Comparing the Performance of Centralized and Distributed Coordination on Systems with Improved Combining Switches

NYU Computer Science Technical Report TR2003-849

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. The widely used "MCS" distributed-spin algorithms take a software approach: they reduce hot spot congestion by polling only variables stored locally.**

Our investigations detected performance problems in existing designs for combining networks and we propose mechanisms that alleviate them. Simulation studies described herein indicate that a centralized readers writers algorithms executed on the improved combining networks have performance at least competitive to the MCS algorithms.

I. 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 [2], the CM5 Control Network [3], or the "combining network" of [1] and have the processors reference centralized memory. The idea behind combining 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.

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 [7] in section III). 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 dis-

astrous 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 those based on distributed local-spinning [MCS].

In this paper we briefly review combining networks and present the surprising data just mentioned. We then present two fairly simple changes to the NYU combining switches that enable the system to perform much better. The first change is 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 propose adapting 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 local-spin algorithms of Mellor-Crummey and Scott (some of which also benefit from the availability of combining).

II. 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 [9], [10], [18]. 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 [16] 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.

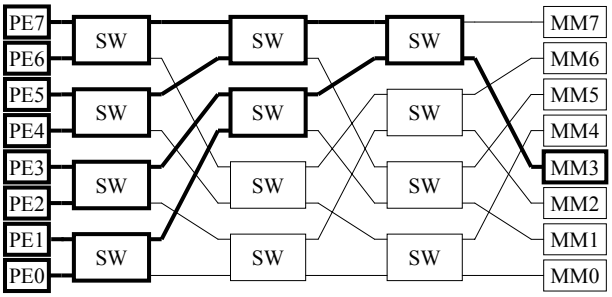

Fig. 1. Eight PE System with Hot-Spot Congestion to MM 3.

The Ultracomputer combining switch design utilizes a variant of cut-through routing [13] that imposes a latency of one clock cycle when there is no contention for an outgoing network link. When there is contention for an output port, messages are buffered on queues associated with each output port. Investigations by Dias and Jump [4], Dickey [5], Liu [14], 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 $d = 3$ stages of routing switches interconnected by a shuffle-exchange [19] routing pattern. References to MM_3 are communicated via components denoted using **bold** outlines.

1) An Overview of Combining: We assume that a single memory module (MM) can initiate at most one request per cycle.[1] 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. When the rate of requests to one MM exceeds its bandwidth, the 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.[2]

Ultracomputer switches combine pairs of memory requests accessing the same location into a single re-

[1]This restriction applies to each bank if the MM is itself composed of banks that can be accessed concurrently. The MM design simulated in our experiments can accept one request every four cycles.

[2]Pfister and Norton [8] called this funnel *tree saturation* and observed that access patterns containing only 5% hot spot traffic substantially increase memory latency

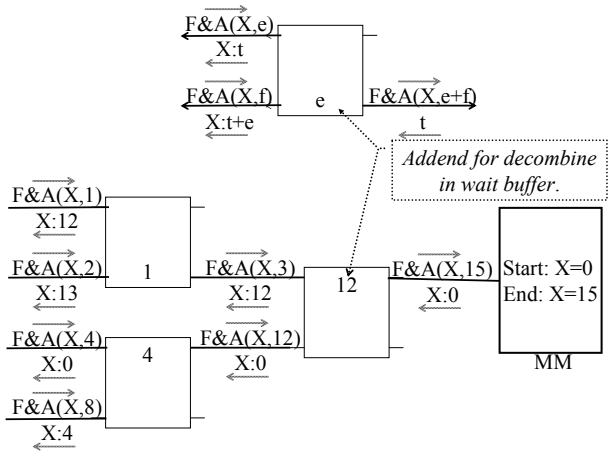

Fig. 2. Combining of Fetch-And-Adds at a single switch (above) and at multiple switches (below).

quest 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.

2) 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.[3] Thus, as indicated above, it is crucial in a design supporting 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 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 and fetch-and-add operations directed towards the same hot spot variable can be combined if loads are transmitted as fetch-and-adds whose addends are zero.

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 $T + e$ 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.

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

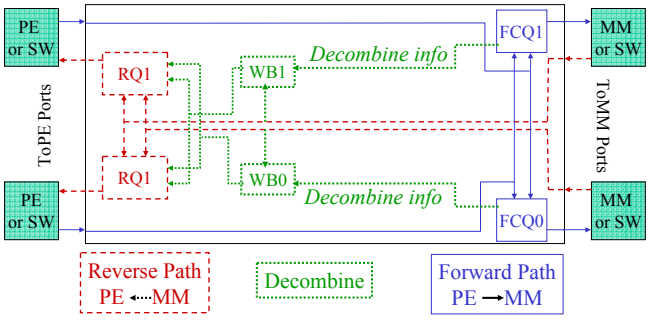

Fig. 3. Block Diagram of Combining 2-by-2 Switch*Notation: RQ: Reverse (ToPE) Queue, WB: Wait Buffer, FCQ: Forward (ToMM) Combining Queue*

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 $X + e$.

A. 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.

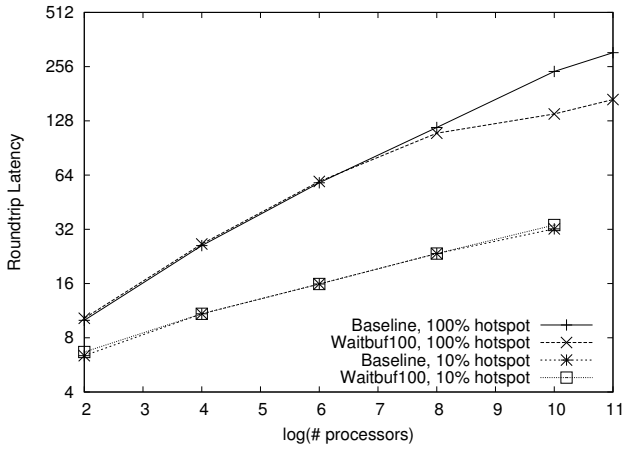

Fig. 4. Memory Latency for Ultraswitches with Wait Buffer Capacities of 8 and 100 messages for 10% and 100% Hotspot Traffic, 1 Outstanding Request/PE.

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 sounding observation is important.

To enable the dual input queues to accept items on each input in one cycle, the queue is 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.

III. Improving the Performance of Busy-Wait Polling

Figure 4 plots memory latency for 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 simulated multiprocessor approximates the Ultracomputer design. Observe that for 10% hot spot references the latency is only slightly greater than the minimum network transit time (1 cycle per stage per direction) plus the simulated memory latency of 2 cycles.

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. It is clear that our current objective of exhibiting high-performance centralized coordination algorithms cannot be achieved using these simulated switches.

We have discovered that there were two design flaws in the combining switch design. The first is that the wait buffer was too small, the second was that, in a sense to be explained below, the combining queues were too *large*.

A. Increasing the Wait Buffer Capacity

The fabricated switches had wait buffers capable of holding 8 entries. We see in 4 that increasing the capacity to 100 entries (feasible with today's technology) reduces the latency for a 2048PE system from 306 to 168 cycles an improvement of 45%. While this improvement certainly helps, the latency of hot spot polling traffic is seven times the latency of uniformly distributed reference patterns (24 cycles).

B. 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 by [7], busy wait polling (100% hot spot), however, 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.[4] Our expectation was that with N processors and hence $logN$ stages of switches, pairs would combine at each stage resulting in just one (or perhaps more realistically a few) request reaching memory.

[4]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.

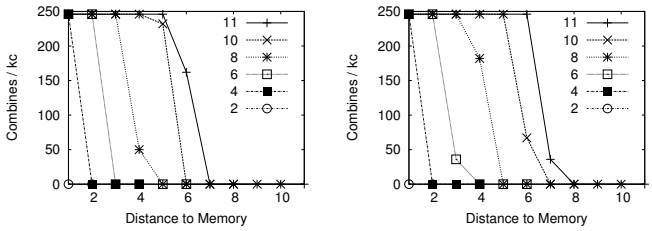

Fig. 5. Combining rate, by stage for simulated polling on systems of 2^2 to 2^{11} PEs. Wait buffers have capacity 100 and combining queues can hold 4 combined or uncombined messages. In the right plot the combining queues are declared full if two combined requests are present.

What happens instead 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.[5]

Figure 5 illustrates this effect. Both plots are for hot spot polling. The left plot is for simulated switches modeled after those fabricated for the Ultracomputer, but with the wait buffer size increased to 100. Each queue has four slots each of which can hold 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 compare the graphs labeled 10 (representing a system with 1024 PEs) in each plot.

In the left plot with the original switches, 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 restricted switches do better, combining at maximal rate for five stages and at 1/4 of the maximum for the sixth stage.

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.

[5]Alternate designs could combine more than two requests into one, but, as observed by [7], when this "combining degree" increases, congestion arises at the point where the single response is decombined into many.

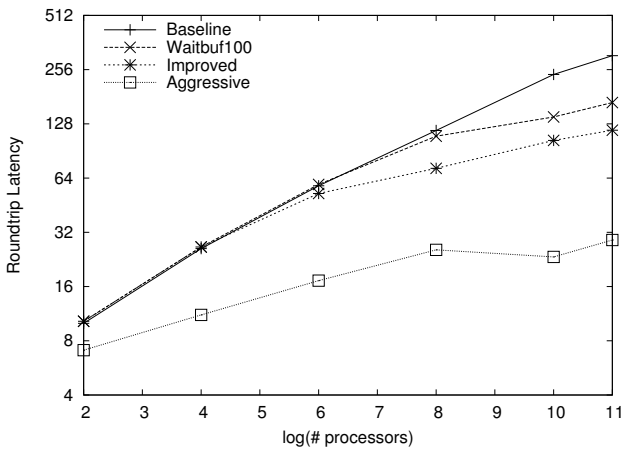

Fig. 6. Memory latency for simulated hot spot polling traffic, 4-2048 PEs.

We see in Figure 6 that the increased combining rate achieved by the adaptive queues 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 two plots are for more aggressive switch designs that are discussed below. In Section IV we shall see that centralized coordination algorithms executed on systems with adaptive queues and large wait buffers are competitive with their 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). 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.

C. 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

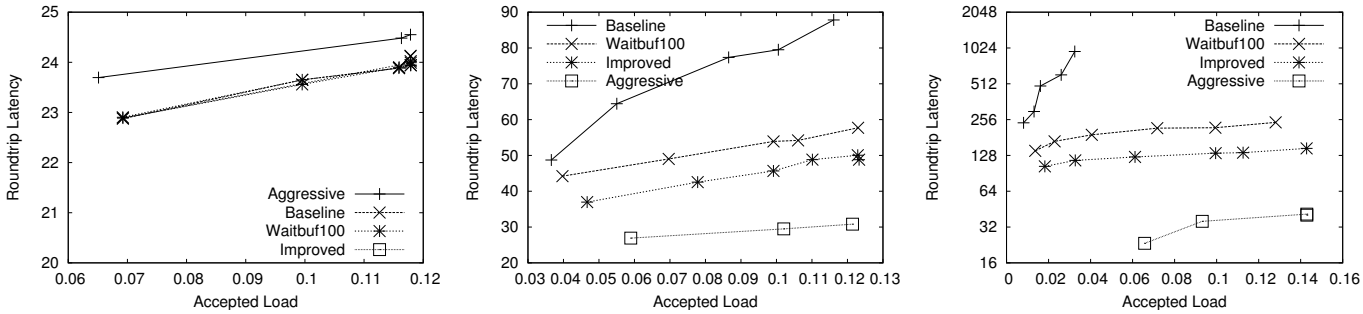

Fig. 7. Simulated Round-trip Latency over a Range of Offered Loads for 1% (left), 20% (middle) and 100% (right) Hot Spot Traffic.

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 IV, 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.

D. 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 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. These results are plotted in Figure 8 and indicate that the advantage achieved by the adaptive switch design is even greater than before.

Various techniques can mitigate the effect of slow

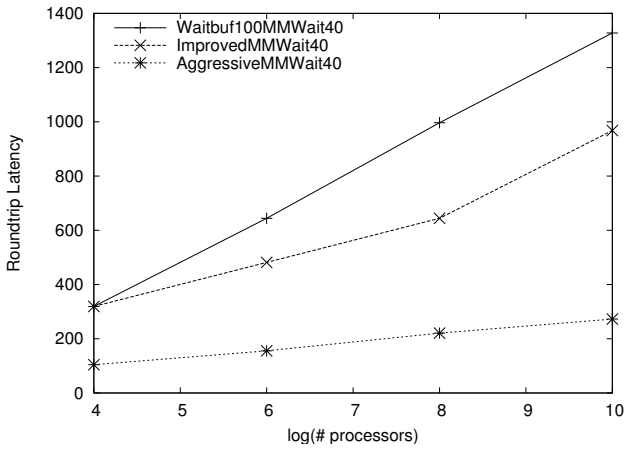

Fig. 8. Memory latency for hot spot polling on systems with MMs that can accept one message every 40 cycles.

memory. For example, having multiple sub-banks per MM would increase memory bandwidth and improve the performance on uniformly distributed traffic, and caches at the MM are well suited to the temporal locality of hot spot accesses.

IV. Performance Evaluation of Centralized and MCS Reader-Writer Coordination

Many algorithms for coordinating readers and writers have appeared. Courtois and Heyman [17] introduced the problem and presented a solution that serializes the entry of readers as well as writers. More importantly, these algorithms generate hot spot polling traffic and therefore suffer slowdowns due to memory system congestion on systems that do not combine hot spot references.

A centralized algorithm is presented in [1] that, on systems capable of combining fetch-and-add operations, eliminates the serialization of readers in the absence of writers. However, no commercial systems with this capability have been constructed, and in their absence, alternative "distributed local-spin" MCS algorithms have been developed [11]

that 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.[6]

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.[7] 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.

Roughly similar results hold for barrier algorithms. The interested reader is referred to [6].

A. Centralized Algorithms for Readers and Writers

Figure 9 presents a centralized reader-writer lock using fetch-and-add. This algorithm issues only a single shared-memory reference (a fetch-and-add) when the lock is uncontested. We know of no other algorithm with this desirable property. A more complete description of the techniques employed by this algorithm appears in [6].

B. Hybrid Algorithm of Mellor-Crummey and Scott

The writer-priority readers-writers algorithm of Mellor-Crummey and Scott [11] is commonly used on large SMP systems. This algorithm, commonly referred to as MCS, 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 at any time and to head the lists of waiting processors. NUMA memory is used for busy waiting, which eliminates network contention.

C. Overview of Experimental Results

A series of micro-benchmark experiments were performed to evaluate the performance of centralized fetch-and-add based coordination with the hybrid algorithm of Mellor-Crummey and Scott. The simulated hardware includes combining switches, which improves the performance of MCS and is crucial for the centralized algorithm, as well as NUMA memory, which is important for MCS and not used by the centralized algorithm. All the switch designs described above plus others were simulated in the junior author's dissertation [6]. A summary of the results is presented below.

```
shared int C = 0;

faaReaderLock() {
    for (;;) {
        int c = faa(C,1);          // request
        if (c < K)                 // no writers?
            return true;           //  succeeded!
        c = faa(C, -1);            // cancel reqeust
        while (c >= K) {           // while a writer is active...
            c = C;                 // read C
}   }   }

faaReaderUnlock() { faa(C, -1); }

bool faaWriterLock() {
    for (;;) {
        int c = faa(C, K);         // request
        if (c < K) {               // am next writer?
            if (c)                 // any readers?
                while ((C % K) != 0); // wait for readers
            true;                  // succeeded
        }
        c = faa(C, -K) - K;        // conflict: must decrement & retry
        while (c >= K) {           // while a writer is active...
            c = C;                 // read C
}   }   }

faaWriterUnlock() { faa(C, -K); }
```

Fig. 9. Fetch-and-add Readers-Writers Lock

Neither algorithm dominates the other: 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.

D. Experimental Framework

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 (at a fixed, but parameterized, probability) whether to obtain a reader

[6] Some authors use the term NUMA to simply mean that the memory access 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.

[7] But naturally permits writers starving readers.

or writer lock.[8]

- Issues a fixed sequence of non-combinable shared memory references distributed among multiple MMs—the "simulated work".
- Releases the lock.
- Waits a fixed delay.

Each experiment measures the rate that locks are granted over a range of system sizes (higher values are superior). In order to generate equivalent contention from writers in each plot, the expected number of writers E_W is held constant for all system sizes. Thus, the probability of a process becoming a writer is $E_W/Parallelism$.

Each micro-benchmarks has four parameters:

- PAR: the number of processors in the simulated system.
- E_w: The expected number of writers.
- $Work$: The number of shared accesses executed while holding a lock.
- $Delay$: The number of cycles a process waits after releasing a lock.

Two classes of experiments were performed: Those classified "I" measure the cost of *intense* synchronization in which each processors request and release locks at the highest rate possible, $Work = Delay = 0$. Those classified "R" are somewhat more *realistic*, $Work = 10$ and $Delay = 100$.

1) 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 algorithms have significantly superior performance in this experiment. For some configurations, an order of magnitude difference is seen. We also observe that MCS does benefit significantly from combining. Even when only MCS uses the aggressive switches, the centralized algorithm is superior. Naturally, the differences are greater in the "intense" experiments.

2) 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.

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

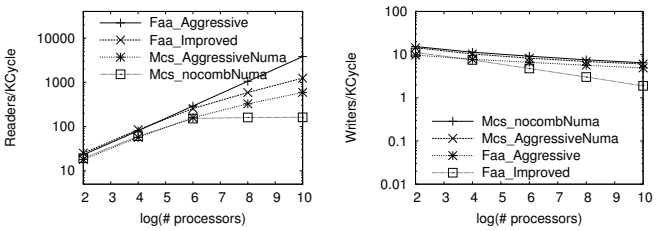

Fig. 10. Experiment R, All Readers (left), All Writers (right)

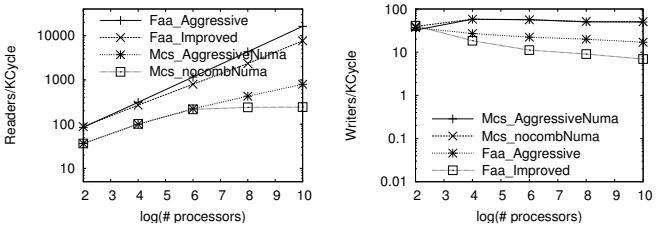

Fig. 11. Experiment I, All Readers (left), All Writers (right)

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. The anomalous speedup observed for MCS between 4 and 16 processors is due to the algorithm's decoupling of queue insertion and lock passing. This effect is described in [11]. As explained in [6] the MCS algorithm issues very little hot spot traffic when no readers are present and thus does not benefit from combining in these experiments.

3) Mixed Reader-Writer Experiments: Figures 12 through 15 present results of experiments with both readers and writers. In the first set, there is on average 1 writer present (this lock will have substantial contention from writers) and in the second set 0.1 (a somewhat less contended lock).

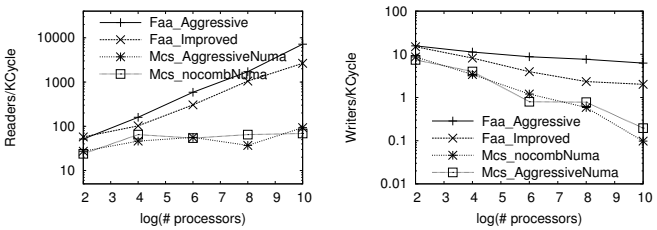

Fig. 12. Experiment I, $E_W = 1$

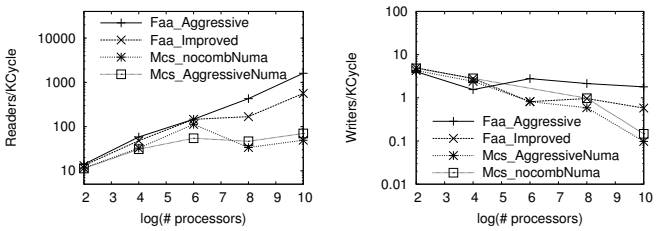

Fig. 13. Experiment R, $E_W = 1$

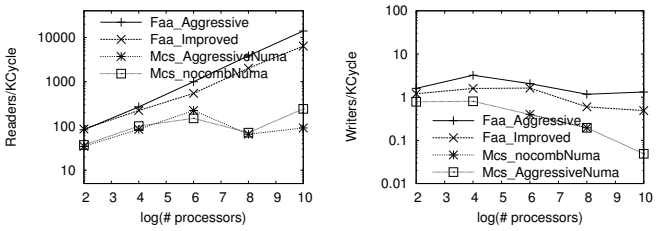

Fig. 14. Experiment I, $E_W = 0.1$

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. Even without the aggressive switches, the difference normally exceeds and order of magnitude.

Even though writers have priority over readers, there are vastly fewer writer locks granted with these values of E_w. Here also the centralized algorithm outperforms MCS, often by an order of magnitude. For some experiments, writer lock granting rates are so low that none were observed for some MCS experiments.

V. Open Questions

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

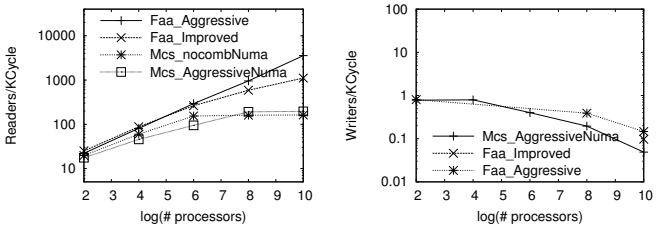

Fig. 15. Experiment R, $E_W = 0.1$

B. 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 [12] presents similar challenges for Internet Service Providers. In [15] 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.

C. 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.

VI. Conclusions

Having been surprised at the comparatively poor performance attained by the Ultracomputer's combining network when presented with 100% hot spot traffic that typifies busy wait polling of coordination variables, we identified two improvements to the combining switch design that increase its performance significantly. 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.

We then compared the performance of a centralized algorithm for the readers writers problem with that of the widely used MCS algorithm that reduces hot spots by busy wait spinning only on variables stored in the portion of shared memory that is co-located with the processor in question.

Our simulation studies of these two algorithms have yielded several results: First, the performance of the MCS algorithm is improved by the availability of combining. Second, when no readers are present, the MCS algorithm outperforms the centralized algorithm. Finally, when readers are present, the results are reversed. For many of these last experiments, an order of magnitude improvement is seen.

A switch capable of combining memory references is 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] 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.

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

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

[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] 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.

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

[9] 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.

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

[11] 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.

[12] "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".

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

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

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

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

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

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

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