To appear in the *Proceedings of the Sixth International Conference on Parallel Architectures and Compilation Techniques.*

Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory

Ioannis Schoinas[♦], Babak Falsafi[†], Mark D. Hill[*], James R. Larus[*], and David A. Wood[*]

[♦]Server Architecture Lab
Intel Corporation
CO3-202
15220 NW Greenbrier Pkwy
Beaverton, OR 97006 USA
Email: ioannis_t_schoinas@intel.com

[†]School of Electrical and Computer Engineering
Purdue University
1285 Electrical Engineering Building
West Lafayette, IN 47907-1285 USA
URL: http://www.ece.purdue.edu/~babak
Email: babak@ecn.purdue.edu

[*]Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706-1685 USA
URL: http://www.cs.wisc.edu/~wwt
Email: wwt@cs.wisc.edu

Abstract

Software fine-grain distributed shared memory (FGDSM) provides a simplified shared-memory programming interface with minimal or no hardware support. Originally software FGDSMs targeted uniprocessor-node parallel machines. This paper presents Sirocco, a family of software FGDSMs implemented on a network of low-cost SMPs. Sirocco takes full advantage of SMP nodes by implementing inter-node sharing directly in hardware and overlapping computation with protocol execution. To maintain correct shared-memory semantics, however, SMP nodes require mechanisms to guarantee atomic coherence operations. Multiple SMP processors may also result in contention for shared resources and reduce performance. SMP nodes also impact the cost trade-off. While SMPs typically charge higher price-premiums, for a given system size SMP nodes substantially reduce networking hardware requirement as compared to uniprocessor nodes.

In this paper, we ask the question "are SMPs cost-effective building blocks for software FGDSM?" We present experimental measurements on Sirocco implementations ranging from an all-software system to a system with minimal hardware support. Together with simple cost models we show that low-cost SMP nodes: (i) result in competitive performance with uniprocessor nodes, (ii) substantially reduce hardware requirement and are more cost-effective than uniprocessor nodes, (iii) significantly benefit from hardware support for coherence operations, and (iv) are especially beneficial for FGDSMs with high-overhead coherence operations.

1 Introduction

Clusters of small-scale symmetric multiprocessors (SMPs) are emerging as a promising approach to building cost-effective large-scale parallel computers. The relatively high volumes of small-scale SMP servers make them extremely cost-effective as building blocks. By connecting these low-cost nodes, system designers hope to construct large-scale parallel machines with better cost-performance than has been previously possible [4].

To preserve application compatibility while maintaining a low system cost, many designers implement software distributed shared memory (DSM) over a network of SMPs. Most software DSMs [7,15] use standard virtual memory translation mechanisms to maintain coherence at a page granularity (or larger) across SMPs. Transparent page-level coherence, however, can result in poor performance for applications with fine-grain sharing.

Alternatively, some systems implement fine-grain distributed shared memory (FGDSM) which allows for sharing data across the nodes at a cache block (e.g., 32-128 byte) granularity. FGDSMs are particularly attractive for implementing DSM on a network of SMPs because they transparently (i.e., without the involvement of the application programmer) extend the SMP fine-grain shared-memory abstraction across all the nodes.

SMP nodes provide an opportunity to improve performance in software FGDSM [22]. By sharing a single large memory cache for remote data among multiple processors, SMPs improve memory utilization. Processors *within* a node can directly share memory using fast SMP hardware mechanisms. Multiple processors can overlap computation with protocol handling to reduce execution time. SMP nodes can also reduce remote miss frequency by allowing data fetched by one processor to be used by others.

Sharing a node's resources also comes at a cost. Shared-memory semantics dictates that coherence operations appear to execute atomically [22]. By overlapping protocol execution with computation, coherence checks in the application (on one processor) may execute simultaneously with coherence operations in a protocol handler (on another). While some systems directly support atomic coherence operations in hardware (e.g., Typhoon-0 [20]), others implement these operations in a non-atomic sequence of software instructions (e.g., Blizzard-S [25] or Shasta [23]). Non-atomic coherence operations require additional synchronization and may result in low SMP-node performance.

Contention for resources in an SMP node may also lower performance. Commodity network interface cards are typically placed far from processors on a slow peripheral bus and do not provide support for multiple message queues [6,8]. As such, frequent network communication using a single pair of message queues on an SMP may result in a bottleneck [12]. Multiplexing computation and protocol execution on processors may also lead to cache interference, lower cache performance, and result in higher memory bus contention.

Besides performance, clustering processors into SMP nodes also impacts the cost trade-off. SMPs typically charge higher price premiums than uniprocessors. However, for a

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, National Science Foundation with grants MIP-9225097, MIPS-9625558, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

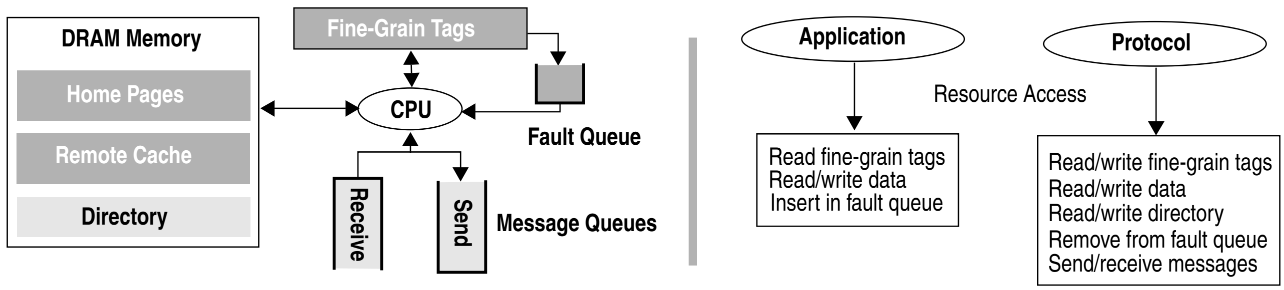

FIGURE 1. Anatomy of an FGDSM node: resources (left), application/protocol access breakdown (right).

system with a given aggregate number of processors and amount of memory, SMPs substantially reduce the networking hardware requirement by reducing the number of nodes in the system as compared to uniprocessors.

In this paper we present *Sirocco*, a family of software FGDSMs derived from Wisconsin Blizzard [25] and implemented on network of Sun SparcStation 20s interconnected by Myrinet [5]. Sirocco systems range from an all-software design to a design with minimal custom hardware support for coherence operations. We identify and evaluate the sources of overhead in SMP-node implementations of software FGDSM. We compare Sirocco's performance on SMP nodes against uniprocessor nodes for systems with a given aggregate number of processors and amount of memory. We use performance measurements running eight shared-memory applications together with simple cost models to ask the question: "are SMPs cost-effective building blocks for software FGDSM?"

Our results indicate that SMP nodes: (i) result in performance competitive with uniprocessor nodes, (ii) substantially reduce hardware requirement and are more cost-effective than uniprocessor nodes, (iii) significantly benefit from hardware support for coherence, and (iv) especially benefit systems with high-overhead coherence operations.

Our results also indicate that SMP-node performance may be highly sensitive to the protocol scheduling policy. In Sirocco, an idle processor on a node can handle protocol operations on behalf of another. Scheduling one processor to handle protocol messages for another may result in adverse cache effects in applications with bursty communication patterns.

The rest of the paper is organized as follows. Section 2 describes how Sirocco implements FGDSM on an SMP node. Section 3 qualitatively analyzes the impact of SMP nodes on FGDSM performance. Section 4 and Section 5 evaluate the performance and cost-effectiveness of the Sirocco systems respectively. Finally, Section 6 concludes the paper.

2 Sirocco: FGDSM on an SMP Node

Figure 1 illustrates the anatomy of a software FGDSM node [25]. The figure depicts the protocol-only resources in light gray, and resources used by both the protocol and the application in dark gray. The protocol maintains memory block sharing information in a directory and uses a pair of send/receive message queues to communicate with other nodes. A remote cache in memory temporarily stores fetched remote data. Shared data pages are distributed among designated home nodes. A set of fine-grain tags enforce access semantics for shared data in the remote cache and home pages. Upon a block access fault (i.e., an access violation on a shared memory block), the system inserts the relevant information into a fault queue. Processors execute both the application and the protocol software.

Sirocco extends the software FGDSM in Blizzard [25] to target small-scale SMP rather than uniprocessor nodes. Unlike other SMP-node software FGDSMs (e.g., Shasta [22]), Sirocco fully shares a node's resources among SMP processors. A single remote cache improves memory utilization by eliminating redundant copies of shared remote data. Sharing a memory cache especially benefits FGDSMs because memory caches typically suffer from page fragmentation [11]. In Sirocco, SMP processors directly share data in the remote cache and home pages using shared-memory hardware and obviate the need for intra-node messaging. Sharing memory also enables combining request messages from multiple processors for a single memory block and allows a processor to use memory blocks fetched by others. Sharing protocol resources (e.g., the directory, message queues) allows idle processors to execute protocol handlers while other processors are busy computing [12,7].

Sharing resources, however, may violate the shared-memory access semantics. Shared-memory dictates that coherence operations on data in the remote cache and home pages must appear to execute atomically [22,25]. Figure 2 illustrates examples of atomic sequences required in FGDSM coherence operations. Coherence operations either correspond to fine-grain tag lookups upon a memory load or store operation in the application, or protocol actions (e.g., a writeback request for a dirty block) which require an atomic pair of accesses to the fine-grain tags and memory.

Uniprocessor-node implementations of FGDSM [25] or SMP-node implementations that do not allow resource sharing [23] guarantee atomicity of coherence operations in three ways. First, the resources are replicated among the processors and each processor always executes its own protocol handlers. As such, an application and a protocol handler simultaneously executing on multiple processors never access the same resources. Second, protocol handlers are only invoked if there is an access violation or through polling for messages and always execute to completion. As such, protocol actions always appear to execute atomically with respect to the application. Third, FGDSM's that implement tag lookup in software (e.g., Blizzard-S [25] and Shasta [23]) carefully insert polling code to avoid handling messages in the middle of a coherence lookup.

In the rest of the section, we describe Sirocco's approach to sharing resources among SMP processors. The next section describes the protocol dispatch and execution model and how Sirocco coordinates accesses to protocol-only

Memory Operation in Application	Required Atomic Sequence
load data[address]	read tag[address] if (access is valid given tag) load data[address] else invoke an access fault

Block Invalidation in Protocol Handler	Required Atomic Sequence
read and invalidate data[address]	read data[address] set tag[address] to invalid

FIGURE 2. Examples of atomic sequences required by FGDSM coherence operations.

Coherence check on application share-memory loads and stores (left), coherent read and invalidate request in a protocol handler writing back a dirty block to the home node.

resources and the fault queue. Section 2.2 presents the alternative mechanisms in various Sirocco systems to support atomic coherence operations.

2.1 Protocol Dispatch and Execution Model

The protocol-only resources may require access coordination if multiple processors simultaneously execute protocol handlers. FGDSM protocol handlers, however, only consist of code to move a small data block between memory and the network, and update the corresponding protocol state. As such, moving network data in/out of memory often dominates a handler's execution time. Parallel handler execution is only beneficial if the network interface card provides mechanisms to efficiently transfer data blocks in/out of the network [17], and implements either multiple message queues [6] or mechanisms to efficiently dispatch messages from a single queue to multiple handlers [8]. Unfortunately, many commodity network interface cards fail to satisfy the above requirements and hence preclude efficient simultaneous execution of multiple protocol handlers.

Unlike Shasta [22], Sirocco obviates the need for synchronization around the protocol-only resources (Figure 1) by serializing handler execution. In Sirocco, processors contend for a (software) lock to assume the role of the *protocol processor* upon an access violation or a message arrival, or while waiting at a barrier synchronization. The network interface card signals a message arrival by setting a flag in a user-accessible memory location. We use executable editing [16] and instrument the application code to poll the flag on every loop-backedge. Backedge polling obviates the need for user-level message interrupts which incur prohibitively high overheads (~ 70μs) in our commodity operating system. A protocol processor always goes back to computation by releasing the lock when it no longer needs to wait—e.g., a remote block arrives or all messages are received.

Sirocco multiplexes computation with running protocol handlers on all the processors. Alternatively, the system could dedicate one processor on every node to execute protocol handlers. A recent study [12], however, concludes that a dedicated processor is not advantageous for slow commodity networking hardware and small-scale SMP nodes. While alternative scheduling policies are possible, they are beyond the scope of this paper.

Sirocco uses an array of per-processor fault records to implement the shared fault queue. Fault array accesses are of a producer-consumer nature in which the application always inserts new data and the protocol simply removes them. A simple per-processor signal flag in the fault array guarantees that fault information is correctly handed-off to the protocol.

2.2 Support for Atomic Coherence Operations

Support for atomic coherence operations depends on the fine-grain tag implementation. Much like Blizzard [25], Sirocco provides a spectrum of tag implementations including custom hardware tags in a snoopy board [20], ECC-based tags managed by the memory-controller [25], and table-based tags maintained in software [23,25]. Hardware tags perform a lookup atomically with the memory reference and eliminate overhead either entirely or for most memory references. Software tags perform a lookup in a (non-atomic) sequence of instrumented instructions and require explicit software synchronization to guarantee atomicity. Tag implementations also vary in the degree of support for atomic coherence operations in handlers. In the rest of the section, we describe in detail how various Sirocco systems support atomic coherence operations.

Sirocco-T0: Custom Board SRAM Tags

Sirocco-T0 uses the Typhoon-0 (T0) custom board [20] to snoop memory bus transactions, perform fine-grain access control tests through an SRAM lookup, and coordinate intra-node communication. T0 enforces fine-grain access semantics by asserting a bus error in response to memory transactions that incur access violations—e.g., read/write to an invalid memory block or write to a read-only block. An optimized kernel trap table delivers the bus error to the user level [26,18]. The user-level code inserts the appropriate fault information into an array of per-processor fault records which the protocol code polls on.

Sirocco-T0 supports atomic coherence operations from the protocol directly in hardware. By writing to a T0 control register, handlers can atomically read a memory block while invalidating/downgrading the corresponding tag. Upon a write to the control register, T0 updates the tag and reads the data into a handler-accessible block buffer. When placing a fetched block into memory, a handler must atomically execute a sequence of memory writes and a tag upgrade. T0 provides uncached page-mapping aliases to memory [25] to allow bypassing hardware tag lookup while writing the data. Because handlers in Sirocco always execute to completion without interruption, the non-atomic sequence of memory writes and tag upgrade appear to execute atomically with respect to the application. Any application access violations during handler execution are caught by the system and are resumed immediately after the tag update.

Sirocco-E: Error-Correcting Code (ECC)

Sirocco-E (a descendent of Blizzard-E [25]) uses deliberately incorrect error-correcting code (ECC) bits to identify invalid from read-only/read-write blocks. To distinguish read-only from read-write blocks, Sirocco-E uses virtual memory page protection to mark a page with at least one read-only block as a read-only page. Both bus errors and page protection traps use the same custom trap table as in Sirocco-T0. The kernel maintains read-write state for each block on a read-only page, detects writes to writable blocks, and directly executes the writes [25]. Writes to read-only blocks and bus errors are delivered to a user-level trap handler as in Sirocco-T0.

Sirocco-E manipulates page protection and implements ECC invalidates/downgrades in the OS using a custom system call interface. Atomicity is guaranteed by suspending memory activity on all but one of a node's processors, and through handshakes in the kernel. In Sirocco, there is a master processor on the memory bus capable of masking bus arbitration. A system call to invalidate a block issued from any processor but the master will send an interprocessor interrupt to the master. The master masks bus arbitration, reads the data into a user-accessible buffer in memory, writes incorrect ECC to memory, and releases bus arbitration. Downgrading the tag (from read-write to read-only) may involve changing the page protection and consequently a TLB shootdown. Sirocco-E performs atomic tag upgrades using uncached page-mapping aliases as in Sirocco-T0.

Sirocco-S: Software Tags

Sirocco-S stores the tags in memory and uses executable editing [16] to insert access control tests around shared-memory loads and stores. Unlike its predecessor Blizzard-S [25], Sirocco-S uses two forms of tests to detect access violations. Invalid memory is marked with a sentinel value that has a low probability of occurring in the program [24]. The most common test case uses a sequence of 3 instructions (3 cycles) to detect word and doubleword load operations to invalid memory blocks. When the test detects a sentinel, it performs a complete table lookup in order to distinguish access violations from innocent uses of the sentinel value. The rest of the memory operations (i.e., all stores and some loads) use a sequence of 5 test instructions (6 cycles) to index a tag table prior to the memory reference to detect access violations.

Unlike hardware tags, software tag table lookups use memory instructions and are not atomic with respect to data references. Sirocco-S guarantees atomicity through a software handshake between the application and the protocol handlers. The handshake augments the instrumentation with a pair of store and clear instructions to per-processor memory locations that protocol handlers poll on (Figure 3). Upon invalidating/downgrading a block, the protocol handler can safely modify the tag in advance, but must guarantee that all writes to the data from the application have completed.

Unfortunately, the handshake overhead may be high for applications with a large number of non-atomic instrumentations (i.e., all stores and some loads). Moreover, frequent handshaking with the protocol is unnecessary in applications with less frequent protocol activity. Sirocco-SB (B stands for backedge) addresses this problem and only uses a single clear instruction in loop-backedges in the application (Figure 3). Upon a tag update, a handler sets the flags for all processors and simply verifies that all processors have reached a loop backedge at least once before reading the block. Sirocco-SB reduces overhead in applications with a

Application on Processor i	Protocol Handler on Processor j
	Sirocco-S
store address into poll_flag[i]	

read tag[address]
if (access is valid given tag)
access data[address]
else
insert access, address
into fault queue
	update tag[address]

	for all processors on the node wait until (poll_flag[proc] != address)
clear poll_flag[i]	read data[address]
	Sirocco-SB

top:
:
read tag[address]
if (access is valid given tag)
access data[address]
else
insert access, address
into fault queue
:
	update tag[address]

	for all processors on the node set poll_flag[proc] = 1
	for all processors on the node wait until (poll_flag[proc] != 1)
	read data[address]
clear poll_flag[i]	
branch top	

FIGURE 3. Handshake between application and protocol handlers in Sirocco-S and Sirocco-SB.

The figure depicts the software handshake between the application (left) and protocol (right). In Sirocco-S, synchronization statements (shaded gray) consist of a store indicating the block being accessed and a clear indicating there are no accesses in progress. Sirocco-SB reduces the synchronization statements in the application to a single clear in every loop backedge.

low frequency of protocol activity while increasing the protocol waiting time in communication-intensive applications.

Upon an access violation the test code in Sirocco-S (Sirocco-SB) inserts the fault information into the fault array. To place a fetched block in memory, a handler first writes the data and then upgrades the tag. Because handlers execute to completion without interruptions, such an operation appears to execute atomically with respect to the application.

Our handshake methods in Sirocco assume a sequentially consistent memory system. Weaker memory models require fence instructions which may incur high overheads. Shasta replicates fine-grain tags among the processors and uses intra-node messaging to obviate the need for a software handshake and fence instructions on an Alpha Server [22]. Modern microprocessors, however, are using aggressive speculative techniques to provide sequentially consistent systems with performance competitive to weaker models [14]. Since these techniques also enhance performance of fence instructions in processors with weaker models, we expect our handshake methods to remain low-overhead alternatives to intra-node messaging in future systems.

Sirocco-ES: A Hybrid of ECC and Software Tags

Sirocco-ES is an attempt to take advantage of features in both Sirocco-E and Sirocco-S. Sirocco-ES uses ECC to identify invalid memory blocks and software tags to distinguish read-write from read-only blocks. In comparison to Sirocco-S, Sirocco-ES eliminates instrumentation overhead on load operations altogether, but introduces the cost of maintaining ECC tags. Compared to Sirocco-E, Sirocco-ES eliminates the use of high-overhead page protection mecha-

System	Tag Lookup (cycles) Base Cost	Tag Lookup (cycles) +SMP Overhead	Backedge (cycles) Base Cost	Backedge (cycles) +SMP Overhead	Tag Update (µs) Base Cost	Tag Update (µs) +SMP Overhead	Remote Miss (µs) Base Cost	Remote Miss (µs) +SMP Overhead
Sirocco-T0	0	0	5	0	15-21	0	61-90	4-7
Sirocco-E	0	0	5	0	16-64	0,30	85-157	8-16
Sirocco-S	3,6	0,2	5	0	7-10	< 1	56-80	4-7
Sirocco-SB		0		1				
Sirocco-ES	0,6	0,2	5	0	14-58	0,30	77-146	9-13
Sirocco-ESB		0		1				

TABLE 1. Cost of operations in uniprocessor- and SMP-node implementations of Sirocco.

The table presents the cost of coherence operations in uniprocessor nodes (Base) and the additional overheads in SMP nodes. Tag lookup corresponds to the lookup overhead for loads and stores (separated by a comma) respectively. Backedge corresponds to the poll overhead in every loop-backedge. Tag update corresponds to the overhead of validating/invalidating, or upgrading/downgrading the tags. Remote miss times correspond to roundtrip time from an access violation until resuming the access.

nisms at the cost of introducing instrumentation overhead for write operations. Much like Sirocco-SB, Sirocco-ESB implements the alternative form of application-protocol handshake.

3 Factors Affecting SMP-Node Performance

An FGDSM's performance on a network of SMPs depends on both application and system characteristics. Clustering processors into SMP nodes is beneficial if the application sharing patterns favor fast (local) SMP hardware shared-memory mechanisms over high-latency (remote) FGDSM mechanisms. An SMP node also provides the opportunity for an idle processor to overlap running protocol handlers with computation on other processors. SMP nodes, however, introduce additional overheads, which may result in lower overall performance. In this section we identify the sources of overhead in SMP-node implementations, and quantify overhead for common FGDSM operations.

We classify overhead into *correctness* and *contention* overhead. Correctness overhead corresponds to the minimum overhead associated with SMP-node implementations in the absence of contention among processors. Contention overhead refers to additional overhead due to resource sharing among multiple SMP processors.

Correctness Overhead

Table 1 depicts the cost of common FGDSM operations in a base system without SMP-node support, and the additional overhead of supporting SMP nodes. Software handshake incurs between 0 (for sentinel) to 2 (for table lookup) cycles of overhead upon tag lookup in Sirocco-S and Sirocco-ES, and 1 cycle of overhead upon loop-backedges in Sirocco-SB and Sirocco-ESB. Manipulating tags in ECC implementations may require a system call which incurs about 30 µs. SMP nodes may require an additional interprocessor interrupt (Section 2.2) for an overhead of 30 µs if the system call originates from a processor incapable of masking memory bus arbitration. The tag update overhead for software tags corresponds to the handshake cost in the handlers (Figure 3).

The table also presents remote miss times in Sirocco. The measurements correspond to minimum roundtrip miss times for a 128-block software protocol between two machine nodes. The range of miss times corresponds to the three types of remote misses: a read miss, a write miss, and an upgrade (write to a read-only block) miss. SMP nodes incur the additional overheads of passing information through a fault array and acquiring/relinquishing the protocol lock upon an access violation. In comparison, a uniprocessor-node implementation (such as Blizzard) directly calls the appropriate protocol handler upon access violation and passes the fault information through processor registers. SMP nodes on average increase roundtrip miss times by about 7-18%.

Contention Overhead

SMP nodes also incur contention overhead due to resource sharing among multiple processors. While contention for (local) memory accesses can lead to queuing delays on the memory bus, contention for (remote) memory accesses can result in queuing delays for running protocol handlers. Applications not benefiting from clustering increase the demand for protocol execution by increasing the aggregate frequency of remote misses on a node. Allowing one processor to execute protocol handlers on behalf of others may also pollute the protocol processor's cache and increase latency by requiring a cache-to-cache transfer of data between a requesting processor and the receiving protocol processor [12].

4 Performance Evaluation

In this section, we first present architectural details of our network of SMPs. Next, we present application speedups for our *base system*s which are uniprocessor-node FGDSM implementations (as in Blizzard) incurring no SMP-node overhead. We use the bases systems for performance comparisons against Sirocco in the rest of the paper. We proceed by evaluating the correctness overhead in Sirocco and finally measure the impact of clustering processors into SMP nodes on application performance.

Our platform consists of 16 Sun SPARCstation 20s running Solaris 2.4, each with up to four 66 MHz Ross HyperSPARC processors [21], and 64 MB of memory. The processors each have a unified 256 KB cache and a 50 MHz memory bus (Sun MBus) that maintains the caches coherent. The memory bus contains two slots, each accommodat-

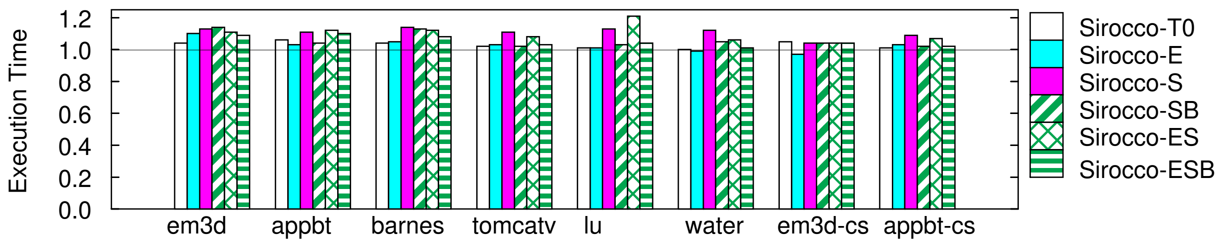

FIGURE 4. Correctness overhead in Sirocco.

The figure presents application execution times on Sirocco systems with 16 uniprocessor nodes normalized to our base systems.

Apps	Input Data Set	Sirocco Speedup			
		T0	E	S	ES
em3d	128K nodes, 40% remote edges	3.8	2.2	3.7	2.7
appbt	40x40x40 matrices	6.2	2.8	4.5	4.8
barnes	16K particles	6.1	3.8	5.3	4.7
tomcatv	512x512 matrices	10.6	8.8	6.9	9.0
lu	512x512 matrix	10.0	4.5	5.8	8.1
water	4096 molecules	12.0	9.6	8.3	10.8
em3d-cs		17.2	16.4	12.2	15.7
appbt-cs		9.9	9.8	6.2	9.7

TABLE 2. Application data sets and speedups.

The table depicts application input data sets and speedups running on our base systems. Em3d-cs and appbt-cs have similar input data sets as their transparent shared-memory counterparts (i.e., em3d and appbt). For hardware implementations, em3d-cs shows superlinear speedups due to cache effects.

ing a dual-processor module. The T0 custom board occupies one of the bus slots in Sirocco-T0 and therefore allows for only dual-processor nodes. The SMPs are interconnected using Myricom's Myrinet [5] switch-based network. Myrinet network interface cards connect to a node via a 25 MHz I/O bus. We use a 128-byte Stache software coherence protocol [19] to implement shared memory across the nodes.

4.1 Base System Performance

Table 2 presents speedups from shared-memory applications running on our base systems. We also take advantage of software DSM's flexibility, and use customized protocols that bypass shared memory and use direct messaging in two of our applications, *em3d* and *appbt* [9]. Speedups vary depending on an application's inherent parallelism, and its interaction with the FGDSM system. Sirocco-T0 implements the fine-grain tags in hardware and always achieves the best speedups. Sirocco-S always incurs instrumentation overhead and therefore favors applications with frequent access violations (e.g., *em3d)*. In contrast, protocol coherence operations in Sirocco-E are expensive and hence it favors applications with less frequent (e.g., *tomcatv* and *water)* or no (e.g., *em3d-cs* and *appbt-cs*) access violations.

Page protection overhead in Sirocco-E can degrade performance even in the absence of sharing if an application incurs frequent writes to read-only pages (i.e., pages with at least one read-only block). For instance, *lu* achieves reasonable speedups on Sirocco-T0, but exhibits a much lower performance on Sirocco-E. Sirocco-ES addresses this problem by performing tag lookups for stores in software and obviating the need for page protection. Sirocco-ES often either outperforms both Sirocco-E and Sirocco-S or performs close to the best of the two.

4.2 Correctness Overhead in SMP Nodes

We measure correctness overhead by comparing the Sirocco systems running on uniprocessor nodes against our base systems (incurring no SMP-node overhead). Figure 4 illustrates application execution times on Sirocco systems normalized to those on the corresponding base systems. On average, correctness overhead is negligible (< 3%) in hardware tags. The software tags require an application-protocol handshake and incur a higher overhead of up to 11%.

The performance impact of correctness overhead also varies across applications. In Sirocco-T0, applications with high sharing activity (e.g., *em3d*, *appbt*, *barnes*, and *em3d-cs)* incur higher correctness overhead. Correctness overhead in Sirocco-S has a higher performance impact on applications with frequent non-atomic instrumentations (e.g., *barnes* and *lu*). The loop-backedge handshake on average lowers the incurred correctness overhead in Sirocco-S (Sirocco-ES) by up to 4%.

4.3 Performance Impact of Clustering

In this section, we investigate the impact of clustering—i.e., grouping processors into SMP nodes—on application performance. We evaluate clustering by comparing Sirocco's (SMP-node) performance against that of our base system, while keeping the aggregate number of processors and amount of memory in the system constant.

Clustering affects the number of accesses to both local and remote memory. An SMP must satisfy all of the clustered processors' local memory accesses. While clustering converts certain memory accesses among neighboring (clustered) processors from remote to local, it aggregates all of the clustered processors' remote accesses. Because clustered processors share the remote cache and home pages in memory, a processor fetching remote data may also (implicitly) prefetch and convert remote accesses by others to local cache accesses.

Clustering affects performance in applications with dominant local memory accesses in three ways. First, clustered implementations at a minimum incur the SMP-node correctness overhead. Second, an increase in local accesses can introduce queuing delays in the node's memory bus. Third, executing protocol handlers on one processor on behalf of another may impact cache performance and an increase in the number of local accesses. Likewise, clustering affects

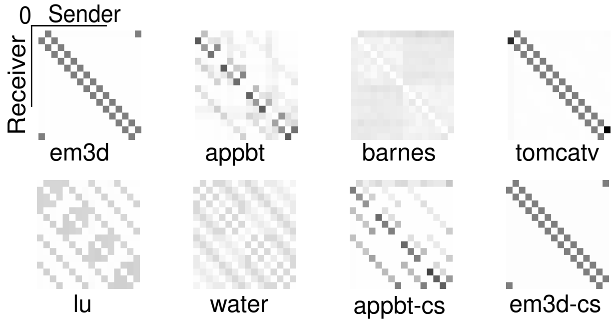

FIGURE 5. Application sharing patterns.

The figure illustrates application sharing patterns running on our base systems. The patterns also correspond to those in a clustered configuration since data partitioning in these applications is static. The shades of gray indicate message traffic intensity from senders (the x-axis) to the receivers (y-axis). Top left-most corner indicates traffic from node 0 to itself.

performance in applications with dominant remote memory access patterns in two ways. First, the aggregate remote memory accesses increase the demand for executing protocol handlers. Second, SMP-node processors can improve performance by overlapping computation with protocol execution.

The per-node aggregate number of remote accesses in a clustered configuration depends on an application's sharing patterns (Figure 5). Sharing patterns vary from strictly nearest-neighbor sharing in *em3d* and *tomcatv*, to mostly all-to-all sharing in *barnes*. Nearest-neighbor sharing results in the same per-node aggregate number of remote accesses in both clustered and uniprocessor-node configurations; on every node, there are exactly two immediate neighboring remote processors in all configurations. In more complex sharing patterns, the per-node aggregate number of remote accesses depends on the degree of sharing in the remote cache and home pages. When the network is the bottleneck, performance improvements with clustering are due to implicit prefetching of shared memory blocks, which cannot occur in applications with nearest-neighbor sharing.

Figure 6 presents application execution times on sixteen uniprocessor nodes, eight dual-processor nodes, and four quad-processor nodes for all Sirocco systems. The results are normalized to the corresponding base uniprocessor-node systems. *Tomcatv*, *lu*, *water, em3d-cs* and *appbt-cs* are computation-intensive and primarily access local data, *em3d* and *appbt* are communication-intensive and frequently access remote data, and *barnes* accesses moderate amounts of remote data. *Em3d*, *tomcatv* and *em3d-cs* all exhibit nearest-neighbor sharing patterns (Figure 5). As such, clustering does not affect the per-node aggregate number of remote accesses in these applications. *Appbt* uses shared-memory spin-locks and incurs frequent remote accesses on the critical path of execution. Clustering substantially reduces these remote accesses by converting them to local spin-lock accesses. The per-node aggregate number of remote accesses, however, increases in all the other applications (up to 50% in *barnes*).

Our overall results indicate that clustering offers competitive performance specially for hardware tag implementations. Dual-processor nodes perform very close to uniprocessor nodes for Sirocco-T0 and Sirocco-E and increase execution time on average by 13% in Sirocco-S and 11% in Sirocco-SB. Quad-processor nodes also exhibit performance competitive to uniprocessor nodes, and are especially beneficial for Sirocco-E, converting high-overhead FGDSM operations (e.g., write to read-only pages) to fast SMP local accesses. This result corroborates previous findings for (high-overhead) DVSM implementations on a network of SMPs [28]. Quad-processor nodes also increase synchronization time in the loop-backedge handshake because a protocol handler must wait for three processors to reach a loop-backedge. This result indicates that the loop-backedge handshake may be suitable for small-scale SMP nodes while instrumented synchronization may be suitable for larger SMP nodes.

Em3d-cs consistently exhibits the largest performance degradation across all tag implementations. At the end of each iteration in *em3d-cs*, the system schedules one processor (the first one to become the protocol processor) to receive all the incoming data. Because, the data are received in protocol processor's cache, subsequent accesses to the data by the consuming (i.e., computing) processor miss. Data belonging to other processors also pollute the protocol processor's cache. The combined effect significantly increases computation time in *em3d-cs* (> 50% for quad-processor nodes). This result suggests that systems should allow custom protocols to schedule protocol execution on a particular processor for effective cache utilization. Our transparent shared memory protocol does not exhibit performance sensitivity to the scheduling policy because a (protocol) processor resumes computation as soon as the block it is waiting for arrives.

Appbt and *em3d* significantly benefit from clustering across tag implementations. In *appbt,* clustering improves performance by reducing the number of remote accesses. The performance impact is more pronounced for systems with high-overhead coherence operations such as Sirocco-E and Sirocco-ES. *Appbt* performs 89% and 32% better respectively on quad-processor nodes than uniprocessor nodes. *Em3d* makes effective use of the processors' idle time to service remote requests. Because of nearest-neighbor sharing patterns, quad-processor nodes eliminate remote accesses for two of the processors. In every iteration, these processors complete computation more quickly than others and compete acting as the protocol processor on behalf of the node, overlapping protocol execution with computation and improving performance by up to 18%.

Lu exhibits irregular clustering trends on Sirocco-E. Clustered neighbors are located along the y dimension of the 2-dimensional input matrix. Partitioning the matrix among sixteen processors using uniprocessor and dual-processor nodes results in a large number of writes to read-only pages. Because protocol execution is serialized, queueing delays at the protocol processor in dual-processor nodes degrade performance. In quad-processor nodes, all the processors along the y dimension belong to the same node eliminating the read-only pages. As such, performance significantly improves by 96% since the high-overhead FGDSM accesses are converted to local SMP accesses.

Not surprisingly, the choice of a handshake method impacts the clustering performance for software tags. Frequent access violations in *em3d* (on dual-processor nodes) and *barnes,* as well as large loop bodies with sparse loop-backedges in *tomcatv* increase handshake waiting time in

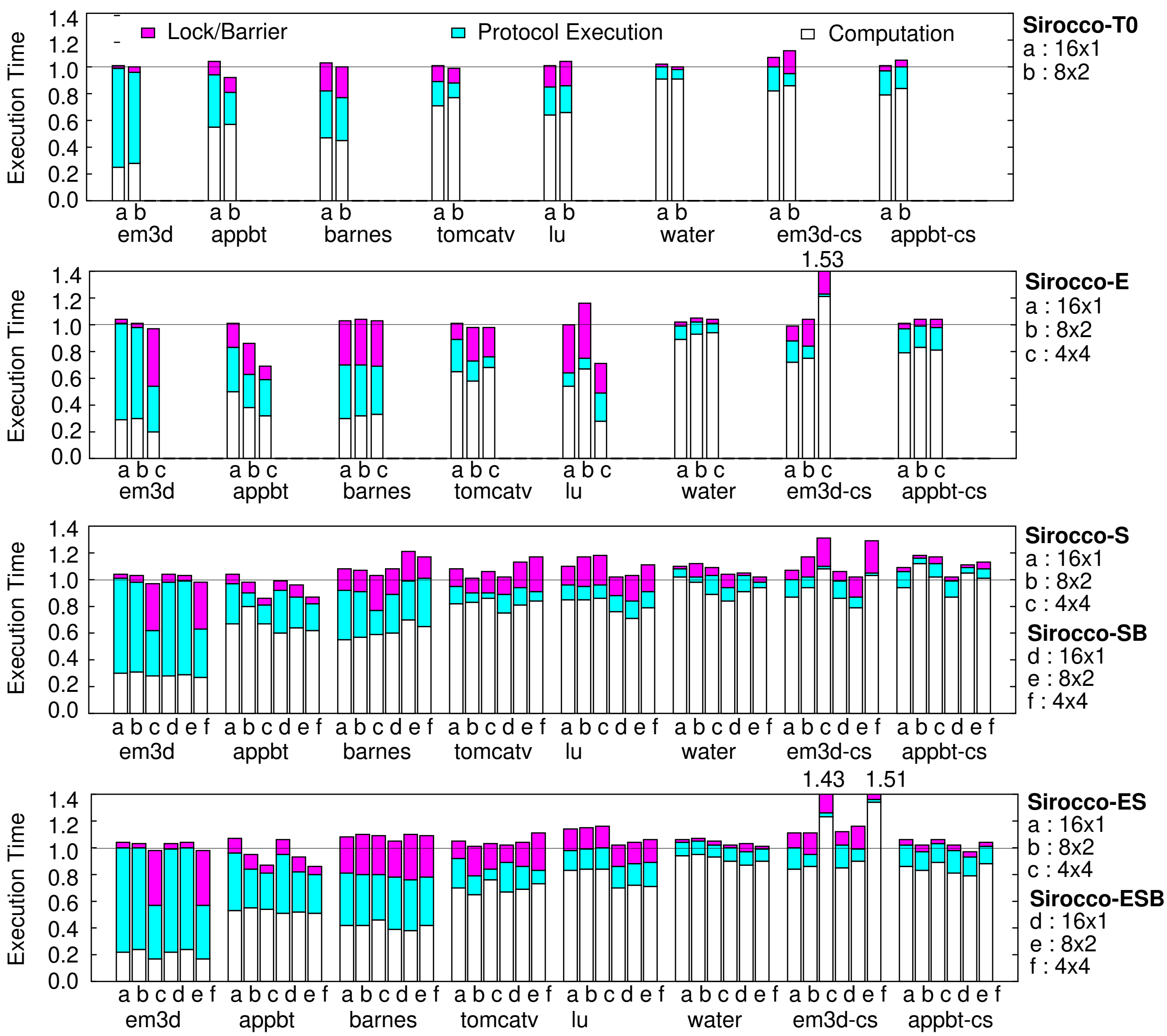

FIGURE 6. Performance of clustering in Sirocco.

The figure depicts application execution times on Sirocco-T0, Sirocco-E, Sirocco-S, and Sirocco-ES for sixteen uniprocessor nodes (16x1), eight dual-processor nodes (8x2), and four quad-processor nodes (4x4) each normalized to the corresponding base system. Sirocco-T0 supports only up to two processors per node. Sirocco-S and Sirocco-ES graphs also present execution time on the alternative loop-backedge handshake implementations, Sirocco-SB and Sirocco-ESB. The execution times are broken down into three components: (a) computation time including polling and tag lookup instrumentation overhead, (b) protocol execution time including message and block access fault handling and waiting for messages, and (c) lock/barrier waiting time.

the loop-backedge method. The latter, however, consistently improves performance for *lu* and *water* because of the low frequency of protocol activity in these applications.

5 Cost-Effectiveness of SMP Nodes

Although the manufacturing cost of computer products is typically related to the cost of components, cost from the perspective of a customer is related to price, which is also dictated by market forces [13]. High-performance products, for instance, tend to target smaller markets and as such carry larger margins, charging higher premiums.

A software FGDSM may either use uniprocessors or SMPs as building blocks. Depending on the degree of multiprocessing, SMP products can belong to either a low-margin desktop or high-margin server market. When clustering processors into SMP nodes, many machine components such as the number of processors and memory modules remain fixed across platforms. A clustered system, however, reduces the number of nodes in the machine and thus requires fewer motherboards, network interface cards, and network switches/routers. Because SMPs can carry higher price premiums, a reduction in the number of these components may be offset by an increase in the cost of a node.

In this section we ask the question: "are SMPs cost-effective building blocks for software FGDSM?" We use cost-performance as the metric [27,10], and our base uniproces-

Application	Equal Cost			DELL-based Cost			Sun-based Cost		
	16x1	8x2	4x4	16x1	8x2	4x4	16x1	8x2	4x4
em3d	1.00	1.08	**0.89**	1.00	0.92	**0.73**	1.00	0.78	**0.48**
appbt	1.00	1.03	**0.88**	1.00	0.88	**0.72**	1.00	0.75	**0.48**
barnes	**1.00**	1.11	1.04	1.00	0.95	**0.91**	1.00	0.81	**0.56**
tomcatv	**1.00**	1.02	1.07	1.00	0.87	**0.84**	1.00	0.74	**0.58**
lu	**1.00**	1.25	1.27	**1.00**	1.07	1.03	1.00	0.91	**0.69**
water	**1.00**	1.16	1.12	1.00	0.99	**0.96**	1.00	0.84	**0.61**
em3d-cs	**1.00**	1.17	1.37	1.00	1.00	**0.96**	1.00	0.85	**0.74**
appbt-cs	**1.00**	1.19	1.19	1.00	1.01	**0.98**	1.00	0.86	**0.64**

TABLE 3. Cost-effectiveness of clustering in Sirocco-S.

The table depicts cost-performance (i.e., cost times execution time) for clustered configurations normalized to that of the base system (uniprocessor-node implementation) in Sirocco-S. Numbers lower than 1 indicate a cost-effective clustered configuration. Numbers appearing in underlined bold indicate the most cost-effective configuration. The table compares cost-performance for systems with equal cost—i.e., when there is no cost advantage, and for DELL- and Sun-based systems.

sor-node FGDSM systems as the reference point. We say an SMP-node system is *cost-effective*, if it has a lower cost-performance than a uniprocessor-node system. A system is most cost-effective when it achieves the lowest cost-performance of all compared systems.

We compare and contrast uniprocessor and SMP products from two vendors representing the low and high ends of the price spectrum respectively. Vendor prices vary over time but the trends suggest that the relative prices remain the same. We evaluate cost-performance—i.e., cost multiplied by execution time—using application execution times (from Section 4.3) and cost estimates for sample systems built from DELL [1] and Sun Microsystems [3] products representing two ends of the price spectrum for desktops and servers. Our platforms consist of a total of 16 processors and 1 GB of memory and use Myrinet [2] for networking. DELL products are low-end uniprocessor Dimension XPS PCs, high-end dual-processor Model 400 workstations, and quad-processor flagship PowerEdge 6100 servers. Sun Microsystems products are single and dual-processor desktops Model E1300, and low-end quad-processor Enterprise 450 servers.

Table 3 depicts the cost-performance of Sirocco-S's clustered configurations normalized to that of its corresponding base uniprocessor-node system. Numbers lower than one indicate a cost-effective clustered configuration. Without a cost advantage (i.e., in equal-cost systems), clustering results in machines that are not cost-effective for most of the applications; clustered implementations incur higher overheads and therefore lower performance. For all vendor platforms and applications, quad-processor nodes perform best except for *lu* on DELL-based systems; *lu* exhibits a large performance degradation in a clustered system and therefore results in lower cost-performance. Sun's quad-processor configurations offer a significant cost-performance advantage over their uniprocessor and dual-processor counterparts. SMP nodes also improve cost-performance for DELL-based platforms but the high price premium of DELL's SMP products prevent them from having a high impact on cost-performance. These results, however, are based on a high-overhead clustered implementation (Sirocco-S) and may be conservative. Implementations with hardware assist will be more favorable towards quad-processor nodes.

6 Conclusions

In this paper, we presented Sirocco, a family of FGDSM systems implemented on a network of SMP workstations. Unlike previous implementations of FGDSM, Sirocco targets low-cost SMPs rather than uniprocessors as building blocks. SMP nodes provide an opportunity to improve performance by allowing processors to communicate within SMP using fast hardware shared-memory mechanisms. Multiple SMP processors can also overlap application's execution with protocol execution thereby reducing execution time. Simultaneous sharing of node's resources (e.g., memory) between the application and protocol, however, requires mechanisms for guaranteeing atomic accesses. Contention for shared resources among SMP processors may also result in queueing delays and lower performance.

We measured performance for various Sirocco implementations ranging from an all-software approach with no additional hardware to a mostly-software approach with custom hardware support for atomic coherence operations. We evaluated the impact of clustering—i.e., grouping processors into SMP nodes—by comparing the performance of clustered implementations against that of a uniprocessor-node implementation while keeping the aggregate number of processors and amount of memory in the system constant. Using simple cost models for desktop/server products, we finally asked the question "are SMPs cost-effective building blocks for software FGDSM?"

Experimental results from running shared-memory applications indicated that SMP nodes: (i) result in competitive performance with uniprocessor nodes, (ii) substantially reduce hardware requirement and are more cost-effective than uniprocessor nodes, (iii) significantly benefit from hardware support for coherence operations, and (iv) are especially beneficial for FGDSMs with high-overhead coherence operations.

Acknowledgments

We would like to thank all the members of Wisconsin Wind Tunnel project specially Eric Schnarr for helping with the software instrumentation program, Steven Reinhardt and Rob Pfile for designing and implementing the Typhoon-0 hardware, and Steve Swartz for helping with the Myrinet messaging layer. Many thanks to Alain Kãgi for his helpful comments on earlier drafts of this paper.

References

[1] Dell products price listing at www.dell.com/products.

[2] Myrinet products at www.myri.com/myrinet/product_list.html.

[3] Sun Microsystems telesale price listings at www.sun.com/sales-n-service/index.html#sales.

[4] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (networks of workstations). *IEEE Micro*, 15(1), February 1995.

[5] 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, Feb. 1995.

[6] D. Dunning and G. Regnier. The virtual interface architecture. In *Hot Interconnects V*, pages 47–58, 1997.

[7] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[8] B. Falsafi. *Fine-Grain Protocol Execution Mechanisms and Scheduling Policies on SMP Clusters*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1998.

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

[10] B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1):104–130, Jan. 1997.

[11] 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.

[12] B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, Feb. 1997.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[14] M. D. Hill. A case for making multiprocessors sequentially consistent. *IEEE Computer*, 1998. To appear.

[15] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, J. Wagner Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[17] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[18] S. K. Reinhardt, B. Falsafi, and D. A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, Sept. 1993.

[19] 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 325–337, Apr. 1994.

[20] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[21] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, Sept. 1993.

[22] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[23] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[24] I. Schoinas. *Fine-Grain Distributed Shared Memory on a Cluster of Workstations*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[25] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, Oct. 1994.

[26] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.

[27] D. A. Wood and M. D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, Feb. 1995.

[28] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A multigrain shared memory system. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.