Exploring the Value of Supporting Multiple DSM Protocols in Hardware DSM Controllers

Ravindra Kuramkote, John Carter

UUCS-99-024

Computer Systems Laboratory
University of Utah

The performance of a hardware distributed shared memory (DSM) system is largely dependent on its architect's ability to reduce the number of remote memory misses that occur. Previous attempts to solve this problem have included measures such as supporting both the CC-NUMA and S-COMA architectures in the same machine and providing a programmable DSM controller that can emulate any DSM mechanism. In this paper we first present the design of a DSM controller that supports multiple DSM protocols in custom hardware, and allows the programmer or compiler to specify on a per-variable basis what protocol to use to keep that variable coherent. The simulated performance of this DSM controller compares favorably with that of conventional single-protocol custom hardware designs, often outperforming the conventional systems by a factor of two. To achieve these promising results, the multi-protocol DSM controller needed to support only two DSM architectures (CC-NUMA and S-COMA) and three coherency protocols (both release and sequentially consistent write invalidate and release consistent write update). This work demonstrates the value of supporting a degree of flexibility in one's DSM controller design and suggests what operations such a flexible DSM controller should support.

Exploring the Value of Supporting Multiple DSM Protocols in Hardware DSM Controllers

Ravindra Kuramkote, John Carter

Computer Systems Laboratory
University of Utah

1 Introduction

Hardware distributed shared memory (DSM) systems are primarily targeted for parallel applications with fine-grained communication requirements. Minimizing the performance penalty induced by remote memory misses is a major design goal for all such systems. Remote memory misses fall into two classes: *essential* and *non-essential* [4]. Hence, to maximize performance, DSM architects can either incorporate features that reduce the number of non-essential misses or employ a high performance network so that the latency of a remote miss is close to that of a local memory access.

The DSM controller in most commercial hardware DSM systems supports a single architecture and protocol, most often cache coherent NUMA (CC-NUMA) with a sequentially consistent write invalidate protocol [16, 19, 29]. To improve performance, designers of such systems employ fast and expensive networks [16], thereby dropping the latency of a remote miss to close to that of a local miss, or add memory to the DSM controller to cache remote data [19], thereby turning remote misses into local misses. Because these machines support only a single DSM protocol, programmers sometimes must resort to data restructuring or software page migration [5, 13, 28] to eliminate remote misses when their impact on performance is too high.

Non-essential misses can be divided into two categories: (i) misses caused by inefficient use of the memory hierarchy on each node and (ii) misses caused by the use of sub-optimal coherency protocols for consistency management. Researchers have proposed a number of techniques for attacking each of these categories of non-essential misses, including COMA-style architectures that cache remote data in unused main memory [26] and programmable DSM controllers that can support application-specific coherency protocols [15, 25].

In this paper we first present the design of a DSM controller that supports multiple protocols in custom hardware. In this design, the programmer or compiler can specify on a per-variable basis what DSM mechanisms should be used to maintain the consistency of the variable (e.g., CC-NUMA employing a sequentially consistent write invalidate protocol or S-COMA employing a release consistent write update protocol). We believe that such a controller provides an interesting middle ground between a single protocol custom state machine and a fully programmable controller, such as is present in FLASH [15] and Typhoon [25].

Several researchers, both in industry and academia, have proposed systems that can support relaxed consistency models or multiple protocols. Our research differs from these in several ways. First, we assume a generic system bus with no special support for the protocols [12]. Second, we do not assume the ability to modify any of the processor cache hierarchy [7, 22]. Finally, we assume that data will be mapped to protocols statically, not dynamically [14].

To evaluate the potential benefits of a multi-protocol DSM controller, we first model and simulate an *optimal* DSM system. In this *optimal* model, all conflict misses to remote memory are satisfied from the local memory by modeling a perfect S-COMA architecture and all coherency misses are eliminated by modeling a multiple-writer protocol that sends updates to remote nodes via a zero-latency infinite bandwidth network. All other sources of system overhead are modeled, including the time required to perform the application's instructions, local memory accesses, system calls, and synchronization. This optimal model gives us a best case for performance against which we can compare the various implementation alternatives.

We then measure the performance of a set of programs from the Splash-2 and Wisconsin Wind Tunnel benchmark suites on the optimal DSM system, a conventional custom single-protocol DSM controller, our custom multi-protocol DSM controller design, and a programmable DSM controller. We find that architectures employing a single write-invalidate CC-NUMA protocol perform up to a factor of three worse than the optimal system. We further show that by employing an appropriate combination of local memory caching, release consistency, and a write update protocol for some of the data structures in an application, application runtimes can be reduced by up to 60% and network bandwidth consumption can be reduced by up to 95% compared to an architecture with just the write-invalidate protocol.

Even when the programmer *wants* to use the protocol provided by the single-protocol design, we show that the performance of a multi-protocol custom controller (*MPCC*) is almost equivalent to that of the single-protocol custom controller (*SPCC*). The performance of a DSM system that employs a fully programmable protocol processor (*PPC*) performs roughly 20% worse than the SPCC machine when it implements the base coherence protocol. However, when the PPC employs multiple protocols, it performs 30-80% better than an SPCC system on some applications. As a result of these experiments, we believe that DSM hardware should support the ability for programmers or compilers to specify how particular variables should be kept coherent.

This paper makes the following contributions:

- We characterize the performance of an *optimal* DSM controller on a large number of applications and show that conventional DSM controller designs have considerable room for improvement compared to such an optimal system.

- We provide the design of a custom hardware DSM controller that supports multiple architectures (release consistent and sequentially consistent CC-NUMA and S-COMA) and coherence protocols (write invalidate and write update). This design allows different applications to use different protocols, and even allows a single application to use different protocols to manage different data.

- We introduce a novel mechanism called the *release state table* that implements an efficient release consistent write update protocol in hardware.

- We provide the first evaluation of a custom hardware DSM controller that supports multiple DSM architectures and protocols.

- We evaluate the overheads associated with employing a programmable DSM controller to support multiple protocols and find that the overhead induced by the higher occupancy and latency of such systems is high.

However, we also find that the ability to support multiple protocols and architectures can more than make up for the increased overhead on some applications.

The remainder of this paper is organized as follows. We present background material on DSM controller design alternatives and other related work in Section 2. In Section 3 we describe the design of the multiple-protocol custom coherence controller. We describe our simulation environment, test applications, and experiments in Section 4. We present the results of our detailed simulation experiments in Section 5. Finally, we draw conclusions and discuss possible future work in Section 6.

2 Background

The MIT Alewife machine [3] was the first hardware-based DSM system to use software for protocol processing. However, Alewife limited its use to extending directory pointers and providing a limited number of specialized synchronization operations.

The Stanford DASH multiprocessor [18] was designed with custom state logic. It supported a CC-NUMA memory architecture that employed both sequential and release consistent write invalidate protocols. It did not, however, provide a way for an application to control the consistency model used to maintain the consistency of its data.

The Stanford FLASH [8] and Wisconsin Typhoon [25] systems use special-purpose processors to implement their DSM mechanisms. Both of these systems in theory allow users to develop their own coherence protocols that are perfectly suited for how their application accesses data. For example, FLASH's programmability has been used to implement a scalable directory scheme and a flat-COMA memory model. However, no study has been done to show the benefits of using FLASH's flexibility to support multiple coherence protocols.

Tempest/Typhoon [25] supports a memory architecture similar to SCOMA called *stache* and a hybrid memory CC-NUMA/S-COMA architecture called *Reactive NUMA* [6]. Another study by the same group showed how the flexibility of Tempest can be used to support coherency protocols specific to an application [5], but this work relied on the programmer or compiler restructuring application data and did not compare performance against a custom controller.

Unfortunately, recent studies have shown that the high occupancy of a programmable controller can result in a performance penalty of 4% to 93% in an SMP node compared to a custom controller [10, 21]. In a generic workstation environment, this can impact the performance of applications that do not use shared memory. This brings into question whether the benefits of this degree of flexibility are outweighed by the higher overheads caused by using a general purpose processor as opposed to a custom solution.

The SHRIMP Multicomputer [1] supports a write-update protocol to efficiently write directly to remote nodes. To implement this protocol, SHRIMP relies on the processor caches being write though, which is not the case on all commodity workstations. Similarly, the competitive update protocols of Grahn et al. [7] rely on the processor caches being write-through or the memory controller being able to detect cache state transitions in the second level cache. These systems also do not provide a way for the user to control the protocol implemented by the hardware.

The S3.mp multiprocessor system [23] contains programmable micro-controllers to implement DSM. This programmable feature is used to support both the CC-NUMA and SCOMA memory models. However, it does not provide a way for the user to select the model – this selection is performed when the machine is booted.

There have been several systems that support more than one protocol [14, 27]. All these systems use information collected during runtime to determine the protocol to use for individual cache lines. These systems, in addition to incurring run-time overhead, require all programs to followed the consistency model that the system adapts to.

3 Design of a Multi-Protocol DSM Controller

In this section, we describe the design of a DSM controller that supports two DSM architectures (CC-NUMA and S-COMA) and three consistency protocols (sequentially consistent write invalidate, release consistent write invalidate, and release consistent write update). The results in Section 5 are based on an accurate model of this design.

Software can specify a combination of architecture and protocol for each shared variable via optional parameters to the `GMALLOC()` memory allocation routine. As a side effect of allocating shared storage, the operating system stores the protocol that should be used to manage each page in a table. Whenever the operating system allocates a shared page, or brings it back off of the disk, it copies the relevant protocol information to a table in the DSM controller via writes to a control register. As a result, each page can be managed with a separate protocol, even two pages (two variables) in the same program.

A typical node in a DSM system is illustrated in Figure 1. Individual nodes are composed of an SMP node with one or more commodity microprocessor's, each with its own private caches, connected to a coherent split-transaction SMP bus. Also on the memory bus is a main memory controller and a communication controller connected to a node interconnect. The aggregate main memory of the machine is distributed across all nodes. The processors, main memory controller, and communication controller all snoop the coherent memory bus, looking for memory transactions to which they must respond. Our multi-protocol DSM controller plays the role of a communication controller in this model.

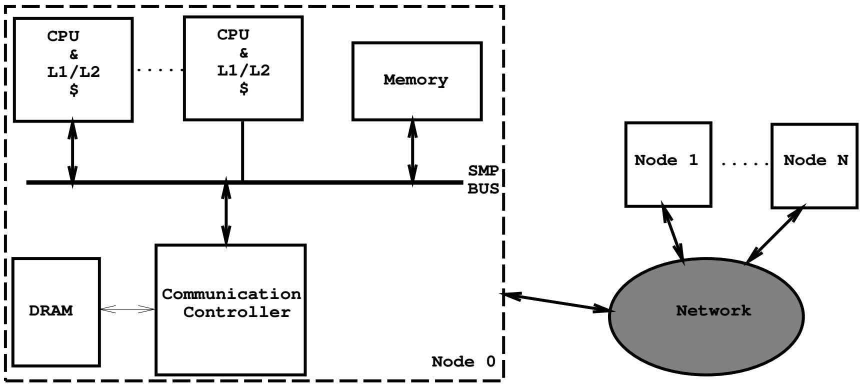


Figure 1. A Typical Node in a DSM System

The internals of a communication controller that supports our multiple protocol design are shown in Figure 2. It consists of a Bus Interface Diff/Merge (BIDM) unit, a multi-protocol consistency controller (MPCC), a Message Controller (MC), a Network Interface (NI) unit, a staging buffer that holds data going in and out of the controller, and some local DRAM storage that is used to store DSM state.

The BIDM is responsible for sniffing the system bus for transactions of interest (e.g., ones involving accesses to shared memory), handling bus arbitration, and performing on-the-fly *diffs* and *merges* of write update data. The details of this diff/merge operation are given below. The NI is simply a network interface that routes packets between the network and the MC and MPCC sub-controllers. The MC handles message passing traffic – the details of its operation are outside the scope of this paper. The MPCC contains the custom state machines necessary to implement the various consistency mechanisms supported in our design. The staging unit is simply a collection of SRAM where data can be placed while operations related to it are being performed.

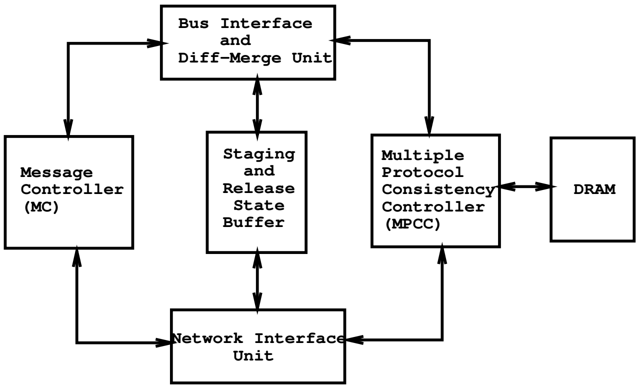


Figure 2. Internals of the Communication Controller

The basic operation of the communication controller is as follows. When a local processor makes an access to data that is not satisfied by its cache, a memory request is put on the system bus, where it is observed by the BIDM. If the request is for shared memory, the BIDM forwards the request to the MPCC. If the BIDM sees interesting data being written on the bus, e.g., a response to an MPCC-initiated read request or a writeback of shared data, it passes the control information to the MPCC unit and places the data in a staging buffer. Analogously, if the NI receives a DSM packet directed to the MPCC, it forwards the control portion of the packet to the MPCC and places the data (if any) in a staging buffer.

The core of our multi-protocol DSM support is handled by the MPCC, illustrated in Figure 3. The structure of the MPCC is similar to the FLASH macro-pipeline [15], except that custom state machines are used to implement the protocol operations, as opposed to a protocol processor. The MPCC is connected to each of the NI and BIDM through a pair of queues, one for input and one for output. After selecting which pending operation to handle, from either the front of the NI or BIDM input queues, the MPCC examines the page number in the address of the request and uses it as an index into the directory cache to read the protocol metadata associated with this page. This metadata consists of two pieces: the protocol to use for the page and the current state of the DSM cache line. These two pieces of information are stored separately, since one is per-page and one is per-line. Thus, the MPCC must perform two directory cache lookups before it can start executing, our single-protocol CC-NUMA

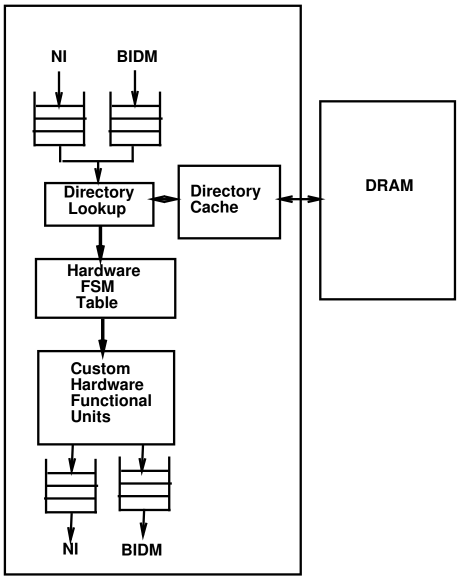


Figure 3. Internals of the MPCC

write-invalidate protocol implementation requires only one directory cache lookup (to find the cache line state). In the analysis in Section 5, we model both a dual-ported directory cache that can handle both accesses in single cycle as long as they fall in different banks and a single-ported cache that requires two cycles to read both pieces of metadata. If either entry is not in the directory cache, an 8-cycle delay is introduced while the data is read out of the DRAM.

Once the metadata is read, the operation type (6-bits) is concatenated with the cache state bits (4-bits) and the protocol bits (4-bits: two for the coherency protocol, one to select sequential or release consistency, and one to select CC-NUMA or S-COMA). This 14-bit value is used an input to the Hardware State Machine (HSM) Table. In response to this input, the HSM issues command signals to the various functional units, which generate request or reply messages, which are issued to the NI or BIDM as appropriate. The HSM also sends the new state of the cache line to the message decoder unit and updates the directory cache. If both pieces of metadata hit in the directory cache and there is no stall in any part of the protocol pipeline, the total latency to process any request in the MPCC is 3 cycles, excluding the time to enqueue and dequeue the operations.

To illustrate how the controller works, we will walk through an example of a read miss to a line being managed with the sequentially consistent S-COMA write-invalidate protocol. In response to the read miss, the processor issues a *read share* request on the bus. The BIDM notices this operation and issues *bus read share* operation, including the physical address, to the MPCC queue. The BIDM also stashes away the bus transaction ID and the address for later use. The message decoder reads the state and protocol and applies the concatenation of (SC, SCOMA, WI, INVALID) to the HSM table. The rest of the state information, e.g., the data's home node, is passed directly to the functional unit that needs that information. The HSM table issues a command to the message encoding unit, which forms a *read shared* request message and adds it to the NI queue. When the read reply is received, the MPCC issues a *cache to cache reply* command to the BIDM. The BIDM looks up the transaction ID associated with that address, which it stored earlier in the process, and returns the data on the bus. Since the data that is returned is for a local memory address, the memory controller also grabs the data and writes it to the local copy of the page.

All the protocols use a full-map directory scheme. The CC-NUMA write-invalidate protocol is similar to invalidation-based write-back protocol implemented in DASH [18], except remote owners send the reply data to the home node, which then forwards it to the requester. We implemented it this way to avoid a number of deadlock situations. Our implementation of the release consistent write invalidate protocol is also similar to the one in DASH.

The update protocol is similar to Grahn's competitive-update protocol [7]. Cache lines handled via the update protocol can be written or read by multiple nodes simultaneously. When a processor wants to write to a line it issues a *read exclusive* request. If the line is not cached in local memory, the MPCC issues a *read shared* request to the home node. If it is cached in local memory, the MPCC allows the memory controller to supply the data and issues a *read capture* command to the BIDM. When the data returns from the remote node, the MPCC issues a *bus cache-to-cache-reply no-deallocate* command to the BIDM. The BIDM inserts this address in a lookup-table known as the *Release State Table* (RST) along with the index of where the cache line data is stored in the staging buffer. The clean copy of the line is not deallocated from the staging buffer after it is supplied by the processor. When the number of entries in the RST crosses a threshold or the application signals a release synchronization point by issuing a read to a register in the BIDM, the BIDM selects an entry from RST, issues a *read shared* request on the bus, and reads the clean block from the staging buffer into an internal FIFO. As the data returns back on the bus, each double-word is fed to a *diff* engine that determines the differences between the clean and potentially dirty line at a word granularity. The *diff* engine keeps track of what words differ and stores them into a separate buffer. It also maintains a mask of the words that were changed. Once the *diff* is complete, the BIDM issues a *send update* to the MPCC along with the 32-bit *diff* mask. The BIDM repeats this process of selecting an RST entry and creating its *diff* until the number of entries in RST drops below a preconfigured threshold or, in the case where the processor has issued a release synchronization, until the RST is empty. To avoid overloading the RST and network, the update protocol is used only for the data objects with true producer-consumer sharing.

Upon receiving a *send update* command from the BIDM, the MPCC sends the update to the home node via the NI. The NI examines the mask and transmits only the dirty words from the staging buffer. The mask is attached to the network header. Upon receipt of the update, the MPCC on the home node issues a *read exclusive merge* request to its BIDM, along with the mask. The BIDM issues a *read exclusive* request on the bus to invalidate the line if it is in a processor cache. When the data returns in response to the *read exclusive* operation, the BIDM merges the update with the flushed back data and writes the merged cache line back to memory. In parallel with performing the merge, the home node forwards the update to all nodes with a copy of the cache line and sends a *write-update ack* message back to the node that initiated the update along with a count of the number of other nodes sharing the data. Upon receive of an update, each of these nodes applies the update and sends a *release ack* message to the node where that performed the update. To ensure release consistency, the MPCC on the initial node maintains two counters - one that keeps track of the number of updates sent while the other keeps track of the number of acknowledgments received. The update counter is incremented whenever an update is sent and decremented whenever an update acknowledgement is received. The release counter is incremented with the number of acknowledgements that it is waiting to receive and is decremented whenever a release acknowledgement is received. Only when both counters reach are zero and the RST is empty can the processor that performed the release operation be allowed to continue, which is done by (finally) responding the the I/O register read that initiated the release in the first place.

This explanation omits quite a few subtle design issues that are needed to avoid deadlocks and race conditions. A complete description will appear in a companion technical report.

4 Performance Evaluation

4.1 Experimental Setup

All experiments were performed after modifying the execution-driven simulation of the *elided-for-your-benefit* architecture. Our simulation environment includes detailed simulation modules for the first and the second level cache, system bus, memory controller, network interconnect, and DSM engine. It provides a multiprogrammed processor model with support for operating system code, so the effects of OS/user code interactions are modeled. The simulation environment includes a kernel based on 4.4BSD that provides scheduling, interrupt handling, memory management, and limited system call capabilities. The modeled physical page size is 4 kilobytes. The VM system was modified to provide the page translation, allocation, and replacement support needed by the various distributed shared memory models. However, we assume that enough memory is available at each node for home pages and for replicating any number of S-COMA pages. We used the first touch algorithm [20] to distribute home pages to nodes.

The DSM controller, the system bus, and the network are all clocked at 120MHz. All cycle counts reported herein are with respect to this clock. The model of the processor and bus interconnect, the system bus and memory controller are similar to the one found in HP KittyHawk systems [11]. The processor is clocked at 360Mhz. The processor model does not include several features present in modern microprocessors, such as speculation and out-of-order issue. However, we do approximate the behavior of a 4-issue processor allowing up to four instructions to issue per cycle without checking for hazards. We damp runaway speculation by accurately modeling contention and queueing at the load/store unit. The processor model simulates the PA-RISC 1.1 [9] instruction set.

For all of our experiments, we model a direct mapped 8-kilobyte L1 data cache with 32-byte lines. The L1 cache is virtually indexed and physically tagged. The L2 cache is 4-way set associative with 128-byte lines. Unless otherwise specified for a particular experiment, we model a 256-kilobyte L2 cache. It is physically indexed and physically tagged, non-blocking, and can support up ten outstanding misses. We assume perfect I-cache behavior. Finally, we model a 4-bank main memory controller with 100ns DRAM. The block size of shared memory is equal to the L2 cache line size, i.e., 128 bytes.

We simulated an 8-node multiprocessor, where each node has a single processor. The main reason we restricted our experiments to eight nodes was simulation costs. We could not run more than one processor per node as the operating system kernel that we model does not support SMP nodes.

For all of our experiments other than the ones in which we explicitly study the effects of the directory cache size and organization, we assume a directory cache with 1-cycle latency to read the directory and remote data state information. The simulated release state table contains 64-entries and is fully associative, with a flush threshold of 44 (70%).

Local Read	Cycle
Issue on bus	3
Sent to DRAM	5
Data from DRAM	20
1st d-word to CPU	24
Remote Read (800MB Network)	Latency
Issue on bus	3
Request sent on wire	8
Request at home NI	24
Read on bus	38
Reply at requester NI	80
1st d-word to CPU	96
Remote Read (200MB Network)	Latency
Read on bus	39
Reply at requester NI	81
1st d-word to CPU	139

Figure 4. Breakdown of Read Miss to Local and Remote Memory (in 120MHz bus cycles)

We modeled three interconnects with 3 different bandwidths: 200MBps, 400MBps and 800MBps. All the three networks had an end to end latency of 16 cycles. The 800MBps network is similar in bandwidth to that of the Craylink used in the SGI Origin 2000 [2]. By employing a high bandwidth network, such systems try to balance the performance of the different parts of the system. The 200MBps network is comparable in bandwidth to a Myrinet [2] network, while the 400MBps model represents an alternative between the two extremes. These three interconnects represent a set of reasonable design alternatives that could be selected by a DSM system architect. We simulate only input port contention in the network.

Finally, Figure 4 shows the breakdown of no-contention latency while accessing local and remote memory. A read miss to local memory takes 200ns. A read miss to remote memory takes 800ns using the high bandwidth network and 1160ns using the low bandwidth network. Even though the first piece of data arrives at cycle 81 for the slow network, it can only be supplied to the bus at cycle 138, because we must wait for the entire 128-byte cache line to be delivered before we can arbitrate for the bus.

4.2 Benchmark Programs

We ran all of the programs from the SPLASH-2 benchmark suite [30], except for *radiosity* and *raytrace* which had uninteresting behavior and very long simulation times. We also ran *em3d* and *lcp* from the Wisconsin Wind

Tunnel suite [24] and *mp3d* from SPLASH-1. Figure 5 shows the inputs used for each test program. Except for changes to the global memory allocation interface (*G_MALLOC*) to specify the desired consistency mechanism and minor changes to make some of the programs release consistent, no other data restructuring or reorganization was done to the benchmark programs.

Program	Input parameters
radix	4M Keys, Radix = 128, Max Key = 2M
fft	256K Points, tuned for cache sizes
LU (contiguous)	512x512 matrix, 16x16 blocks
LU (contiguous)	512x512 matrix, 16x16 blocks
cholesky	`tk23` input, tuned for cache sizes
ocean (contiguous)	258x258
ocean (non-contiguous)	258x258
water (N-squared)	512 molecules, 10 steps
water (spatial)	512 molecules, 3 steps
volrend	head
em3d	5000 bodies
lcp	System size 4096
mp3d	10000 Molecules and 10 steps

Figure 5. Programs and Problem Sizes Used in Experiments

5 Results

Our experiments fall into two classes. The first set of experiments evaluate the value of supporting multiple DSM protocols in hardware. The second set of experiments evaluate the tradeoffs involved with supporting DSM using a custom single-protocol DSM controller, a custom multi-protocol DSM controller, and a programmable protocol processor.

5.1 Impact of Supporting Multiple DSM Protocols

The most basic question to answer is whether or not support for multiple protocols in hardware provides sufficient performance benefits to warrant the greater design complexity of this support. Figures 6 and 7 present the execution time of seven applications simulated on five different DSM architectures and three different networks. The architectures considered are *CC-NUMA* (a sequentially consistent CC-NUMA architecture that employs a single write invalidate coherency protocol), *SCOMA* (a sequentially consistent Simple COMA architecture that employs a single write invalidate coherency protocol), *RC-SCOMA* (a *release* consistent Simple COMA architecture that employs a single write invalidate coherency protocol), *MULTI* (the multi-protocol DSM architecture described in Section 3), and *OPTIMAL* (an ideal memory architecture where all conflict misses are satisfied from local memory). For each application, we perform each experiment with three different network models: *FAST*

(800 MBps), *MEDIUM* (600 MBps), and *SLOW* (200 MBps). The execution time denoted by each vertical bar is divided into four categories: *SUSP* (the time that the application spends suspended in the kernel), *I+LM* (the time spent executing instructions and accessing local memory), *SYNC* (the time spent performing synchronization), and *SM* (the time spent stalled accessing shared memory).

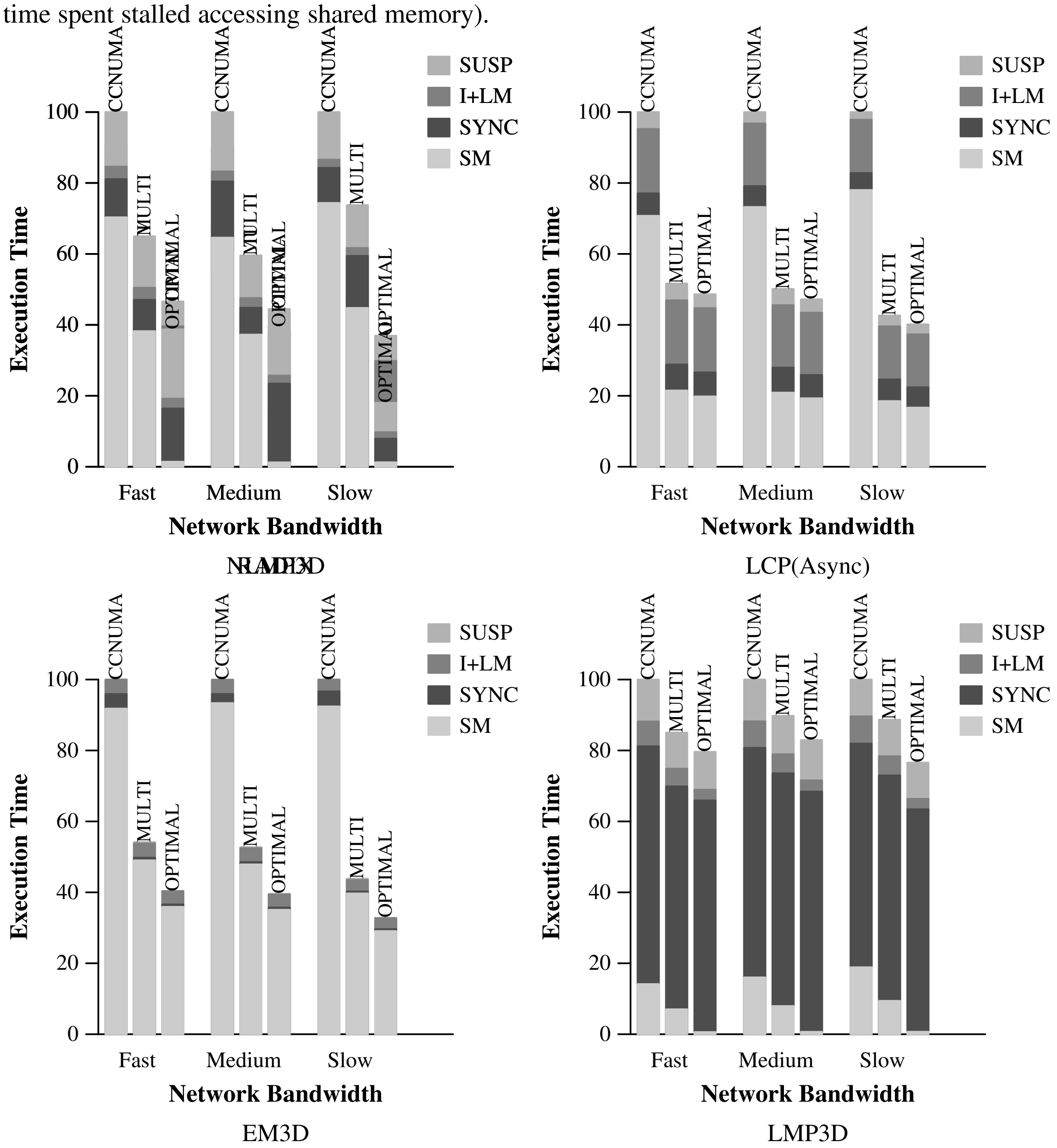
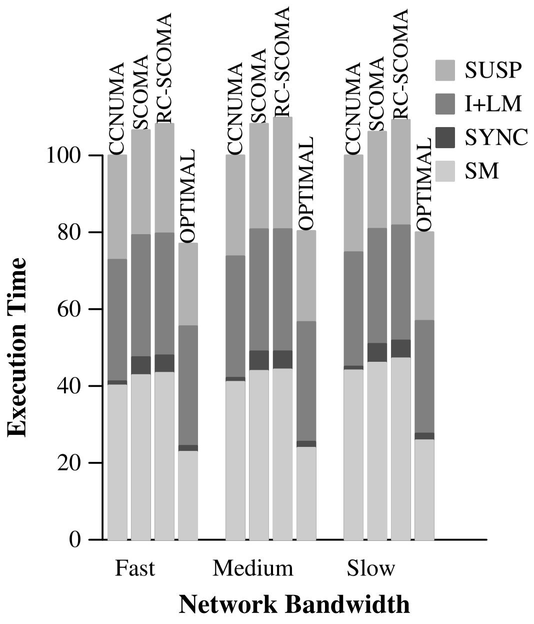


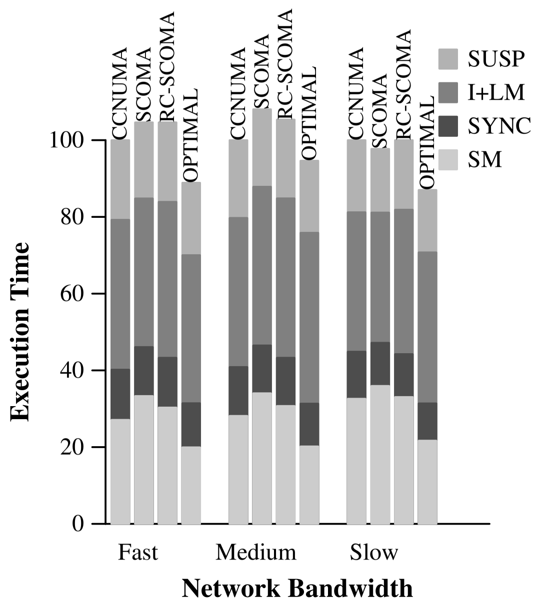
Figure 6. Relative Execution Time of `radix`**,** `lcp`**, and** `em3d`

The results presented in Figures 6 and 7 indicate that there are substantial benefits to supporting multiple DSM mechanisms on a single machine.

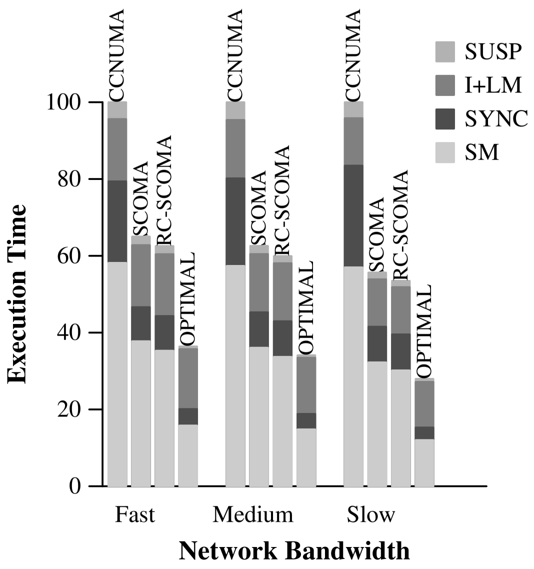
Figure 6 presents the results for three applications where different data structures within the same application are best managed using different protocols. In particular, while an S-COMA employing a release-consistent write



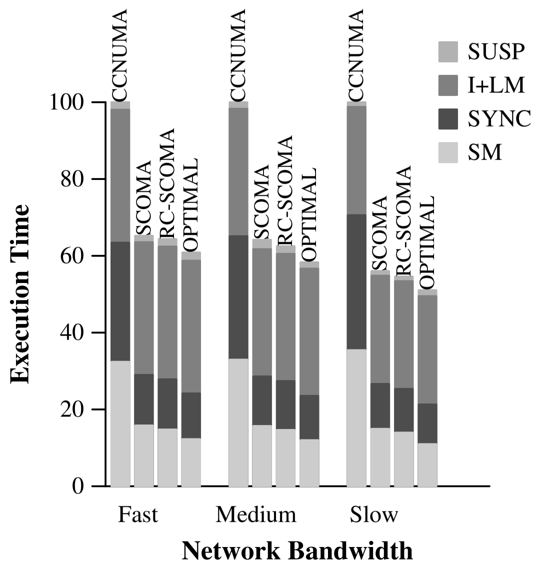
FFT



Cholesky



Ocean(Non-contiguous)



LU(Contiguous)

Figure 7. Relative Execution Time of `fft`, `cholesky`, `ocean (non-contiguous)` **and** `lu (contiguous)`

invalidate protocol works best for most data structures, the *rank* structure in *radix*, the *E* and *H* nodes in *em3d*, and the *solution* vector in *lcp* are best managed by an S-COMA architecture employing a release-consistent *write update* protocol; the *key* array in *radix* is best managed by a CC-NUMA architecture employing a release-consistent write invalidate protocol. By allowing the programmer (or compiler) to specify that these data structures should be managed thusly, a DSM architecture that supports multiple protocols improves performance by 42%-58% for these applications.

Figure 7 presents the results for four applications where performance changes substantially depending on the choice of architecture and coherence protocol employed. For these programs, it is best to manage all data using the same architecture and coherence protocol. For *fft* and *cholesky*, a conventional CC-NUMA architecture with a write invalidate protocol performs best; unfortunately, no combination of architecture and coherence protocol performs within 10% of *OPTIMAL* for these applications. We saw similar results for *ocean* (contiguous) and *mp3d*. The reason that CC-NUMA outperformed S-COMA for these three applications (*fft*, *cholesky* and *radix*) is that when a dirty line is replaced in CC-NUMA, it is flushed all the way back to its home node. In S-COMA, it is flushed only to local memory. Flushing dirty lines that are replaced from the processor cache(s) back to their home node is in essence a form of dynamic self-invalidation [17]. All four of these applications had data that exhibited one of two behaviors: (i) producer-consumer sharing in which the home node was a consumer or (ii) multiple-writer false sharing. In either case, when a node writes to an object that it is not going to read later, it is best to use a CC-NUMA protocol for the object. This reduces the number of hops taken by the coherency traffic from three to two, reduces memory pressure, and thereby improves performance by 4% for *cholesky* and 5% for *fft*.

The relative performance of S-COMA and CC-NUMA tends to change as processor cache sizes change, since the value of S-COMA's local caching mechanism is strongly dependent on the cache conflict miss rate. Thus, we measured the sensitivity of these applications to L2 cache size. In all cases, CC-NUMA outperformed S-COMA for L2 cache sizes ranging from 64-kilobytes to 2-megabytes. Hence, a CC-NUMA architecture employing a write invalidate coherence protocol is the best option for these applications.

In contrast to *fft* and *cholesky*, *ocean* (non-contiguous) and *lu* (contiguous) substantially from using an S-COMA architecture. In the case of *ocean* (non-contiguous), this choice captures over 50% of the potential performance improvements achievable with an optimal coherence mechanism. When an S-COMA architecture is used to manage the data of *lu* (contiguous), performance is within 3% of *OPTIMAL*. We saw similar results for *volrend* and *water* (n-squared). When we performed a sensitivity analysis similar to the one described above, we found that the relative importance of employing an S-COMA architecture increase as L2 cache size decreased, as expected.

Looking at both Figures 6 and 7, we see that the value of supporting multiple coherence mechanisms is high even when using an expensive high-bandwidth network – there is a significant gap between the performance of any single-protocol design and the *OPTIMAL* performance. As expected, the performance gap increases when one uses a less expensive, low-bandwidth commodity network.

Taken together, Figures 6 and 7 provide strong support for the use of programmable DSM controllers [8, 25] that can support a variety of architectures and protocols.

Given the results from Figure 6, one might wrongly conclude that the exclusive use of a write update protocol will lead to optimal performance. Although a write update protocol works well for producer-consumer data, it works very poorly when updates cannot be coalesced effectively, either because the number of objects being updated constantly overflows the RST or there is no locality to the updates. Figure 8 compares the performance of *radix*, *fft* and *cholesky* using only an update protocol or only an invalidate protocol for all data. *radix* runs over five times faster using write invalidate while *fft* runs twice as fast. In *cholesky*, the use of an update protocol reduces the amount of time the application stalls waiting for shared memory. However, performance does not improve because the shared memory stalls are replaced with synchronization stalls as the application waits at the release synchronization point collecting update acknowledgements.

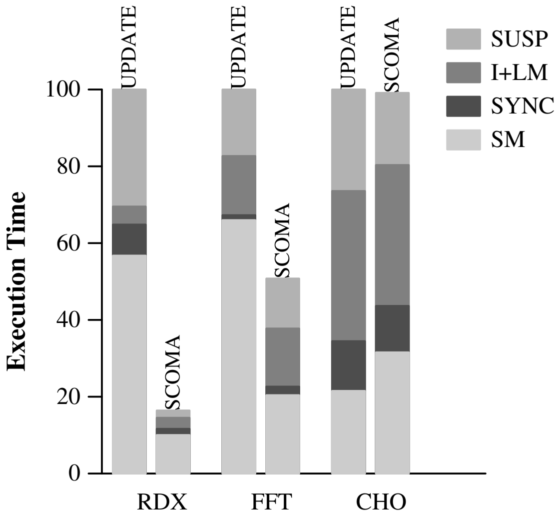


Figure 8. Relative Performance of Pure Write Update and Pure Write Invalidate

Finally, there are also applications for which any combination of the simple update and invalidate protocols we model is insufficient to achieve performance within a small fraction of *OPTIMAL*. For these applications, e.g., *fft* and *ocean* (non-contiguous), a much finer grained protocol must be employed. The study of such protocols is beyond the scope of this paper.

5.2 Why Multiple Protocols Benefit Performance

To understand why supporting multiple protocols benefits performance, we can examine the types of cache misses that their use eliminates. Figure 9 provides a breakdown of the types of cache misses suffered by the seven applications detailed in Figures 6 and 7. We divide cache misses into five categories: (i) misses to non-shared memory (*NSM*), (ii) conflict misses to shared data satisfied by local memory (*LCC*), (iii) coherence misses to shared data satisfied by local memory (*LCOH*), (iv) conflict misses to shared data satisfied by remote memory (*RCC*), and (v) coherence misses to shared data satisfied by remote memory (*LCOH*).

An update protocol can eliminate remote coherency misses by not invalidating and subsequently reloading producer-consumer data. This benefit of update protocols is clearly the primary source of performance improve-

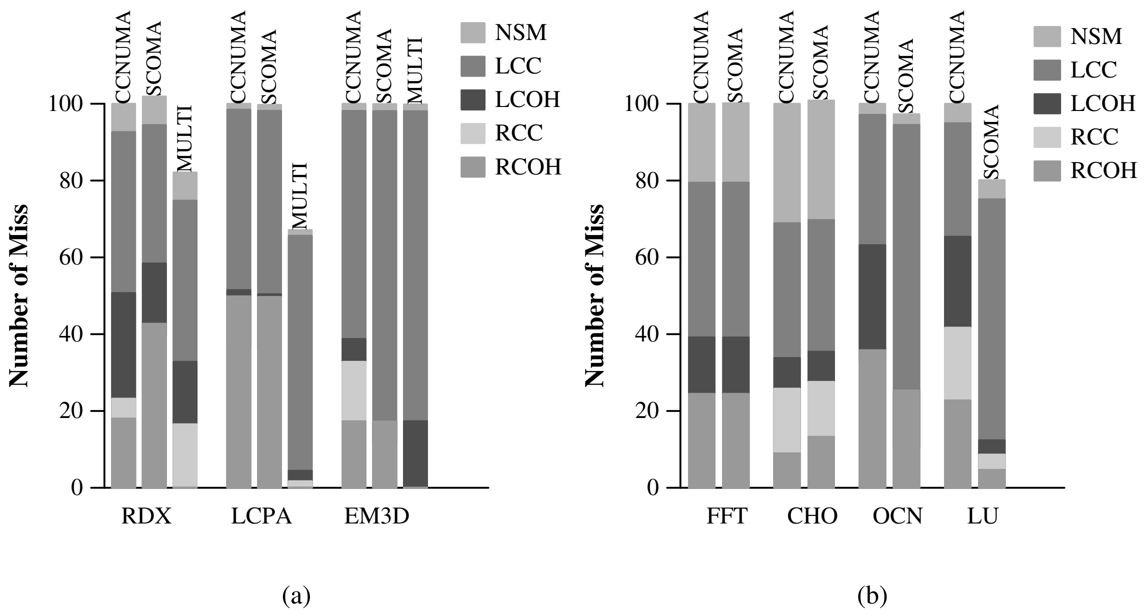


Figure 9. Breakdown of Types of Cache Misses Experienced

ment for *radix* and *lcp*. However, using an update protocol can increase the number of local conflict misses by leaving data in the processor cache that will not be accessed by the local node in the near future.

The use of a CC-NUMA architecture also can convert remote misses into local misses, thereby improving performance. This phenomenon can be observed in Figure 9(a) for *radix* and Figure 9(b) for *cholesky*.

5.3 Impact of Protocol Choice on Network Usage

Figure 10(a) presents the relative networking requirements of various mechanisms, broken down by message types. The message types are updates (*UPDATE* – only relevant for the *MULTI* mechanism), acknowledgements (*ACK*), data (*DATA*), and requests (*REQ*). As shown in the figure, the occasional use of an update protocol can eliminate 66-95% of request and data messages.

Similarly, the use of an update protocol can reduce the number of bytes transferred dramatically. It does so by sending only the *modified* words in a cache line instead of the whole line when it sends back an update message. In contrast, when a particular piece of data on a node is invalidated and subsequently accessed again, a full cache line of data is sent. As a result, an update protocol is able to reduce the network bandwidth consumed by the applications by 40% to 90% (Figure 10(b)). This property of write update protocols is one of the key reasons that the multi-protocol design performed well when coupled with a low-bandwidth network.

5.4 Sensitivity Analysis: Machine Size

Figure 11 contrasts the relative benefits of employing a multi-protocol scheme on 8 and 32 nodes for two representative applications. For *radix* and *lcp*, the use of a multi-protocol controller improves performance dramatically

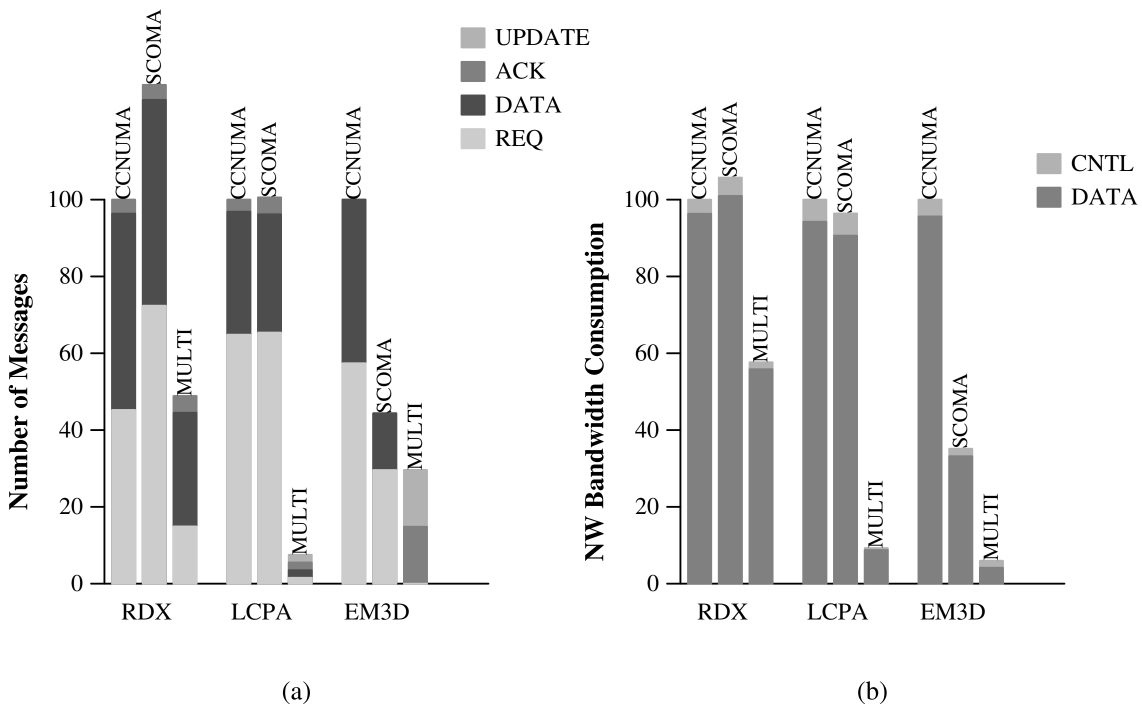


Figure 10. (a) Types of Messages Sent (b) Network Bandwidth Consumed

compared to a conventional write invalidate S-COMA system. The multi-protocol system runs about twice as fast as a conventional system for *radix* and about ten times as fast for *lcp*.

The benefit of employing a multi-protocol controller is higher at 32-nodes for *lcp*, because the percentage of time spent stalled on shared memory in the conventional system increases from 75% to 93%. However, the difference between a conventional design employing only a S-COMA write invalidate protocol and a multi-protocol machine is *smaller* on a 32-node machine than on an 8-node machine for *radix*. We believe that this result is due to the fact that we did not scale the input size, which led to a large increase in the percentage of time the application spent suspended in the kernel due to load imbalance. In any event, the benefit of a multiple protocol DSM controller is not particularly sensitive to the size of the modeled configuration.

5.5 Custom versus Programmable DSM Controllers

Although programmable DSM controllers have appeared in research machines [8, 25], an argument has been raised that occupancy issues make them inherently inefficient [21, 10]. Since support for multiple coherence mechanisms would be relatively easy to support on a programmable DSM controller, we simulated three DSM controller configurations: (i) a single protocol custom controller (SPCC), (ii) a multiple protocol custom controller (MPCC), and (iii) a programmable DSM controller (PPC). The two custom controllers (SPCC and MPCC) implement their protocols in dedicated custom logic, while the programmable controller implements its protocols in firmware using an embedded protocol processor.

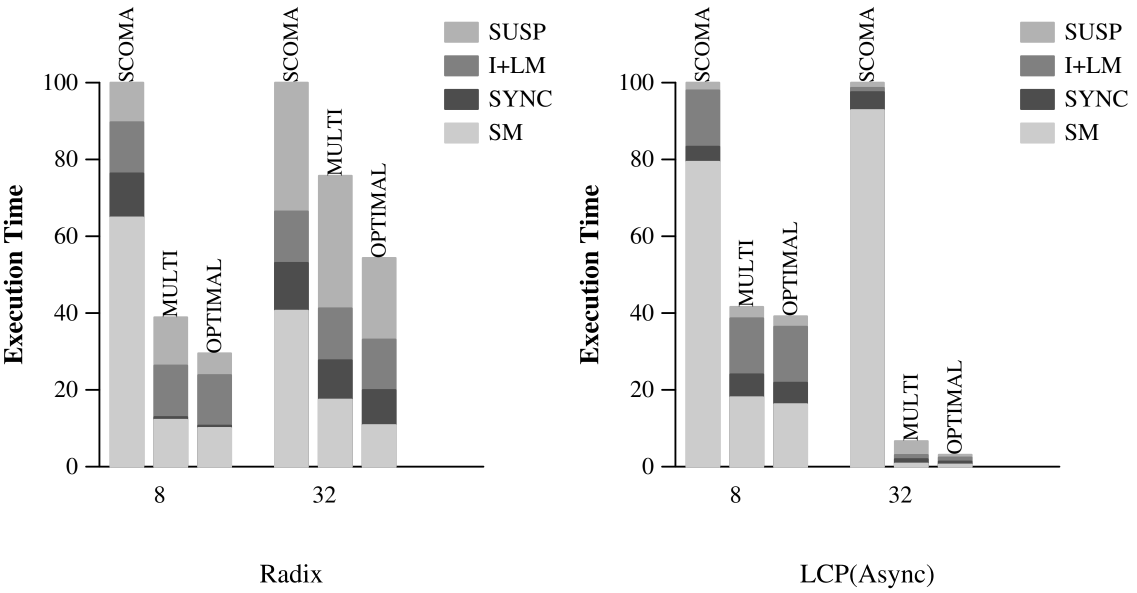


Figure 11. Sensitivity of Performance to the Number of Nodes (8 vs 32) *radix* **and** *lcp*

The simulated latency of various protocol operations are reported in Figure 12. PPC latency and occupancy are based on results reported by Michael et al. [21]. We model two different multi-protocol controllers: one that runs at the same speed as a single-protocol controller (*MPCC*) and a less aggressive design that runs at half the effective speed (*MPCC-*). We include the *MPCC-* model in our simulations to investigate the sensitivity of performance to the speed of the controller.

To determine the impact that controller design has on performance, we simulated the MPCCs with 4-way set associative directory caches that varied in size from 2 kilobytes to 32 kilobytes.

The *MPCC-* model is designed to capture the potential (worst case) impact of two sources of overhead in a multi-protocol controller compared to a similar technology single-protocol controller. First, a multiprotocol controller is more complex than a single-protocol controller, which might lead to an increase in the time required to perform each operation. Second, a multi-protocol controller requires access to more state information, which increases the demands on the directory cache. The faster *MPCC* model assumes the use of a dual-ported cache that allows access to two data structures in one cycle; the slower *MPCC-* model assumes that the directory cache must be accessed twice (2 cycles total) to read the necessary data structures. For both models we assume that directory cache misses require 8 cycles. Our PPC model assumes a dual-ported data cache and a perfect PPC instruction cache.

We simulated the performance of four applications (*lu* (contiguous), *cholesky*, *fft*, and *lu* (non-contiguous)) on all four of the described controller design options. We used a sequentially consistent CC-NUMA write invalidate protocol for all of the applications so that we could make a direct comparison of the overheads of the various

[1]Custom state machines do not perform explicit "branch" operations. Also, diff and merge operations are performed in parallel with protocol processing in the custom designs.

Operation	SPCC	MPCC	MPCC-	PPC
Handler invocation	1	1	2	6
Bus request	1	1	2	4
Response detect	1	1	2	4
Issue message	1	1	2	4
Directory cache hit	1	1	2	1
Directory cache miss	8	8	8	8
Condition check and branch	-[1]	-	-	2
Diff per dirty word	-	-	-	3
Diff per clean word	-	-	-	2
Merge per dirty word	-	-	-	3
Merge per clean word	-	-	-	2

Figure 12. Execution Time of Protocol Operations

implementation strategies without clouding the results with differences in the protocols implemented. Because of this constraint, the custom single-processor controller will always outperform the other designs - the question is by how much.

Figure 13 shows the results of this experiment. In all cases, PPC performance was within 20% of that of a custom single-protocol controller. The performance of the fast multi-protocol custom controller (*MPCC*) was within 2% of the single-protocol design with a 2-kilobyte directory cache, and equivalent when the directory size was at least 4 kilobytes. The performance of even the slower multi-protocol custom controller (*MPCC-*) was always within 4% of that of the SPCC.

Figure 14 shows how a programmable DSM controller performs compared to a custom controller when a diff-and-merge update mechanism is needed. With a PPC, the *diff* and *merge* operations are done by the protocol handlers. To model the overhead of performing *diffs* and *merges* in firmware, we assume a perfect cache for the *diff* and *merge* data, and only perform *diff* operations when the PPC is idle. Once started, the operation runs to completion without interruption. The *merge* operation is invoked upon receipt of an update message. We find that even when implementing the more compute-intensive update protocol, the performance of programmable protocol processor design is within 20% of a design employing a custom solution. When the benefits of exploiting multiple protocols are consider, the PPC design outperformed the SPCC design by 30% to 80%. This result indicates that there is definite potential for improving the performance of FLASH and Typhoon through the introduction of multiple coherence protocols in their protocol processors.

Figure 14(b) illustrates a problem with the PPC architecture that is hiding just below the surface. While the PPC-based multi-protocol system performs well, its occupancy is considerably higher than that of the custom solutions. This indicates that the PPC will saturate first as we increase the number of processors per node, which will ultimately limit the scalability of a PPC-based solution.

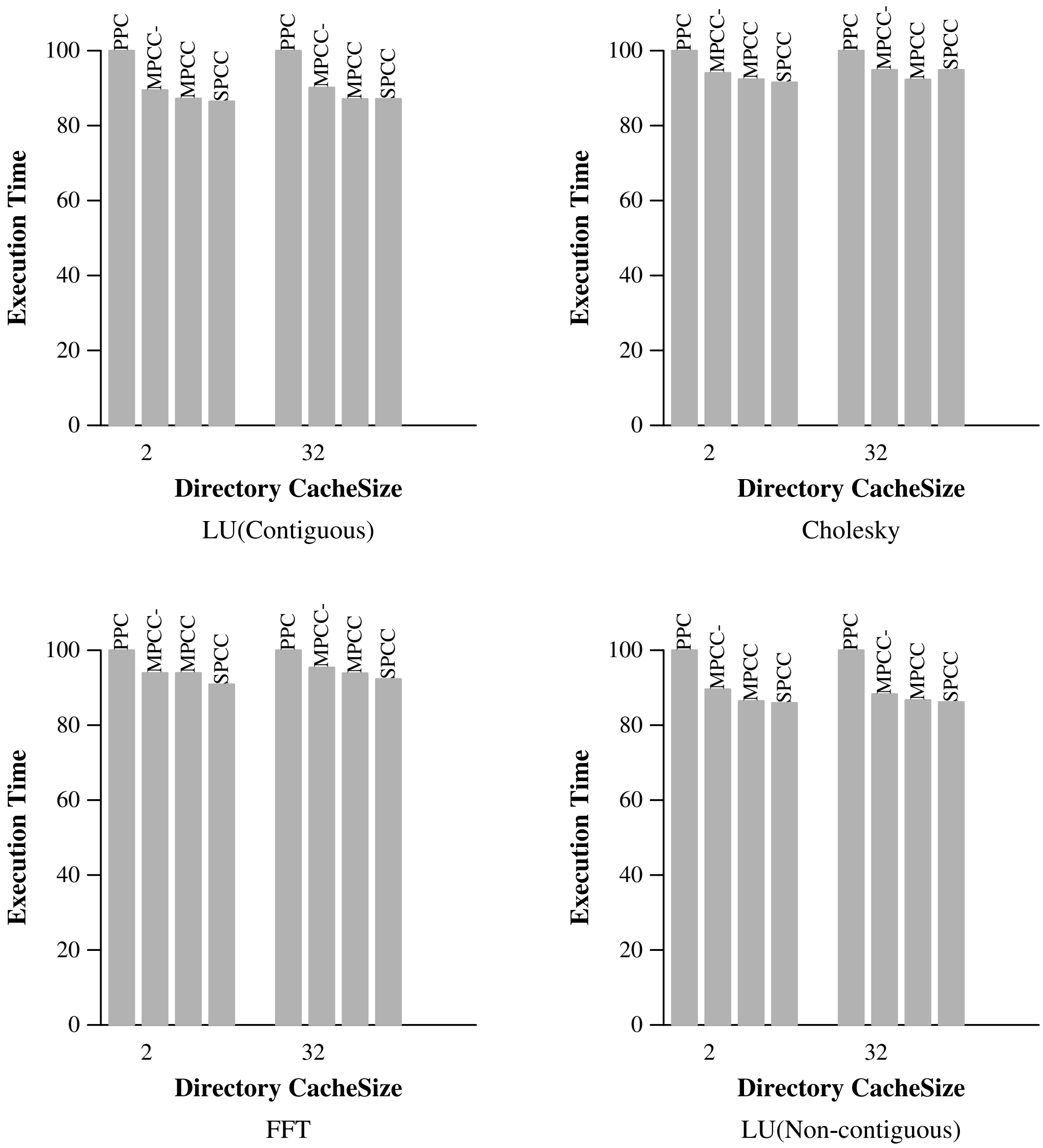


Figure 13. Comparison of MPCC, PPC and SPCC Architecture

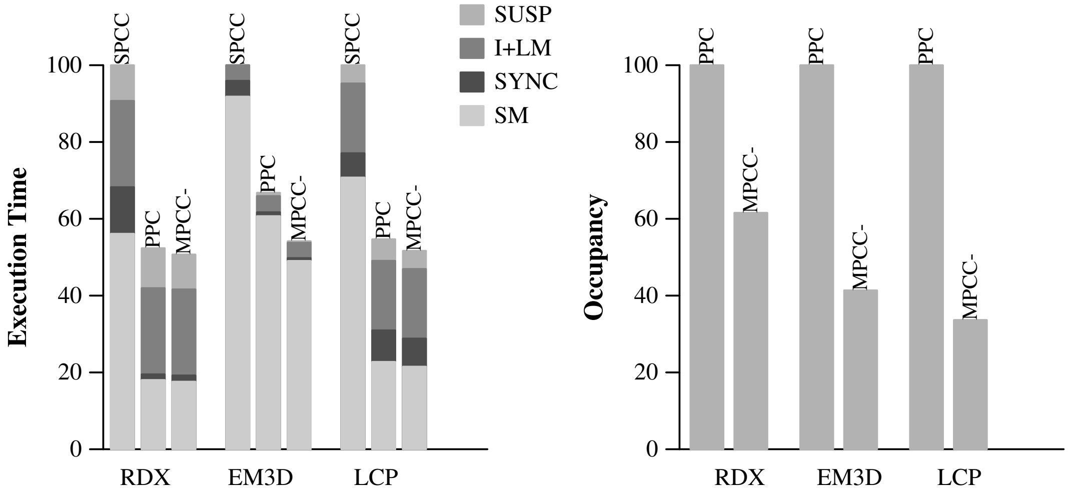


Figure 14. Comparison of MPCC and PPC Designs with Multiple Protocols In-Use

6 Conclusions

In this paper, we characterized the performance of an *optimal* DSM controller on a large number of applications and show that conventional DSM controller designs have considerable room for improvement compared to such an optimal system.

We then discussed the design of a custom hardware DSM controller that supports multiple architectures (release consistent and sequentially consistent CC-NUMA and S-COMA) and coherence protocols (write invalidate and write update). This design allows different applications to use different protocols, and even allows a single application to use different protocols to manage different data. It employed a novel *release state table* mechanism to implement an efficient release consistent write update protocol in hardware.

We then measured the performance of a set of programs from the Splash-2 and Wisconsin Wind Tunnel benchmark suites on the optimal DSM system, a conventional custom single-protocol DSM controller, our custom multi-protocol DSM controller design, and a programmable DSM controller. In this study, we found architectures employing a single write-invalidate CC-NUMA protocol can perform up to a factor of three worse than the optimal system. We also found that exploiting a combination of local memory caching, release consistency, and write update protocols can reduce the execution time of some applications by up to 60% and reduce network bandwidth consumption can be reduced by up to 95% compared to an architecture with just the write-invalidate protocol.

Overall, this work demonstrates the value of supporting a degree of flexibility in one's DSM controller design and suggests what operations such a flexible DSM controller should support.

References

[1] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[2] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO*, 15(1):29–36, February 1995.

[3] D. Chaiken and A. Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[4] M. Dubois, J. Skeppstedt, L. Ricciulli, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 88–97, May 1993.

[5] B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing'94*, pages 380–389, November 1994.

[6] B. Falsafi and D.A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[7] H. Grahn, P. Stenström, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247–271, June 1995.

[8] M. Heinrich and J. Kuskin et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[9] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[10] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford University, 1995.

[11] T. Hotchkiss, N. Marschke, and R. McClosky. A new memory system design for commercial and technical computing products. *Hewlett-Packard Journal*, 47(1):44–51, February 1996.

[12] L. Iftode, C. Dubnicki, E. Feltin, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 26–37, February 1996.

[13] D. Jiang and J. P. Singh. Scaling application performance on cache-coherent multiprocessors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[14] S. Kaxiras and J. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, January 1999.

[15] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, May 1994.

[16] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *ISCA97*, pages 241–251, June 1997.

[17] A. Lebeck and D. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[19] T. Lovett and R. Clapp. STiNG: A CC-NUMA compute system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[20] M. Marchetti, L. Kontothonassis, R. Bianchini, and M.L. Scott. Using simple page placement policies to reduce the code of cache fills in coherent shared-memory systems. In *Proceedings of the Ninth ACM/IEEE International Parallel Processing Symposium (IPPS)*, April 1995.

[21] M. M. Michael, A. K. Nanda, B.H. Lim, and M. L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 219–228, June 1997.

[22] A. Moga and M. Dubois. The effectiveness of SRAM network caches in clustered DSMs. In *Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture*, 1998.

[23] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.

[24] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[25] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[26] A. Saulsbury, T. Wilkinson, J.B. Carter, and A. Landin. An argument for Simple COMA. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 276–285, January 1995.

[27] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[28] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[29] W. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *ISCA97*, June 1997.

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.