots by polling only variables stored in memory that is co-located with the processor in question.

Our simulation studies of these algorithms have yielded several results: First, the MCS and centralized barrier algorithms have roughly equal performance. Second, the MCS readers-writers algorithm benefits from combining. Third, when no readers are present, the MCS algorithm outperforms the centralized algorithm. Finally, when readers are present, the results are reversed. In summary, MCS and the centralized algorithms are roughly equal in performance. That is, with debugged switches we are able to duplicate the previous claims of good performance for busy-wait coordination.

Switches capable of combining memory references are more complex than non-combining switches. An objective of the previous design efforts was to permit a cycle time comparable to a similar non-combining switch. In order to maximize switch clock frequency, a (type B, uncoupled) design was selected that can combine messages only if they

arrive on the same input port and is unable to combine a request at the head of an output queue. We also simulated an aggressive (type A, coupled) design without these two restrictions. As expected it performed very well, but we have not estimated the cycle-time penalty that may occur.

References

[1] Carl J. Beckmann and Constantine D. Polychronopoulos. Fast barrier synchronization hardware. In *Proc. 1990 Conference on Supercomputing*, pages 180–189. IEEE Computer Society Press, 1990.

[2] Thinking Machines Corp. *The Connection Machine CM-5 Technical Summary*, 1991.

[3] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Comm. ACM*, 14(10):667–668, October 1971.

[4] Daniel M. Dias and J. Robert Jump. Analysis and simulation of buffered delta networks. *IEEE Trans. Comp.*, C-30(4):273–282, April 1981.

[5] Susan R. Dickey. *Systolic Combining Switch Designs*. PhD thesis, Courant Institute, New York Univeristy, New York, 1994.

[6] Eric Freudenthal. *Comparing and Improving Centralized and Distributed Techniques for Coordinating Massively Parallel Shared-Memory Systems*. PhD thesis, NYU, New York, June 2003.

[7] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir. The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comp.*, pages 175–189, February 1983.

[8] Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors . *ACM TOPLAS*, pages 164–189, April 1983.

[9] J. Jung, B. Krishnamurthy, and M. Rabinovich. ”flash crowds and denial of service attacks: Characterization and implications for cdns and web sites”. In *Proc. International World Wide Web Conference*, pages 252–262. ”IEEE”, May ”2002”.

[10] P. Kermani and Leonard Kleinrock. Virtual Cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[11] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Int’l Symposium for Computer Architecture*, pages 81–87, 1981.

[12] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. *ACM SIGARCH Computer Architecture News*, 1997.

[13] Alvin R. Lebeck and Gurindar S. Sohi. Request combining in multiprocessors with arbitrary interconnection networks. *IEEE, TPDS*, November 1994.

[14] Gjyngho Lee, C. P. Kruskal, and D. J. Kuck. On the Effectiveness of Combining in Resolving ‘Hot Spot’ Contention. *Journal of Parallel and Distributed Computing*, 20(2), February 1985.

[15] Yue-Sheng Liu. *Architecture and Performance of Processor-Memory Interconnection Networks for MIMD Shared Memory Parallel Processing Systems*. PhD thesis, New York University, 1990.

[16] R. Mahajan, S. Bellovin, S. Floyd, J. Vern, and P. Scott. Controlling high bandwidth aggregates in the network, 2001.

[17] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[18] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared Memory Multiprocessors. *ACM Trans. Comput. Systems*, 9(1):21–65, 1991.

[19] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proc. ISCA IV*, pages 269–278, 1991.

[20] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfielder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss. The ibm research parallel processor prototype (rp3). In *Proc. ICPP*, pages 764–771, 1985.

[21] Gregory F. Pfister and V. Alan Norton. “Hot Spot” Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, c-34(10), October 1985.

[22] Randall D. Rettberg, William R. Crowther, and Phillip P. Carvey. The Monarch Parallel Processor Hardware Design. *IEEE Computer*, pages 18–30, April 1990.

[23] Steven L. Scott and Gurindar S. Sohi. Using feedback to control tree saturation in multistage interconnection networks. In *Proc. Int’l Symposium for Computer Architecture*, pages 167–176, May 1989.

[24] Marc Snir and Jon A. Solworth. Ultracomputer Note 39, The Ultraswitch - A VLSI Network Node for Parallel Processing. Technical report, Courant Institute, New York University, 1984.

[25] Harold Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computing*, C-25(20):55–65, 1971.

[26] Leslie G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, 1990.