A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols

Mark Heinrich
School of Electrical Engineering, Cornell University, Ithaca, NY 14853

Vijayaraghavan Soundararajan and John Hennessy
Computer Systems Laboratory, Stanford University, Stanford, CA 94305

Anoop Gupta
Microsoft Research, Redmond, WA 98052

***Abstract*—Scalable cache coherence protocols have become the key technology for creating moderate to large-scale shared-memory multiprocessors. Although the performance of such multiprocessors depends critically on the performance of the cache coherence protocol, little comparative performance data is available. Existing commercial implementations use a variety of different protocols including bit-vector/coarse-vector protocols, SCI-based protocols, and COMA protocols. Using the programmable protocol processor of the Stanford FLASH multiprocessor, we provide a detailed, implementation-oriented evaluation of four popular cache coherence protocols. In addition to measurements of the characteristics of protocol execution (e.g. memory overhead, protocol execution time, and message count) and of overall performance, we examine the effects of scaling the processor count from 1 to 128 processors. Surprisingly, the optimal protocol changes for different applications and can change with processor count even within the same application. These results help identify the strengths of specific protocols and illustrate the benefits of providing flexibility in the choice of cache coherence protocol.**

I. Introduction

In the late 1980s and early 1990s, the development of directory-based cache coherence protocols allowed the creation of cache-coherent distributed shared-memory (DSM) multiprocessors. These DSM multiprocessors are also called Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines, reflecting the disparity between access times to local and remote memories. The availability of cache coherence, and hence software compatibility with small-scale bus-based machines, popularized the commercial use of DSM machines for scalable multiprocessors.

A. Cache Coherence Protocol Design Space

Commercial CC-NUMA multiprocessors use variations on three major protocols: bit-vector/coarse-vector [10][17][28], SCI [4][9][19], and COMA [5]. In addition, a number of other protocols have been proposed for use in research machines [2][7][20] [23][29]. The research protocols have tended to focus on changing the bit-vector directory organization to scale more gracefully to larger numbers of processors. All existing scalable cache coherence protocols rely on the use of distributed directories [1], but beyond that the protocols vary widely in how they deal with scalability, as well as what techniques they use to reduce remote memory latency.

Cache coherence protocols can be evaluated on how well they deal with the following four issues:

Protocol memory efficiency: how much memory overhead does the protocol require? Memory usage is critical for scalability. We consider only protocols that have memory overhead that scales efficiently with the number of processors. To achieve this efficient scaling, some protocols use hybrid solutions (such as a coarse-vector extension of a standard bit-vector protocol), while others keep sharing information in non-bit-vector data structures to reduce memory overhead. The result is that significant differences in memory overhead can still exist in scalable coherence protocols.

Direct protocol overhead: how much overhead do basic protocol operations require? This often relates to how directory information is stored and updated, as well as attempts to reduce global message traffic. Direct protocol overhead is the execution time for individual protocol operations, measured by the number of clock cycles needed per operation. This research splits the direct protocol overhead into two parts: the latency overhead and the occupancy overhead. In DSM architectures, the node controller contributes to the latency of each message it handles. More subtly, after the controller sends the reply message it may continue with bookkeeping or state manipulations. This type of overhead is controller occupancy, and does not affect the latency of the current message, but it may affect the latency of subsequent messages because it determines the rate at which the node controller can handle messages. Keeping both latency and occupancy to a minimum is critical in high performance DSM machines [14].

Message efficiency: how well does the protocol perform as measured by the global traffic generated? Most protocol optimizations try to reduce message traffic, so this aspect is accounted for in message efficiency. The existing protocols vary widely in this dimension. For example, COMA tries to reduce global traffic and improve performance by migrating cache lines. Other protocols sacrifice message efficiency (e.g., coarse-vector) to achieve memory scalability while maintaining protocol simplicity. Still others add traffic in the form of replacement hints to maintain precise sharing information.

Protocol scalability: Protocol scalability depends on both minimizing message traffic and on avoiding contention. In the latter area, some protocols (such as SCI) have explicit features to reduce contention and hot-spotting in the memory system.

B. Evaluating the Cache Coherence Protocols

The tradeoffs among these coherence protocols are extremely complex. No existing protocol is able to optimize its behavior in all four of the areas outlined above. Instead, a protocol focuses on some aspects, usually at the expense of others. While these tradeoffs and their qualitative effects are important, the bottom line remains how well a given protocol performs in practice. Determining this requires careful accounting of the actual overhead encountered in implementing each protocol.

Perhaps the most difficult aspect of such an evaluation is performing a fair comparison of the protocol implementations. Because most DSM machines fix the coherence protocol in hardware, comparing different DSM protocols means comparing performance across different machines. This is problematic because differences in machine architecture, design technology, or other artifacts can obfuscate the protocol comparison [24]. Fortunately, the FLASH machine [16] being built at Stanford University provides a platform for such a study. FLASH uses a programmable protocol engine that allows the implementation of different protocols while using an identical main processor, cache, memory, and interconnect. This focuses the evaluation on the differences introduced by the protocols themselves. Nonetheless, such a study does involve the non-trivial task of implementing and tuning each cache coherence protocol.

This research provides an implementation-based, quantitative analysis of the performance, scalability, and robustness of four scalable cache coherence protocols running on top of a single architecture, the FLASH multiprocessor. The four coherence protocols examined are bit-vector/coarse-vector, dynamic pointer allocation, SCI, and COMA. Each protocol is a complete implementation that runs without modification both under simulation and on the real FLASH machine. This is critical in a comparative performance evaluation since each protocol is known to be correct and to handle all deadlock avoidance cases, some of which can be subtle and easily overlooked in a paper design high-level protocol implementation. Through this comparison, this research demonstrates the utility of a programmable node controller that allows flexibility in the choice of cache coherence protocol. The results of this study can also be used to guide the construction of future, more robust, scalable coherence protocols.

II. Cache Coherence Protocols

This section describes our implementation of the four different protocols. All of the protocols use a distributed directory: the information about different memory blocks is kept in different directories that are distributed with the memory modules. A memory block is the smallest cacheable unit of storage; each block may be cached in one or more processor's caches. The node containing the memory and directory information for a single memory block is called the *home*.

A. Bit-vector/Coarse-vector

The bit-vector protocol [6] is designed to be fast and efficient for small to medium-scale machines, and is the simplest of all the cache coherence protocols. For each cache line in main memory, the bit-vector protocol keeps a *directory entry* that maintains all of the necessary state information for that cache line. Most of the directory entry is devoted to a series of presence bits from which the bit-vector protocol derives its name. A presence bit is set if the corresponding node's cache currently contains a copy of the cache line, and cleared otherwise.

In systems with large numbers of processors, P, increasing the number of presence bits becomes prohibitive because the total directory memory scales as P^2, and the width of the directory entry becomes unwieldy from an implementation standpoint. To scale the bit-vector protocol to these larger machine sizes the bit-vector protocol can be converted into a coarse-vector protocol [10]. The *coarseness* of the protocol is defined as the number of nodes each presence bit represents. Our 64-bit directory entry contains 48 presence bits, so a 64-processor machine has a coarseness of two, and a 128-processor machine has a coarseness of four. For the coarse-vector protocol, a presence bit is set if any of the nodes represented by that bit are currently sharing the cache line. The coarse-vector protocol therefore keeps *imprecise sharing information*, compromising message efficiency for memory efficiency and continued scalability. With a 64-bit directory entry, the bit-vector/coarse-vector protocol maintains a 6.25% memory overhead at all machine sizes.

The simplicity of the bit-vector/coarse-vector protocol transitions is the main reason for its popularity. The Stanford DASH multiprocessor [18] and the HAL-S1 [28] both implement a bit-vector protocol. Although these machines implement bit-vector at the directory level, they both have a built-in coarseness of four, since each bit in the directory entry corresponds to a single node that is itself a 4-processor symmetric multiprocessor (SMP). In both these machines a snoopy protocol is used to maintain coherence within the cluster, and the bit-vector directory protocol maintains coherence between clusters. The more scalable SGI Origin 2000 [17] implements a bit-vector/coarse-vector protocol where the coarseness transitions immediately from one to eight above 128 processors.

B. Dynamic Pointer Allocation

The dynamic pointer allocation protocol [23] maintains precise sharing information up to very large machine sizes. Its directory entry maintains state bits similar to those kept by the bit-vector protocol, but instead of having a vector of presence bits, the directory entry serves only as a *directory header*, with additional sharing information maintained in a linked list structure. For efficiency, the directory header contains a local bit indicating the caching state of the local processor, as well as a field for the first sharer on the list. It also contains a pointer to the remaining list of sharers. The remainder of the sharing list is allocated from a static pool of data structures called the *pointer/link store* that contains a pointer to another sharer and a link to the next element in the sharing list. Initially the pointer/link store is linked together into a large free list.

When a processor reads a cache line, the controller removes a new pointer from the head of the free list and links it to the head of the linked list being maintained by that directory header. This is analogous to the setting of a presence bit in the bit-vector protocol. When a cache line is written, the controller traverses the linked list of sharers and sends invalidation messages to each sharer in the list. When it reaches the end of the list, the entire list is reclaimed and placed back on the free list.

Unfortunately, the dynamic pointer allocation protocol has an additional complexity. Because the pointer/link store is a fixed resource, it is possible to run out of pointers. To avoid the costly process of forcibly reclaiming pointers by selectively invalidating processor caches, the dynamic pointer allocation protocol makes use of *replacement hints*. In DSM machines, several cache coherence protocols can benefit by knowing when a line has been replaced from a processor's cache, even if the line was only in a shared state. Dynamic pointer allocation uses replacement hints to traverse the linked list of sharers and remove entries from the list. Replacement hints prevent an unnecessary invalidation and invalidation acknowledgment from being sent the next time the cache line is written, and return unneeded pointers to the free list where they can be re-used. However, replacement hints do have a cost in that they are an additional message type that has to be handled by the system.

The directory memory overhead of dynamic pointer allocation is the same as that for the bit-vector protocol, with additional memory required for the pointer/link store. Simoni [23] recommends the pointer/link store have a number of entries equal to eight to sixteen times the number of cache lines in the local processor cache. Assuming a processor cache size of 1MB, a pointer/link store multiple of sixteen, and 64MB of memory per node, the memory overhead of the dynamic pointer allocation protocol is 7.03%.

C. Scalable Coherent Interface

The Scalable Coherent Interface (SCI) protocol is also known as IEEE Standard 1596-1992 [22]. The goal of the SCI protocol is to scale gracefully to large numbers of nodes with minimal memory overhead. The main idea behind SCI is to keep a linked list of sharers, but unlike the dynamic pointer allocation protocol, this list is doubly-linked and distributed across the nodes of the machine. The directory entry for SCI is 1/4 the size of the directory entries for the two previous protocols because it contains only a pointer to the first node in the sharing list.

To traverse the sharing list, the protocol must follow the pointer in the directory header through the network until it arrives at the indicated processor. That processor must maintain a "duplicate set of tags" data structure that mimics the current state of its processor cache. The duplicate tags structure consists of a backward pointer, the current cache state, and a forward pointer to the next processor in the list. The official SCI specification implements this data structure directly in the secondary cache of the main processor, and thus SCI is sometimes referred to as a cache-based protocol. In practice, since the secondary cache is under tight control of the CPU and needs to remain small and fast for uniprocessor nodes, most SCI-based architectures implement this data structure as a duplicate set of cache tags in the main memory system of each node.

The distributed nature of the SCI protocol has two advantages: first, it reduces the memory overhead considerably because of the smaller directory headers and the fact that the duplicate tag information adds only a small amount of overhead per processor, proportional to the number of processor cache lines rather than the much larger number of local main memory cache lines; second, it reduces hot-spotting in the memory system. Assuming 64 MB of memory per node, a 1 MB processor cache, and a cache line size of 128 bytes, the memory overhead of the SCI protocol is 1.66%.

SCI can reduce hot-spotting compared to other protocols by changing the distribution of requests in the system. In the previous two protocols, unsuccessful attempts to retrieve a highly contended cache line repeatedly re-issue to the same home memory module. In SCI, the home node is asked only once, at which point the requesting node is made the head of the distributed sharing list. The requesting node retries by sending all subsequent requests to the old head of the list, rather than the home node. Many nodes in turn may be in the same situation, asking only their forward pointers for the data. Thus, the SCI protocol forms an orderly queue for the contended line, distributing the requests evenly throughout the machine. This even distribution of requests often results in lower application synchronization times.

The distributed nature does come at a cost though, as the state transitions of the SCI protocol are quite complex due to the non-atomicity of most protocol actions. Nonetheless, because it is an IEEE standard, has low memory overhead, and can potentially benefit from its distributed nature, various derivatives of the SCI protocol are used in several machines including the Sequent NUMA-Q [19] machine, the HP Exemplar [4], and the Data General Aviion [9].

D. Cache Only Memory Architecture

The Cache Only Memory Architecture (COMA) protocol is fundamentally different from the protocols discussed earlier. COMA treats main memory as a large cache, called an *attraction memory* (AM), and provides automatic migration and replication of main memory at a cache line granularity. COMA can potentially reduce the cost of processor cache misses by converting high-latency remote misses into low-latency local misses. The notion that the hardware can automatically bring needed data closer to the processor without advanced programmer information is the allure of the COMA protocol.

Our version of COMA is a flat COMA or COMA-F [26] protocol that assigns a static home for the directory entries of each cache line just as in the previous protocols. If the cache line is not in the local AM, the statically assigned home is immediately consulted to find out where the data resides. COMA-F removes the disadvantages of the hierarchical directory structure of the original COMA protocol and makes it possible to implement COMA on a traditional DSM architecture. For brevity we refer to our COMA-F protocol simply as COMA.

Unlike the other protocols, COMA needs extra "reserved" memory on each node to efficiently support cache line replication. Without reserved memory, COMA could only migrate data, since any new data placed in one AM would displace the last remaining copy of another cache line. In COMA, one copy of each cache line is designated the *master copy*, which is carefully tracked on displacements to prevent losing the last remaining copy of the line. By adding reserved memory, COMA can replicate data and need only take additional action if it is displacing a master copy. Extra reserved memory is crucial in keeping the number of AM displacements to a minimum. [15] shows that for many applications half of a direct-mapped AM should be reserved memory.

Our COMA protocol uses dynamic pointer allocation as its underlying directory organization. The only difference in the data structures is that COMA keeps additional tag and state fields in the directory header to identify which global cache line is currently in the AM. Our AM is direct-mapped for both simplicity and speed. Because COMA must perform a tag comparison of the cache miss address with the address in the AM, COMA can potentially have higher miss latencies than the previous protocols. If the line is in the local AM then ideally COMA will be a win since a potential slow remote miss has been converted into a fast local miss. If however, the tag check fails and the line is not present in the local AM, COMA has to go out and fetch the line as normal, but it has delayed the fetch of the remote line by the time it takes to perform the tag check.

Despite the complications of extra tag checks and master copy displacements, the hope is that COMA's ability to turn remote capacity or conflict misses into local misses will outweigh any of these potential disadvantages. Several machines implement variants of the COMA protocol including the Swedish Institute of Computer Science's Data Diffusion Machine [11], and the KSR1 [5] from Kendall Square Research.

III. Simulation Methodology

The Stanford FLASH multiprocessor [16] is an ideal experimental vehicle for studying the performance impact of cache coherence protocols. A FLASH node looks like a standard CC-NUMA node, with one exception—FLASH replaces the hard-wired node controller with a flexible, programmable engine called MAGIC. MAGIC contains an embedded *protocol processor* that runs software code sequences, or *handlers*, to implement the cache coherence protocol. By taking advantage of FLASH's flexibility, we can write handlers for each of our four cache coherence protocols and run them on the protocol processor. Thus, we can hold constant the other aspects of the FLASH architecture (i.e., processor, memory, and network characteristics) and change only the cache coherence protocol the machine is run-

TABLE I
READ LATENCIES OF CURRENT DSM MACHINES (ns)[A]

Machine	Protocol	Local Read	Remote Read: Clean at Home	Remote Read: Dirty Remote
DG NUMALiine	SCI	165	2400	3400
FLASH	Flexible	190	960	1445
HAL S1	BV	180	1005	1305
HP Exemplar	SCI	450	1315	1955
SGI Origin 2000	BV/CV	200	710	1055

A. Remote times assume the average number of network hops for 32 processors (except for HAL-S1 which only scales to 16 processors).

ning. The result is an implementation-oriented, unbiased evaluation of the cache coherence protocols in this study.

One of the FLASH design goals was to maintain the advantages of implementing coherence protocols in software, but operate at the speed of hardware cache-coherent machines. Table I shows read latencies in nanoseconds for FLASH and current commercially available DSM machines. The table shows three read times: a local cache read miss, a remote read miss where the data is supplied by the home node, and a remote read miss where the data must be supplied by a dirty third node. All times assume no contention and are measured from the time the cache miss first appears on the processor bus to the time the first word of the data reply appears on the processor bus. All data is supplied by the machine's designer via personal communication or publication [3][8][17][27].

The main point here is that despite running its protocols in "software", FLASH has read latencies comparable to (and often better than) commercially available hardware cache-coherent machines. The strong baseline performance of FLASH is an important component of this study. If FLASH were running in a realm where node controller bandwidth was consistently a severe bottleneck, then the performance of the cache coherence protocols would be determined almost entirely by their direct protocol overhead. In a more balanced machine like FLASH, direct protocol overhead is only one aspect of the protocol comparison, and other aspects of the comparison like message efficiency and protocol scalability features come into play.

A. Simulation Parameters

At the time of this writing, a four-processor FLASH machine is up and running, but to obtain performance and scalability results up to 128 processors we use execution-driven simulation for this study. The processor simulator is Mipsy, an emulation-based simulator that is part of the SimOS suite [21] and interfaces directly to FlashLite, the system-level simulator. Mipsy models the processor and its caches, while FlashLite models everything from the processor bus downward. FlashLite uses a lightweight threads package that accurately models the timing of the actual FLASH system hardware, and properly simulates contention at all interfaces. The protocol processor thread of FlashLite is itself an instruction set emulator that runs the compiled protocol code that runs on the real machine. To factor out the effect of protocol instruction cache misses, we simulate a perfect MAGIC instruction cache in this study, rather than the normal 16 KB MAGIC instruction cache. Other system parameters are taken directly from the FLASH machine [13].

In this study, Mipsy simulates a single-issue 300 MHz processor with blocking reads and non-blocking writes. The processor has split first-level instruction and data caches of 32 KB each and a combined 1 MB, 2-way set-associative secondary cache with 128 byte cache lines. Though the processor has blocking reads, it supports non-blocking prefetch operations, allowing us to use prefetched versions of our applications to simulate a more aggressive processor design. Moreover, all the protocols operate in a relaxed consistency mode that allows write data to be returned to the processor before all invalidation acknowledgments have been collected. The combination of prefetching and a relaxed consistency mode can elicit occupancy-induced protocol performance problems that might remain latent in lower-performance environments.

B. Applications

To properly assess the scalability and robustness of cache coherence protocols it is necessary to choose applications that scale well to large machine sizes. This currently limits us to the realm of scientific applications, but does not limit the applicability of our results. (See [25] for results at smaller machine sizes with multiprogramming and operating system workloads). Our applications are selected from the SPLASH-2 application suite [30]. In particular, this study examines FFT, Ocean, Radix-Sort, LU, Barnes-Hut, and Water. All applications except Barnes-Hut and Water use hand-inserted prefetches to reduce read miss penalty.

So that the applications achieve reasonable parallel performance, their problem sizes are chosen to achieve a target minimum *parallel efficiency* at 128 processors. Parallel efficiency is defined as speedup divided by the number of processors. An application's problem size is determined by choosing a target minimum parallel efficiency of 60% for the best version of the application running the best protocol at 128 processors. In addition, multiple versions of each application are examined, varying from highly-optimized to less-tuned versions. Most of the applications have two main optimizations that are selectively turned off: data placement, and an optimized communication phase. As another variation, the less-tuned versions of the applications are also run with smaller 64 KB processor caches. Since this cache size is smaller than the working sets of some of our applications, these configurations place different demands on the cache coherence protocols than the large-cache configurations, and lead to some surprising results.

IV. RESULTS

This section presents the results of our protocol comparisons. Although this section does not discuss every simulation result in the study, the results presented are representative of the entire set and show the range of performance observed for each of the cache coherence protocols. The full set of results can be found in [12].

A. Direct Protocol Overhead

To help understand the application performance results, it is first useful to examine what happens on a cache read miss under each protocol. Figure 1 shows the protocol processor latencies and occupancies for two common read miss cases: a local read miss, and a remote read miss satisfied by the home node. The remote read miss is separated into the portion of the request handled at the requester on the way to the home, the portion handled by the home itself, and the reply handled back at the requester. Note also that the latency in Figure 1 is not the overall end-to-end miss latency, but rather just the handler component (path length) of that part of the miss.

Figure 1 shows that the latency for the local read miss case is about the same in all the protocols, as are the latencies incurred at the home for the remote read case and at the requester on the read reply.

The real latency difference appears in the portion of the remote read miss incurred at the requester. The bit-vector and dynamic pointer allocation protocols do not keep any local state for remote lines so they simply forward the remote read miss into the network with a latency of 3 protocol processor cycles. COMA and SCI, however, do keep local state on remote lines, and consulting this state results in a significant extra latency penalty. Although COMA and SCI incur larger latencies at the requester on remote read misses, this is the cost of trying to gain an advantage at another level—COMA tries to convert remote misses into local misses and SCI tries to keep a low memory overhead and reduce contention by distributing its directory state.

The latency differences between the protocols are small compared to the occupancy differences shown on the right-hand side of Figure 1. In the local read case, bit-vector, COMA, and dynamic pointer allocation have only marginally larger occupancies than their corresponding latencies. But SCI incurs almost five times the occupancy of the other protocols on a local read miss. For the remote read case at the requester SCI again suffers a huge occupancy penalty. In addition, both SCI and COMA have large occupancies at the requester on the read reply, although in this case COMA has the highest controller occupancy. The reasons behind the higher occupancies of SCI and COMA at the requester are discussed in turn, below.

SCI's high occupancy at the requester is due to its cache replacement algorithm. SCI does not use replacement hints, but instead maintains a set of duplicate cache tags. On every cache miss, whether local or remote, SCI must roll out the block that is being displaced from the cache after first handling the current miss. The details of SCI roll out are discussed in [12] and account for the large occupancies incurred at the requester in the first two cases in Figure 1. The 23 cycle occupancy at the requester on the reply stems from the way SCI maintains its distributed linked list of sharing information. After the data is returned to the processor, the requesting node must notify the old head of the sharing list that it is no longer the head. The process of looking up the duplicate tag information to check the cache state, and then sending the change-of-head message accounts for the additional occupancy on the read reply.

COMA incurs 10 cycles of occupancy above and beyond its latency for the portion of a remote read miss handled at the requesting node. Besides the normal update of its AM data structures, COMA has to deal with the case of a conflict between the direct-mapped AM and the 2-way set associative processor secondary cache, adding some additional overhead to the handler. The largest controller occupancy for COMA, however, is incurred at the requester on the read reply. COMA immediately sends the data to the processor cache, incurring only one cycle of latency, but then it must check to see if any AM replacements need to be performed, and if so, send off those messages. Because this is the case of a reply generating additional requests, careful resource checks have to be made to avoid deadlock. Once the AM replacement is sent, the handler must then write the current data reply into the proper spot in the AM. Although this particular case incurs high occupancy in COMA, the good news is that it is not incurred at the home, and it occurs on a reply that finishes a transaction, rather than a request which may be retried many times, incurring large occupancy each time.

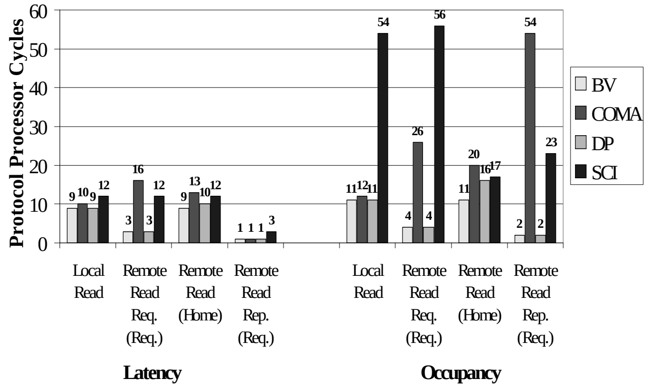

Fig. 1. FLASH protocol latency and occupancy comparison. *Latency* is the handler path length to the first `send` instruction, and *occupancy* is the total handler path length.

B. Message Overhead

While the direct protocol overhead described above is handler-specific, protocol message overhead is application-specific. In particular, message overhead is strongly dependent on application sharing patterns, and specifically on the number of readers of a cache line in between writes to that line. However, examining the average message overhead across all the applications yields a few interesting points.

In uniprocessor systems, both COMA and dynamic pointer allocation send 1.3 times the number of messages of the other protocols. This extra overhead is caused by the replacement hints used to keep precise sharing information in those protocols. However, precise sharing information begins to reap benefits at 64 processors when the bit-vector protocol transitions to a coarse-vector protocol. At 64 and 128 processors, coarse-vector sends 1.03 and 1.47 times more messages than dynamic pointer allocation, respectively.

COMA and SCI maintain about a 1.3 times average message overhead over bit-vector/coarse-vector until the machine size reaches 128 processors. One of the main goals of the COMA protocol is to reduce the number of remote read misses and therefore message count. The fact that COMA's message overhead remains higher than bit-vector/coarse-vector for all but the largest machine sizes foreshadows somewhat the COMA application results. For scalable performance, SCI is willing to tradeoff message efficiency for scalability and improved memory efficiency.

C. Application Performance

Most of the graphs in this section show normalized execution time versus the number of processors, with the processor count varying from 1 to 128. For each processor count, the application execution time under each of the four cache coherence protocols is normalized to the execution time for the bit-vector/coarse-vector protocol for that processor count. In other words, the bit-vector/coarse-vector bars always have a height of 1.0, and shorter bars indicate better performance.

C.1 FFT

Figure 2 shows the results for prefetched FFT. The results indicate that the choice of cache coherence protocol has a significant impact on performance. For machine sizes up to 32 processors both the bit-vector and dynamic pointer allocation protocols achieve perfect speedup. Their small read latencies and occupancies are too much to overcome for the SCI and COMA protocols, both of which are hurt by their higher latencies at the requester on remote read misses, their larger protocol processor occupancies, and their increased message overhead. The relative performance of both SCI and COMA

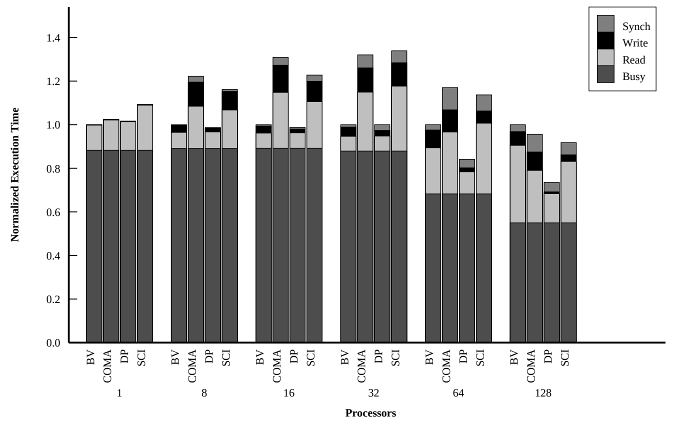

Fig. 2. Results for prefetched FFT.

decreases as the machine scales from 1 to 32 processors because the amount of contention in the system increases and these higher occupancy protocols are not able to compensate.

Surprisingly, the optimal protocol for prefetched FFT changes with machine size. For machine sizes up to 32 processors, bit-vector is the best protocol, followed closely by dynamic pointer allocation. But at 64 processors and above, where the bit-vector protocol turns coarse, the relative execution times of the other protocols begin to decrease to the point where dynamic pointer allocation is 1.36 times faster, SCI is 1.09 times faster, and COMA is 1.05 times faster than coarse-vector at 128 processors. While the bit-vector protocol sends the fewest messages for machine sizes between 1 and 32 processors, it sends the most messages for any machine size larger than 32 processors. At 64 processors coarse-vector sends 1.4 times more messages than dynamic pointer allocation, and at 128 processors coarse-vector sends 2.8 times more messages than dynamic pointer allocation. Even though the bit-vector/coarse-vector protocol handles each individual message efficiently (with low direct protocol overhead), at large machine sizes there are now simply too many messages to handle, and performance degrades relative to the other protocols that are maintaining precise sharing information.

To examine the effect of protocol performance on less-tuned applications, we show the results for prefetched FFT without explicit data placement directives in Figure 3. Qualitatively, for machine sizes up to 64 processors, the results for FFT without data placement are similar to the optimized results in Figure 2. The performance of SCI and COMA relative to bit-vector is slightly worse than in optimized FFT. The lack of careful data placement results in fewer local writes and more handlers per miss, factors that punish the protocols with higher direct protocol overhead.

Although the results for machine sizes up to 64 processors are similar, the results at 128 processors are drastically different. Most importantly, there is now over 2.5 times difference between the performance of the best and the worst cache coherence protocol. At 128 processors, coarse-vector is now considerably worse than the three other protocols—dynamic pointer allocation is 2.56 times faster, SCI is 2.34 times faster, and COMA is 1.83 times faster. The root of the performance problem is once again increased message overhead, as coarse-vector sends over 2.3 times as many messages as dynamic pointer allocation. Without data placement this message overhead is causing more performance problems because the extra messages are contributing to more hot-spotting at the node controller.

For all machine sizes, even without data placement, COMA does not perform as well as expected. Given COMA's ability to migrate data at the hardware level without programmer intervention, conventional wisdom would argue that COMA should perform relatively better without data placement than with data placement. Unfortunately, for the previous version of FFT without data placement, COMA performs worse than it does for the most optimized version of FFT with explicit data placement, despite higher AM hit rates. Since large processor caches seem to mitigate any potential COMA performance advantage, Figure 4 shows the results for a version of FFT without data placement with a processor secondary cache size of 64 KB. With smaller caches there are far more conflict and capacity misses, and COMA is expected to thrive.

Surprisingly, COMA's performance is much worse than expected despite AM hit rates for remote reads around 70% at small machine sizes, and 45% at the largest machine sizes. At 64 and 128 processors the coarse-vector protocol is 2.39 times and 2.34 times faster than COMA, respectively. But note that the coarse-vector protocol is also over 2.64 times faster than dynamic pointer allocation. Even though COMA is expected to perform well with small caches, the same small caches give rise to a large number of replacement hints. Replacement hints invoke high-occupancy handlers that walk the linked list of sharers to remove nodes from the list. The combination of large numbers of replacement hints and high-occupancy handlers leads to hot-spotting at the home node.

At 128 processors, SCI is the fastest protocol because it is least susceptible to hot-spotting, running 1.22 times faster than coarse-vector despite its higher direct protocol overhead. Once again,

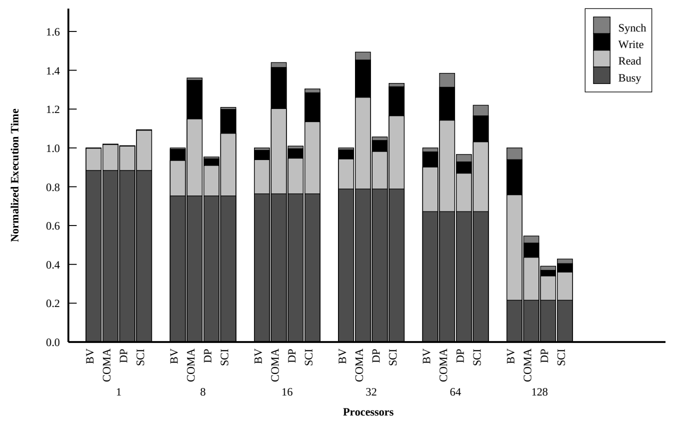

Fig. 3. Results for prefetched FFT with no data placement.

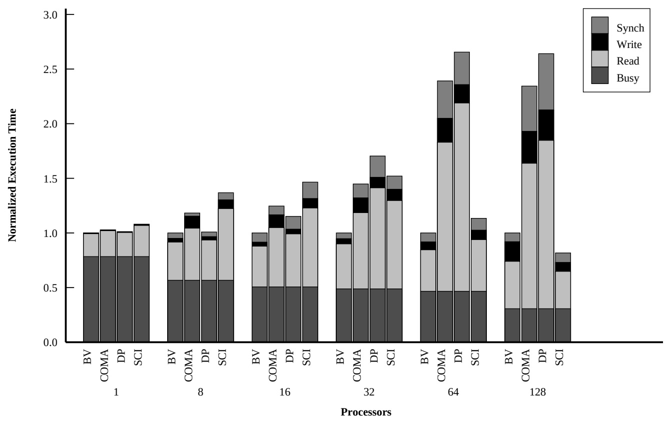

Fig. 4. Results for prefetched FFT with no data placement, an unstaggered transpose phase, and 64 KB processor caches.

coarse-vector is penalized by increased message overhead, sending 1.52 times as many messages as SCI. Interestingly, the SCI and dynamic pointer allocation protocols send the same number of messages, clearly demonstrating that message overhead is not the final word on performance since SCI performs over 3.2 times faster.

C.2 Ocean

Figure 5 shows the protocol performance for prefetched Ocean. Again, for machine sizes up to 32 processors the bit-vector and dynamic pointer allocation protocols perform about the same, but the higher overhead SCI and COMA protocols lag behind. At 32 processors, the bit-vector protocol is 1.25 times faster than SCI and 1.22 times faster than COMA. COMA's AM hit rates are higher than for optimized FFT (at about 10%) but still not high enough to overcome its larger remote read latency.

At large machine sizes the overhead of COMA and SCI both increase sharply. At 128 processors, dynamic pointer allocation is 1.22 times faster than coarse-vector, 1.78 times faster than COMA, and 2.06 times faster than SCI. In this optimized version of Ocean the performance problem at large processor counts is message overhead for SCI and a combination of low AM hit rate and protocol overhead-induced hot-spotting for COMA. SCI sends 2.9 times the number of messages as dynamic pointer allocation, and more surprisingly, 1.6 times more messages than the coarse-vector protocol with its imprecise sharing information.

Since COMA's AM hit rate is already high with large processor caches, we expected smaller processor caches to improve COMA's relative performance by increasing both capacity and conflict misses. Figure 6 shows the results for such a run with a 64 KB secondary cache. At 8 processors COMA is now indeed the best protocol—1.23 times faster than the bit-vector protocol, 1.31 times faster than dynamic pointer allocation, and 1.87 times faster than SCI. The AM hit ratio for remote reads is an impressive 89%. COMA successfully reduces the read stall time component of execution time, and thereby improves performance. But even though the AM hit ratio remains high at 85% for 16 processors and 84% for 32 processors, COMA's overhead begins to increase with respect to the bit-vector protocol, because as in FFT, with smaller caches come replacement hints and with larger machine sizes comes occupancy-induced hot-spotting at the node controller. Nonetheless, COMA remains the second-best protocol as the machine size scales. Dynamic pointer allocation and SCI are also suffering from increased replacement traffic, but COMA is still reducing the read stall time component while the other protocols have no inherent mechanisms to do so.

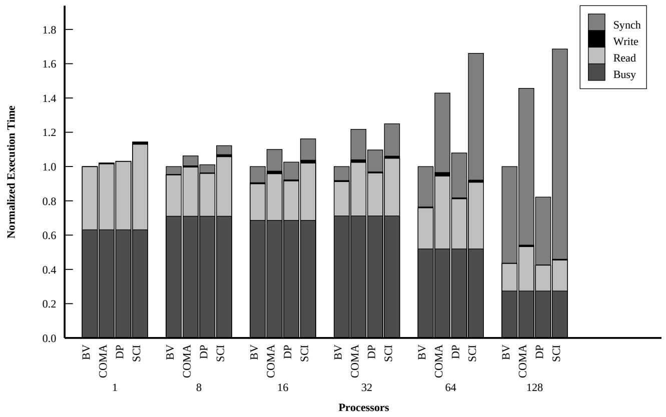

Fig. 5. Results for prefetched Ocean.

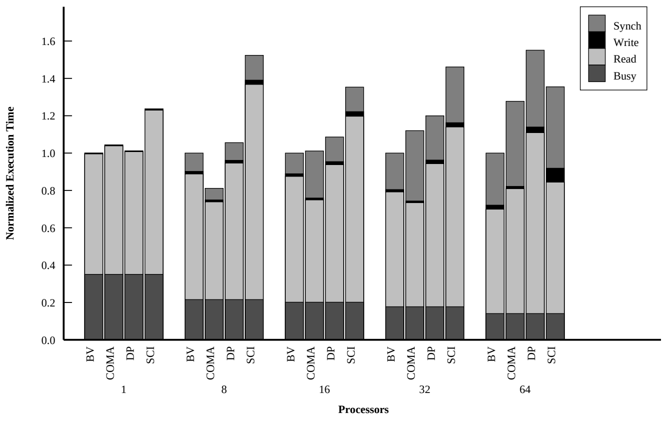

Fig. 6. Results for prefetched Ocean with no data placement and 64 KB processor caches.

C.3 Radix-Sort

Radix-Sort is fundamentally different from the other applications in this study because remote communication is done through writes. The other applications are optimized so that write traffic is local and all communication takes place via remote reads. In Radix-Sort, each processor distributes the keys by writing them to their final destination, causing not only remote write traffic, but highly-unstructured, non-uniform remote write traffic as well. Consequently, the relative performance of the cache coherence protocols for Radix-Sort depends more on their write performance than their read performance.

The results for Radix-Sort are shown in Figure 7. The poor performance of COMA immediately stands out from Figure 7. At 32 processors the bit-vector protocol is 1.78 times faster than COMA, and at 64 processors the coarse-vector protocol is 2.14 faster than COMA. Even though the use of a relaxed consistency model eliminates the direct dependence of write latency on overall performance, the effect of writes on both message traffic and protocol processor occupancy is still present, and in COMA is the fundamental reason for its performance being the poorest of all the protocols for Radix-Sort.

There are two main reasons for increased write overhead in the COMA protocol. First, only the master copy may provide data on a write request. This simplifies the protocol, but it means that on a write to shared data the home cannot satisfy the write miss as it can in the other protocols, unless the home also happens to be the master.

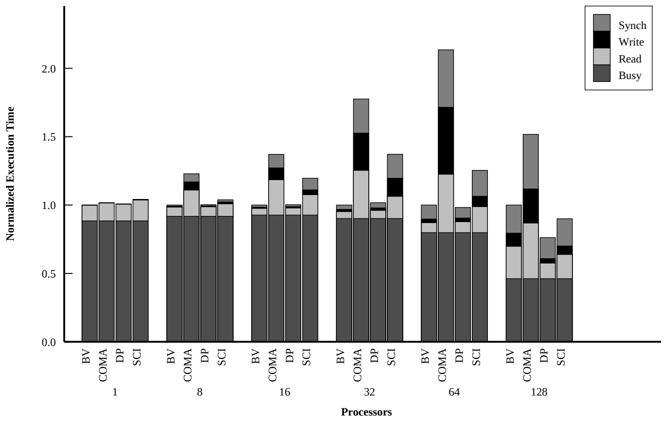

Fig. 7. Results for prefetched Radix-Sort.

Second, Radix-Sort generates considerable writeback traffic because of its random write pattern. This also results in a large number of dirty displacements from COMA's attraction memory, which unlike the writebacks in the other protocols, require an acknowledgment so that the master ownership may be tracked. Both the additional hop on writes and the additional acknowledgments increase COMA's message overhead with respect to the other protocols. At 32 processors COMA sends 1.66 more messages than the dynamic pointer allocation protocol, and at 64 processors that number jumps to 2.05 times the number of messages.

At 64 processors, Radix-Sort is performing well under all protocols except COMA. But at 128 processors, the higher message overhead and the write occupancies of the coarse-vector protocol degrade its performance considerably. For the COMA and coarse-vector protocols the speedup of Radix-Sort does not improve as the machine size scales from 64 to 128 processors. But under dynamic pointer allocation and SCI, Radix-Sort continues to scale, achieving a parallel efficiency of 52% at 128 processors under dynamic pointer allocation. Dynamic pointer allocation is 1.32 times faster than the coarse-vector protocol at 128 processors, and SCI is 1.11 times faster.

C.4 LU

The most optimized version of blocked, dense LU factorization spends very little of its time in the memory system, especially when the code includes prefetch operations. For this reason, the choice of cache coherence protocol makes little difference for optimized, prefetched LU, and we focus instead on other LU variations.

The version of LU shown in Figure 8 does not have data placement and uses full barriers between phases of the computation. Unlike the optimized LU, there are significant differences in protocol performance at 128 processors. This application is a dramatic example of how SCI's inherent distributed queuing of requests can improve access to highly contended cache lines and therefore improve overall performance. As Figure 8 shows, at 128 processors, synchronization time is dominating this version of LU, and the lack of data placement results in severe hot-spotting on the nodes containing highly-contended synchronization variables. The protocol processor utilizations shown in Table II show the effect of the SCI protocol in the face of severe application hot-spotting behavior. While SCI has a much higher average protocol processor utilization, the maximum utilization on any node is drastically smaller, and the variance between the two is by far the lowest of any of the protocols. The result is that despite having the largest message overhead, SCI has the least synchronization stall time and is the best protocol at large machine sizes—2.25 times faster than the coarse-vector protocol at 128 processors.

TABLE II
SCI'S AVERSION TO HOT-SPOTTING AT 128 PROCESSORS

Protocol	Average PP Utilization	Maximum PP Utilization
bit-vector/coarse-vector	1.7%	85.1%
COMA	4.9%	69.5%
dynamic pointer allocation	2.2%	60.9%
SCI	22.0%	32.2%

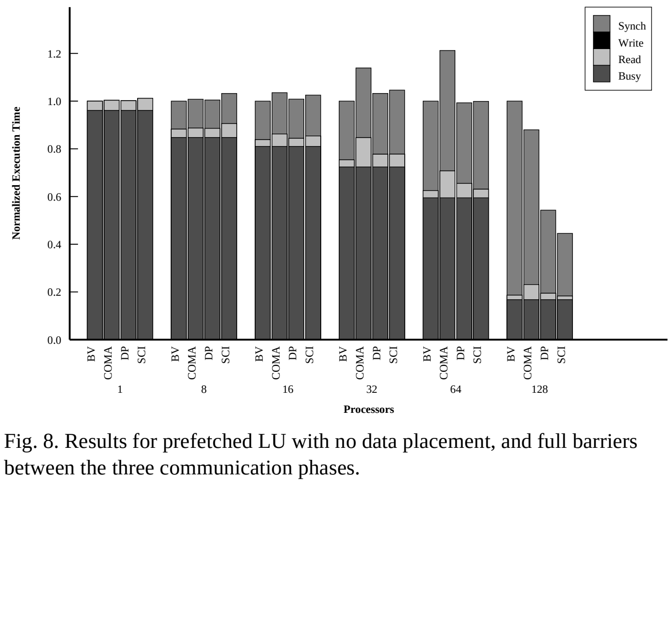

Fig. 8. Results for prefetched LU with no data placement, and full barriers between the three communication phases.

The results for the same version of LU from the previous section, but with smaller 64 KB processor caches are shown in Figure 9. Like the other small cache configurations, dynamic pointer allocation and COMA suffer the overhead of an increased number of replacement hints. Replacement hints exacerbate the hot-spotting present in an application since they on average return more often to the node controller which is being most heavily utilized. The SCI results are again the most interesting. For all but the largest machine size, bit-vector is about 1.2 times faster than SCI. Again, at small cache sizes SCI's distributed replacement scheme has both high direct protocol overhead and large message overhead. SCI's message overhead is consistently 1.4 times that of bit-vector/coarse-vector at all machine sizes. But at 128 processors, despite its message overhead, SCI is by far the best protocol (over 1.6 times faster than the others) because of its inherent resistance to hot-spotting.

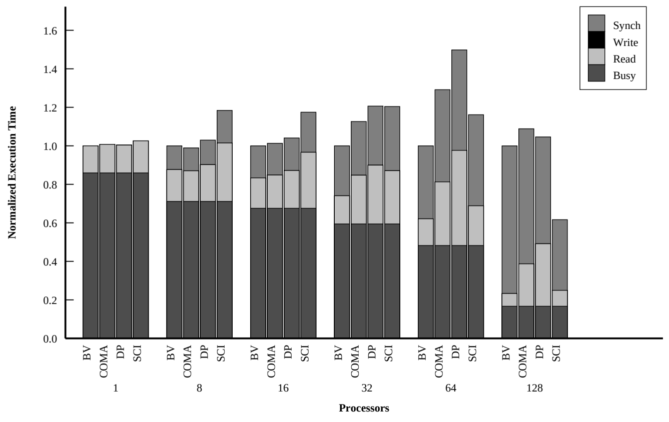

Fig. 9. Results for prefetched LU with no data placement, full barriers between the three communication phases, and 64 KB processor caches.

C.5 Barnes-Hut and Water

The results for Barnes-Hut are shown in Figure 10. The performance results for Water are similar to Barnes-Hut, and are not shown. Full details can be found in [12]. All the protocols perform quite well below 64 processor machine sizes, achieving over 92% parallel efficiency in all cases, with the exception of COMA's 82% parallel efficiency at 32 processors. The only sizable performance difference for these small machine sizes is at 32 processors where dynamic pointer allocation is 1.14 times faster than COMA. In this case, COMA is adversely affected by hot-spotting at one of the node controllers. While the average protocol processor utilization is 8.4% for COMA at 32 processors, the most heavily used protocol processor has a utilization of 42.3%. A significant fraction of the read misses (37%) in Barnes-Hut are "3-hop" dirty remote misses—a case

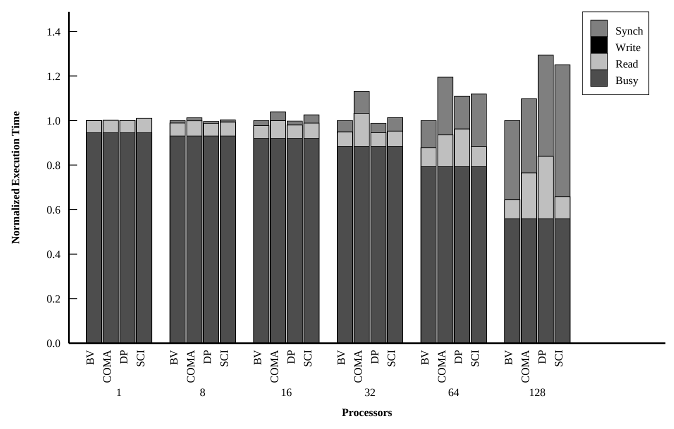

Fig. 10. Results for Barnes-Hut.

where COMA has a higher direct protocol overhead than the other protocols—and the AM hit rate of 30% is not enough to balance out this overhead increase.

At larger machine sizes, load imbalance becomes the bottleneck in Barnes-Hut, and application synchronization stall times dominate the total stall time. The bit-vector/coarse-vector is by far the best protocol at both 64 and 128 processors. Unlike the previous applications, Barnes-Hut has many cache lines that are shared amongst all of the processors. Long sharing lists help the bit-vector/coarse-vector protocol because there is a larger chance that it will not be sending unnecessary invalidations on write misses. Long sharing lists also hurt dynamic pointer allocation and COMA, because replacement hints have to traverse a long linked list to remove a node from the sharing list, resulting in a high occupancy protocol handler. This can degrade performance by creating a hot-spot at the home node for the replaced block. SCI is indirectly hurt by long sharing lists for two reasons: invalidating long lists is slower on SCI than the other protocols due to its serial invalidation scheme, and cache replacements from the middle of an SCI sharing list have higher overhead than a replacement from a sharing list with two or fewer sharers.

V. Conclusions

This implementation-based, quantitative comparison of four scalable cache coherence protocols has shown that none of the protocols in this study always perform best—in fact, there are cases where each protocol performs best, and where each protocol performs worst. The results demonstrate that protocols with small latency differences can still have large overall performance differences because controller occupancy is a key to robust performance in CC-NUMA machines.

Several themes have emerged to help determine which protocol may perform best given certain application characteristics and machine configurations. First, the bit-vector protocol is difficult to beat at small-to-medium scale machines before it turns coarse. Second, with small processor caches both COMA and dynamic pointer allocation perform poorly because of occupancy-induced contention caused by replacement hints. Third, although the other three protocols incur less protocol processor occupancy than the SCI protocol, SCI incurs occupancy at the requester rather than the home, making it less susceptible to hot-spotting and therefore more robust for less-tuned applications. Fourth, for applications with a small, fixed number of sharers running on machines with large processor caches, the dynamic pointer allocation protocol performs well at all machine sizes, and is the best protocol at the largest machine sizes. Fifth, the COMA protocol can achieve very high AM hit rates on applications that do not perform data placement, but its higher remote read miss latencies and protocol processor occupancies often remain too large to overcome. Finally, increased message overhead is often the root of the performance difference between the protocols at large machine sizes, but when hot-spotting or high-occupancy handlers are present, these effects dominate instead.

Surprisingly, this study finds that the optimal protocol changes as the machine size scales—even within the same application. In addition, changing architectural aspects other than machine size (like cache size) can change the optimal coherence protocol. Both of these findings are of particular interest to commercial industry, where today the choice of cache coherence protocol is made at design time and is fixed by the hardware. These results argue for programmable protocols on scalable machines, or the creation of a new, more robust cache coherence protocol.

Acknowledgment

This research is supported by DARPA contract DABT63-94C-0054. The authors wish to thank the FLASH team members as well as Robert Bosch for his tireless support of the simulation environment.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280-289, June 1988.

[2] J. K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. Ph.D. Dissertation, Department of Computer Science, University of Washington, February 1987.

[3] T. Brewer. Personal Communication, February 1998.

[4] T. Brewer and G. Astfalk. The evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring'97: Forty Second IEEE Computer Society International Conference*, pages 81-86, February 1997.

[5] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR-1 Computer System. Tech. Rep KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[6] L. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112-1118, December 1978.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224-234, April 1991.

[8] R. Clark. Personal Communication, February 1998.

[9] Data General Corporation. Aviion AV 20000 Server Technical Overview. Data General White Paper, 1997.

[10] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I.312-I.321, August 1990.

[11] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *IEEE Computer*, pages 44-54. September 1992.

[12] M. Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. Ph.D. Dissertation, Stanford University, Stanford, CA, October 1998.

[13] M. Heinrich, J. Kuskin, D. Ofelt, et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274-285, October 1994.

[14] C. Holt, M. Heinrich, J. P. Singh, et al. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[15] T. Joe and J. L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 82-93, April 1994.

[16] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[17] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241-251, June 1997.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63-79, March 1992.

[19] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308-317, May 1996.

[20] S. Reinhardt, J. Larus, D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-336, April 1994.

[21] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34-43, Winter 1995.

[22] Scalable Coherent Interface, ANSI/IEEE Standard 1596-1992, August 1993.

[23] R. Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, October 1992.

[24] J. P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Proceedings of Supercomputing ‘93*, pages 214-225, November 1993.

[25] V. Soundararajan, M. Heinrich, B. Verghese, et al. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 342-355, July 1998.

[26] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80-91. May 1992.

[27] W.-D. Weber. Personal Communication, February 1998.

[28] W.-D. Weber, S. Gold, P. Helland, et al. The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 98-107, June 1997.

[29] W.-D. Weber. *Scalable Directories for Cache-Coherent Shared Memory Multiprocessors*. Ph.D. Dissertation, Stanford University, Stanford, CA, January 1993.

[30] S. C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, June 1995.